Introduction to R for Natural Resource Scientists

Ben Staton

with contributions from Henry Hershey

Contents

Overview

This book is intended to be a first course in R programming for natural resource professionals. It is by no means comprehensive (no book about R ever could be), but instead attempts to introduce the main topics needed to get a beginner up and running with applying R to their own work. It is intended to be a companion to in-person workshop sessions, in which each chapter is covered in a 2 hour session, however it can be used as "self-teach" manual as well. Although the examples shown have a natural resource/ecological theme, the general skills presented are general to R users across all scientific disciplines.

What is Covered?

The book is composed of six chapters intended to cover a suite of topics in introductory R programming. In general, the material builds in complexity from chapter to chapter and earlier chapters can be seen as prerequisites for later chapters.

- **Chapter 1** covers the basics of working in R through RStudio, including the basics of the R coding language and environment.
- **Chapter 2** covers the basics of plotting using the base R graphics functionality.
- **Chapter 3** covers the basics of fitting statistical models using built-in functionality for generalized linear models as well as non-linear models.

- **Chapter 4** covers the basics of simulation modeling in R.
- **Chapter 5** covers the basics of the `{dplyr}` and `{reshape2}` packages for manipulating and summarizing large data sets using highly readable code.
- **Chapter 6** covers the basics of producing maps and performing spatial analysis in R. *This chapter was contributed by Henry Hershey*

Prerequisites

Chapter 1 starts at the first step (installing R) and progresses by assuming no prior knowledge of programming in R or in any other language. In the later chapters, e.g., Chapters 3 and 4, an understanding of statistics at the introductory undergraduate level would be helpful but not strictly essential.

There are, however, some tasks you'll need to complete before using this book, which are described in the two sections that follow.

Prepare Your Computer

You will install R and RStudio as the first step in Chapter 1. See here for the links to get these programs.

You should create a devoted folder on your computer for this book. All examples will assume this folder is located here: `C:/Users/YOU/Documents/R-Book`.

Change `YOU` to be specific for your computer.

Data Sets

The data sets[1] used in this book are hosted on a GitHub repository maintained by the author. It is located here: https://github.com/bstaton1/au-r-workshop-data.

To acquire the data for this book, you should:

1. Navigate to the GitHub repository
2. click the green *Clone or download* button at the top right,
3. click *Download ZIP*
4. unzip the contents of this folder into the location: `C:/Users/YOU/Documents/R-Book/Data`

Exercises

Following each chapter, there is a set of exercises. You should attempt and complete them, as they give you an opportunity to practice what you learned while reading and typing along. Solutions are provided at the end of this book, however you are **strongly** recommended to attempt to figure the problems out on your own before looking to how the author would solve them.

Some exercises have bonus questions. These are intended to challenge you with some of the more difficult tasks shown in the chapter or ask you to extend what you learned to a completely different problem. If you can get all of the non-bonus questions without looking at the solutions too much, you can consider yourself to have good understanding of that chapter's material. If you can complete the bonus questions with little or no help, that means you have mastered that chapter's material!

Text Conventions

- Regular text: a description of what you you should do, how some code works, or a general narrative of something.
- `monospace`: references something in R
  - `this()` references some function
  - `this` references some other object
  - `{this}` references an R package
  - `C:/This` is a file path
- **Bold** is intended to provide more emphasis to a word or topic. In general, new topics are introduced this way.
- Links: this is a link to some other location in this book. External links are provided with a full URL.
- *Equations*: it is sometimes useful to describe concepts mathematically before showing how to do it in R.
- [2]: a footnote containing more information.

[1] Many of the data sets used in this book were simulated by the author. Cases in which the data set used was not simulated are noted and a citation to the data source is provided. More details on the individual data sets can be found on the GitHub repository.

[2] This is a footnote

Keyboard Shortcuts

Several parts of this book in this book make reference to keyboard shortcuts. They are never necessary, but can help you be more efficient if you commit them to muscle memory. This book assumes you are using a PC for the keyboard shortcuts. If you are using a Mac, they will be different[3]. For a complete list of RStudio's keyboard shortcuts specific to your operating system, go to *Help > Keyboard Shortcuts Help*.

Development of this Book

This book represents the third reincarnation of the Auburn R Workshop Series. The first version was written in Fall 2014 using Microsoft Word, but the author found that making even small changes was clunky - each change to code in the document required a copy-paste of code and output from R to Word. Individual session materials (i.e., handout, exercises, solutions, data) were created in separate documents, saved as pdfs and `.xlsx` files, and uploaded to a wordpress webpage.

The second version was written through R and RStudio using the R packages `{rmarkdown}` (Allaire et al., 2018) and `{knitr}` (Xie, 2015), which allowed the integration of text, code, and output all into one output file. This version was completed in Fall 2015. Like the first version, individual session materials were created in separate documents, and replaced those previously found on the wordpress site.

This third version was written through R and RStudio but used the R package `{bookdown}` (Xie, 2016) which allowed for the individual sessions to be combined into one "book" by turning each session into a chapter. This facilitated cross-references to topics covered in previous chapters and allows the reader to only refer to one location when trying to remember how to use a skill. It also allowed for multiple formats to be published including both HTML and PDF versions.

The book is hosted on GitHub Pages.

About the Author

Ben Staton is a PhD candidate in the School of Fisheries at Auburn University. He studies quantitative methods for assessing fish populations for use in harvest management, with a focus on Pacific salmon in western Alaska. Ben has been using R on a daily basis since the beginning of his graduate work in 2014, and is enthusiastic about helping others learn to use R for their own work.

[3] In for some keyboard shortcuts, you may just need to swap out the **CTRL** keystroke for the **CMD** keystroke for a Mac computer

Chapter 1

Introduction to the R Environment

Chapter Overview

In this first chapter, you will get familiar with the basics of using R. You will learn:

- how to use R as a basic calculator
- some basic object types
- some basic data classes
- some basic data structures
- how to read in data
- how to produce basic data summaries
- how to write out data
- how to write your own functions

Before You Begin: Install R and RStudio

First off, you will need to get R and RStudio[1] onto your computer. Go to:

- https://cran.rstudio.com/ to get R and
- https://www.rstudio.com/products/rstudio/download/ to get RStudio Desktop.

Download the appropriate installation file for your operating system and run that file. All default settings should be fine.

1.1 The R Studio Interface

Once you open up RStudio for the first time, you will see three panes: the left hand side is the **console** where results from executed commands are printed, and the two panes on the right are for additional information to help you code more efficiently - don't worry too much about what these are at the moment. For now, focus your attention on the console.

As a matter of personal preference, you are recommended to configure a few settings. Go to *Tools* > *Global Options*, and in the section listed "General":

[1] While it is possible to run R on its own, it is clunky. You are strongly advised to use the RStudio IDE (integrated development environment) given its compactness, neat features, code tools (like syntax and parentheses highlighting). This workshop will assume you are using RStudio

- Make sure "Restore .RData into workspace at startup" is *unchecked*
- Make sure "Save workspace to .RData on exit" is set to *Never*
- Make sure "Always save history (even when not saving .RData)" is *unchecked*

These settings will prevent you from getting a bunch of useless files and dialog boxes every time you open and close R.

1.1.1 Write Some Simple Code

To start off, you will just use R as a calculator. Type these commands (not the lines with `##`, those are output[2]) one at a time and hit **CTRL + ENTER** to run it. The spaces don't matter at all, they are used here for clarity and for styling.[3]

```
3 + 3
```

```
## [1] 6
```

```
12/4
```

```
## [1] 3
```

Notice that when you run each line, it prints the command and the output to the console.

R is an **object oriented language**, which means that you fill objects with data do things with them. Make an object called `x` that stores the result of the calculation `3 + 3` (type this and run using **CTRL + ENTER**):

```
x = 3 + 3
```

Notice that running this line did not return a value as before. This is because in that line you are **assigning** a value to the object `x`. You can view the contents of `x` by typing its name alone and running just that:

```
x
```

```
## [1] 6
```

When used this way, the `=` sign denotes assignment of the value on the right-hand side to an object with the name on the left-hand side. The `<-` serves this same purpose so in this context the two are interchangeable:

```
y <- 2 + 5
```

You can highlight smaller sections of a line to run as well. For example after creating `y` above, press the **up arrow** to see the line you just ran, highlight just the `y`, and press **CTRL + ENTER**. From this point forward, the verb "run" means execute some code using **CTRL + ENTER**.

You can use your objects together to make a new object:

```
z = y - x
```

Here are some things to note about object names:

- Object names can contain any of the following:
  - letters
  - numbers
  - the `.` or `_` symbols
- Object names must start with a letter, not a number or symbol and cannot contain spaces
- As a general rule, avoid naming your objects things that already have names in R, e.g., `data()`, `mean()`, `sum()`, `sd()`, etc.

[2] The formatting used here includes `##` on output to denote code and output separately. You won't see the `##` show up in your console.

[3] To learn more about standard R code styling, check out Hadley Wickham's great chapter on it: http://adv-r.had.co.nz/Style.html.

- Capitalization matters: `A` and `a` are two different objects

1.2 Saving Your Code: Scripts

If you closed R at this moment, your work would be lost. Running code in the console like you have just done **does not save a record of your work**. To save R code, you must use what is called a **script**, which is a plain-text file with the extension `.R`. To create a new script file, go to *File > New File > R Script*, or use the keyboard shortcut **CTRL + SHIFT + N**. A new pane will open called the **source** pane - this is where you will edit your code and save your progress. R Scripts are a key feature of reproducible research with R, given that if they are well-written they can present a complete road map of your statistical analysis and workflow.

1.3 The Working Directory

Keeping things organized is essential to efficient use of R. For this book, you should have a separate subfolder for your scripts in each chapter. Create a subfolder called `C:/Users/YOU/Documents/R-Book/Chapter1` and save your `Ch1.R` script there.

Part of keeping your work organized in R is making sure you know where R is looking for your files. One way to facilitate this is to use a **working directory**. This is the location (i.e., folder) on your computer where your current R session will "talk to" by default. The working directory is where R will read files from and write files to by default. Because you'll likely be visiting it often, it should probably be somewhere that is easy to remember and not too deeply buried in your computer's file system.

To set the working directory to `C:/Users/YOU/Documents/R-Book/Chapter1`, you have three options:

1. **Go to Session > Set Working Directory > Source File Location**. This will set the working directory to the location of the file that is currently open in your source pane.
2. **Go to Session > Set Working Directory > Choose Directory**. This will open an interactive file selection window to allow you to navigate to the desired directory.
3. **Use code**. In the console, you can type `setwd("C:/Users/YOU/Documents/R-Book/Chapter1")`. If at any point you want to know where your current working directory is set to, you can either look at the top of the console pane, which shows the full path or by running `getwd()` in the console. Note the use of `/` rather than `\` for file paths in R.

The main benefits of using a working directory are:

- Files are read from and written to a consistent and predictable place every time
- Everything for your analysis is organized into one place
- You don't have to continuously type file paths to your work. If `file.txt` is a file in your current working directory, you can reference it your R session using `"file.txt"` rather than with `"C:/Users/YOU/Documents/R-Book/Chapter1/file.txt"` each time.

Note that while it is generally good practice to keep your data in the working directory, this is not recommended for this book. You will be using the same data files in multiple chapters, so it will help if they are all stored in one location. More details on this later (Section 1.8).

1.4 R Object Types

R has a variety of object types that you will need to become familiar with.

1.4.1 Functions

Much of your work in R will involve functions. A function is called using the syntax:

```
fun(arg1 = value1, arg2 = value2)
```

Here, `fun()` is the **function name** and `arg1` and `arg2` are called **arguments**. Functions take input in the form of the arguments, do some task with them, then return some output. The parentheses are a sure sign that `fun()` is a function.

We have passed the function two arguments by name: all functions have arguments, all arguments have names, and there is always a default order to the arguments. If you memorize the argument order of functions you use frequently, you don't have to specify the argument name:

```
fun(value1, value2)
```

would give the same result as the command above in which the argument names were specified.

Here's a real example:

```
print(x = z)
```

```
## [1] 1
```

The function is `print()`, the argument is `x`, and the value we have supplied the argument is the object `z`. The task that `print()` does is to print the value of `z` to the console.

R has a ton of built-in documentation to help you learn how to use a function. Take a look at the help file for the mean function. Run `?mean` in the console: a window on the right-hand side of the R Studio interface should open. The help file tells you what goes into a function and what comes out. For more complex functions it also tells you what all of the options (i.e., arguments) can do. Help files can be a bit intimidating to interpret at first, but they are all organized the same and once you learn their layout you will know where to go to find the information you're looking for.

1.4.2 Vectors

Vectors are one of the most common data structures. A vector is a set of numbers going in only one dimension. Each position in a vector is called an **element**, and the number of elements is called the **length** of the vector. Here are some ways to make some vectors with different elements, all of length five:

```
# this is a comment. R will ignore all text on a line after a #
# the ; means run everything after it on a new line

# count up by 1
month = 2:6; month
```

```
## [1] 2 3 4 5 6
```

```
# count up by 2
day = seq(from = 1, to = 9, by = 2); day
```

```
## [1] 1 3 5 7 9
```

```
# repeat the same number (repeat 2018 5 times)
year = rep(2018, 5); year
```

```
## [1] 2018 2018 2018 2018 2018
```

The `[1]` that shows up is a element position, more on this later (see Section 1.7). If you wish to know how many elements are in a vector, use `length()`:

```
length(year)
```

```
## [1] 5
```

You can also create a vector "by-hand" using the `c()` function[4]:

```
# a numeric vector
number = c(4, 7, 8, 10, 15); number
```

```
## [1]  4  7  8 10 15
```

```
# a character vector
pond = c("F11", "S28", "S30", "S8", 'S11'); pond
```

```
## [1] "F11" "S28" "S30" "S8"  "S11"
```

Note the difference between the numeric and character vectors. The the terms "numeric" and "character"" represent **data classes**, which specify the type of data the vector is holding:

- A **numeric vector** stores numbers. You can do math with numeric vectors
- A **character vector** stores what are essentially letters. You can't do math with letters. A character vector is easy to spot because the elements will be wrapped with quotes[5].

A vector can only hold one data class at a time:

```
v = c(1,2,3,"a"); v
```

```
## [1] "1" "2" "3" "a"
```

Notice how all the elements now have quotes around them. The numbers have been **coerced** to characters[6]. If you attempt to calculate the sum of your vector:

```
sum(v)
```

```
## Error in sum(v): invalid 'type' (character) of argument
```

you would find that it is impossible in its current form.

1.4.3 Matrices

Matrices act just like vectors, but they are in two dimensions, i.e., they have both rows and columns. An easy way to make a matrix is by combining vectors you have already made:

```
# combine vectors by column (each vector will become a column)
m1 = cbind(month, day, year, number); m1
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
# combine vectors by row (each vector will become a row)
m2 = rbind(month, day, year, number); m2
```

[4] The `c` stands for **concatenate**, which basically means combine many smaller objects into one larger object

[5] `" "` or `' '` both work as long as you use the same on the front and end of the element

[6] The coercion works this way because numbers can be expressed as characters, but a letter cannot be unambiguously be expressed as a number.

```
##        [,1] [,2] [,3] [,4] [,5]
## month     2    3    4    5    6
## day       1    3    5    7    9
## year   2018 2018 2018 2018 2018
## number    4    7    8   10   15
```

Just like vectors, matrices can hold only one data class (note the coercion of numbers to characters):

```
cbind(m1, pond)
```

```
##      month day year   number pond
## [1,] "2"   "1" "2018" "4"    "F11"
## [2,] "3"   "3" "2018" "7"    "S28"
## [3,] "4"   "5" "2018" "8"    "S30"
## [4,] "5"   "7" "2018" "10"   "S8"
## [5,] "6"   "9" "2018" "15"   "S11"
```

Each vector should have the same length.

1.4.4 Data Frames

Many data sets you will work with require storing different data classes in different columns, which would rule out the use of a matrix. This is where **data frames** come in:

```
df1 = data.frame(month, day, year, number, pond); df1
```

```
##   month day year number pond
## 1     2   1 2018      4  F11
## 2     3   3 2018      7  S28
## 3     4   5 2018      8  S30
## 4     5   7 2018     10   S8
## 5     6   9 2018     15  S11
```

Notice the lack of quotation marks which indicates that all variables (i.e., columns) are stored as their original data class.

It is important to know what kind of object type you are using, since R treats them differently. For example, some functions can only use a certain object type. The same holds true for data classes (numeric vs. character).You can quickly determine what kind of object you are dealing with by using the `class()` function:

```
class(day); class(pond); class(m1); class(df1)
```

```
## [1] "numeric"
```

```
## [1] "character"
```

```
## [1] "matrix"
```

```
## [1] "data.frame"
```

1.5 Factors

At this point, it is worthwhile to introduce an additional data class: factors. Notice the data class of the `pond` variable in `df1`:

```
class(df1$pond)
```

```
## [1] "factor"
```

The character vector `pond` was coerced to a factor when you placed it in the data frame. A vector with a **factor** class is like a character vector in that you see letters and that you can't do math on it. However, a factor has additional properties: in particular, it is a grouping variable. See what happens when you print the `pond` variable:

```
df1$pond
```

```
## [1] F11 S28 S30 S8  S11
## Levels: F11 S11 S28 S30 S8
```

A factor has levels, with each level being a subcategory of the factor. You can see the unique levels of your factor by running:

```
levels(df1$pond)
```

```
## [1] "F11" "S11" "S28" "S30" "S8"
```

Additionally, factor levels have an assigned order (even if the levels are totally nominal), which will become important in Chapter 3 when you learn how to fit linear models to groups of data, in which one level is the "reference" group that all other groups are compared to (see Section 3.1.2 for more details).

If you run into errors about R expecting character vectors, it may be because they are actually stored as factors. When you make a data frame, you'll often have the option to turn off the automatic factor coercion. For example:

```
data.frame(month, day, year, number, pond, stringsAsFactors = F)
read.csv("streams.csv", stringsAsFactors = F)  # see below for details on read.csv
```

will result in character vectors remaining that way as opposed to being coerced to factors. This can be preferable if you are doing many string manipulations, as character vectors are often easier to work with than factors.

1.6 Vector Math

R does vectorized calculations. This means that if supplied with two numeric vectors of equal length and a mathematical operator, R will perform the calculation on each pair of elements. For example, if you wanted to add the two vectors vector `day` and `month`, then you would just run:

```
dm = day + month; dm
```

```
## [1]  3  6  9 12 15
```

Look at the contents of both `day` and `month` again to make sure you see what R did. You typically should ensure that the vectors you are doing math with are of equal lengths.

You could do the same calculation to each element of a vector (e.g., divide each element by 2) with:

```
dm/2
```

```
## [1] 1.5 3.0 4.5 6.0 7.5
```

1.7 Data Subsets/Queries

This perhaps the most important and versatile skills to know in R. So you have an object with data in it and you want to use it for analysis. But you don't want the whole data set: just a few rows or just a few columns, or perhaps you need just a single element from a vector. This section is devoted to ways you can extract

certain parts of data objects (the terms **query** and **subset** are often used interchangeably to describe this task). There are three main methods:

1. **By Index** – This method allows you to pull out specific rows/columns by their location in an object. However, you must know exactly where in the object the desired data are. An **index** is a location of a element in a data object, like the element position or the position of a specific row or column. The syntax for subsetting a vector by index is `vector[element]` and for a matrix it is `matrix[row,column]`. Here are some examples:

```
# show all of day, then subset the third element
day; day[3]
```

```
## [1] 1 3 5 7 9
```

```
## [1] 5
```

```
# show all of m1, then subset the cell in row 1 col 4
m1; m1[1,4]
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
## number
##      4
```

```
# show all of df1, then subset the entire first column
df1; df1[,1]
```

```
##   month day year number pond
## 1     2   1 2018      4  F11
## 2     3   3 2018      7  S28
## 3     4   5 2018      8  S30
## 4     5   7 2018     10   S8
## 5     6   9 2018     15  S11
```

```
## [1] 2 3 4 5 6
```

Note this last line: the `[,1]` says "keep all the rows, but take only the first column".

Here is another example:

```
# show m1, then subset the 1st, 2nd, and 4th rows and every column
m1; m1[c(1,2,4),]
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     4   5 2018      8
## [4,]     5   7 2018     10
## [5,]     6   9 2018     15
```

```
##      month day year number
## [1,]     2   1 2018      4
## [2,]     3   3 2018      7
## [3,]     5   7 2018     10
```

Notice how you can pass a vector of row indices here to exclude the 3rd and 5th rows.

Table 1.1: The table you should enter in to R by hand for Exercise 1A

Lake	Area	Time	Fish
Big	100	1000	643
Small	25	1200	203
Square	45	1400	109
Circle	30	1600	15

2. By name – This method allows you to pull out a specific column of data based on what the column name is. Of course, the column must have a name first. The name method uses the `$` operator:

```
df1$month
```

```
## [1] 2 3 4 5 6
```

You can combine these two methods:

```
df1$month[3]
```

```
## [1] 4
```

```
# or
df1[,c("year", "month")]
```

```
##   year month
## 1 2018     2
## 2 2018     3
## 3 2018     4
## 4 2018     5
## 5 2018     6
```

The `$` method is useful because it can be used to add columns to a data frame:

```
df1$dm = df1$day + df1$month; df1
```

```
##   month day year number pond dm
## 1     2   1 2018      4  F11  3
## 2     3   3 2018      7  S28  6
## 3     4   5 2018      8  S30  9
## 4     5   7 2018     10   S8 12
## 5     6   9 2018     15  S11 15
```

3. **Logical Subsetting** – This is perhaps the most flexible method, but requires more explaining. It is described in Section 1.11.

Exercise 1A

Take a break to apply what you've learned so far to enter the data found in Table 1.1 into R by hand and do some basic data subsets.

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Create a new file in your working directory called `Ex_1A.R`.

2. Enter these data into vectors. Call the vectors whatever you would like. Should you enter the data as vectors by rows, or by columns? (*Hint: remember the properties of vectors*).
3. Combine your vectors into a data frame. Why should you use a data frame instead of a matrix?
4. Subset all of the data from Small Lake.
5. Subset the area for all of the lakes.
6. Subset the number of fish for Big and Square lakes.
7. You realize that you sampled 209 fish at Square Lake, not 109. Fix the mistake. There are two ways to do this, can you think of them both? Which do you think is better?
8. Save your script. Close R and re-open your script to see that it was saved.

1.8 Read External Data Files

It is rare that you will enter data by hand as you did in Exercise 1A. Often, you have a data set that you wish to analyze or manipulate that is stored in a spreadsheet. R has several ways to read information from data files and in this workshop, we will be using a common and simple method: reading in `.csv` files. `.csv` files are data files that separate columns with commas[7]. If your data are in a Microsoft Excel spreadsheet, you can save your spreadsheet file as a `.csv` file (*File > Save As > Save as Type> CSV (Comma Delimited)*). Several dialog boxes will open asking if you are sure you want to save it as a `.csv` file.

The syntax for reading in a `.csv` file is:

```
dat = read.csv("Path/To/FileName.csv")
```

Make sure your working directory is set to the location of your current file, which should be in the location `C:/Users/YOU/Documents/R-Book/Chapter 1`. According the instructions on acquiring the data, you should have placed all of the data files for this book (downloaded from the GitHub repository) in the location `C:/Users/YOU/Documents/R-Book/Data`. If you have done this already and your working directory is set to the location of the `Ch1.R` script, you can simply run:

```
dat = read.csv("../Data/streams.csv")
```

The `../` tells R to look up one directory from the working directory for a folder called `Data`, then within that, look for a file called `streams.csv`.

If you do not get an error, congratulations! However, if you get an error that looks like this:

```
## Warning in file(file, "rt"): cannot open file 'streams.csv': No such file
## or directory

## Error in file(file, "rt"): cannot open the connection
```

then fear not. This must be among the most common errors encountered by R users world-wide. It simply means the file you told R to look for doesn't exist where you told R to find it. Here is a trouble-shooting guide to this error:

1. The exact case and spelling matters, as do the quotes and `.csv` at the end. Ensure the file name is typed correctly.
2. Check what files are in the path you are specifying: run `dir("../Data")`. This will return a vector with the names of the files located in the `Data` directory (which is one folder up from your working directory). Is the file you told R was there truly in there? Is your working directory set to where you thought it was?

[7] Note that if your computer is configured for a Spanish-speaking country, Microsoft Excel might convert decimals to commas. This can really mess with reading in data - I would suggest changing the language of Excel if you find this to be the case.

A simpler method is to put the data file in your working directory, in which case it can be read into R using `dat = read.csv("streams.csv")`. This method is not recommended for this book, because you will be using the same data files in multiple chapters, and it will help if they are all located in the same location.

If you did not get any errors, then the data are in the object you named (`dat`) and that object is a data frame. Do not proceed until you are able to get `read.csv` to run successfully.

A few things to note about reading in `.csv` files:

- R will assume the first row are column headers by default.
- If there is a space in one of the header cells, a `"."` will be inserted. For example, the column header `Total Length` would become `Total.Length`.
- R brings in `.csv` files in as data frames by default.
- If a record (i.e., cell) is truly missing and you want R to treat it that way (i.e., as an `NA`), you have three options:
  - Hard code an `NA` into that cell in Excel
  - Leave that cell completely empty
  - Enter in some other character (e.g., `"."`) alone in all cells that are meant to be coded as `NA` in R and use the `na.strings = "."` argument of `read.csv()`.
- If at some point you did "Clear Contents" in Microsoft Excel to delete rows or columns from your `.csv` file, these "deleted" rows/columns will be read in as all `NAs`, which can be annoying. To remove this problem, open the `.csv` file in Excel, then highlight and **delete** the rows/columns in question and save the file. Read it back into R again using `read.csv()`.
- If even a single character is found in a numeric column in `FileName.csv`, the *entire column* will be coerced to a character/factor data class after it is read in (i.e., no more math with data on that column until you remove the character). A common error is to have a `#VALUE!` record left over from an invalid Excel function result. You must remove all of these occurrences in order to use that column as numeric. Characters include anything other than a number ([0-9]) and a period when used as a decimal. None of these characters: `!?[]\/@#$%^&*()<>_-+=[a-z];[A-Z]` should never be found in a column you wish to do math with (e.g., take the mean of that column). **This is an incredibly common problem!**

1.9 Explore the Data Set

Have a look at the data. You could just run `dat` to view the contents of the object, but it will show the whole thing, which may be undesirable if the data set is large. To view the first handful of rows, run `head(dat)` or the last handful of rows with `tail(dat)`.

You will now use some basic functions to explore the simulated streams data before any analysis. The `summary()` function is very useful for getting a coarse look at how R has interpreted the data frame:

```
summary(dat)
```

```
##        state    stream_width        flow
##  Alabama  :5   Min.   :17.65   Min.   : 28.75
##  Florida  :5   1st Qu.:46.09   1st Qu.: 65.50
##  Georgia  :5   Median :61.80   Median : 95.64
##  Tennessee:5   Mean   :60.88   Mean   : 91.49
##                3rd Qu.:79.34   3rd Qu.:120.55
##                Max.   :94.65   Max.   :149.54
##                                NA's   :1
```

You can see the spread of the numeric data and see the different levels of the factor (`state`) as well as how many records belong to each level. Note that there is one `NA` in the variable called `flow`.

To count the number of elements in a variable (or any vector), remember the `length()` function:

```
length(dat$stream_width)
```

```
## [1] 20
```

Note that R counts missing values as elements as well:

```
length(dat$flow)
```

```
## [1] 20
```

To get the dimensions of an object with more than one dimension (i.e., a data frame or matrix) we can use the `dim()` function. This returns a vector with two elements: the first number is the number of rows and the second is the number of columns. If you only want one of these, use the `nrow()` or `ncol()` functions (but remember, only for objects with more than one dimension; vectors don't have rows or columns!).

```
dim(dat); nrow(dat); ncol(dat)
```

```
## [1] 20  3
```

```
## [1] 20
```

```
## [1] 3
```

You can extract the names of the variables (i.e., columns) in the data frame using `colnames()`:

```
colnames(dat)
```

```
## [1] "state"        "stream_width" "flow"
```

Calculate the mean of all the `stream_width` records:

```
mean(dat$stream_width)
```

```
## [1] 60.8845
```

Calculate the mean of all of the `flow` records:

```
mean(dat$flow)
```

```
## [1] NA
```

`mean()` returned an `NA` because there is an `NA` in the data for this variable. The way to tell R to ignore this `NA` is by including the argument `na.rm = TRUE` in the `mean()` function (separate arguments are always separated by commas). This is a **logical** argument, meaning that it asks a question. It says "do you want to remove `NA`s before calculating the mean?" `TRUE` means "yes" and `FALSE` means "no." `TRUE` and `FALSE` can be abbreviated as `T` and `F`, respectively. Many of R's functions have the `na.rm` argument (e.g. `mean()`, `sd()`, `var()`, `min()`, `max()`, `sum()`, etc. - most anything that collapses a vector into one number).

```
mean(dat$flow, na.rm = T)
```

```
## [1] 91.49053
```

which is the same as (i.e., the definition of the mean with the `NA` removed):

```
sum(dat$flow, na.rm = T)/(nrow(dat) - 1)
```

```
## [1] 91.49053
```

What if you need apply a function to more than one variable at a time? One of the easiest ways to do this (though as with most things in R, there are many) is by using the `apply()` function. This function applies the same summary function to individual subsets of a data object at a time then returns the individual summaries all at once:

```
apply(dat[,c("stream_width", "flow")], 2, FUN = var, na.rm = T)
```

```
## stream_width         flow
##     581.1693    1337.3853
```

The first argument is the data object you want to apply the function to. The second argument (the number 2) specifies that you want to apply the function to columns, `1` would tell R to apply it to rows. The `FUN` argument specifies what function you wish to apply to each of the columns; here you are calculating the variance which takes the `na.rm = T` argument. This use of `apply()` alone is very powerful and can help you get around having to write the dreaded `for()` loop (introduced in Chapter 4).

There is a whole family of `-apply()` functions, the base `apply()` is the most basic but a more sophisticated one is `tapply()`, which applies a function based on some grouping variable (a factor). Calculate the mean stream width **separated by state**:

```
tapply(dat$stream_width, dat$state, mean)
```

```
##   Alabama   Florida   Georgia Tennessee
##    53.664    63.588    54.996    71.290
```

The first argument is the variable you want to apply the `mean` function to, the second is the grouping variable, and the third is what function you wish to apply. Try to commit this command to memory given this is a pretty common task.

If you want a data frame as output, you can use the `aggregate` function to do the same thing:

```
aggregate(dat$stream_width, by = list(state = dat$state), mean)
```

```
##       state      x
## 1   Alabama 53.664
## 2   Florida 63.588
## 3   Georgia 54.996
## 4 Tennessee 71.290
```

1.10 Logical/Boolean Operators

To be an efficient and capable programmer in any language, you will need to become familiar with how to implement numerical logic, i.e., the Boolean operators. These are very useful because they always return a `TRUE` or a `FALSE`, off of which program-based decisions can be made (e.g., whether to operate a given subroutine, whether to keep certain rows, whether to print the output, etc.).

Define a simple object: `x = 5` . Note that this will write over what was previously stored in the object `x`. We wish to ask some questions of the new `x`, and the answer will be printed to the console as a `TRUE` for "yes" and a `FALSE` for "no". Below are the common Boolean operators.

Equality

To ask if `x` is exactly equal to 5, you run:

```
x == 5
```

```
## [1] TRUE
```

Note the use of the double `==` to denote equality as opposed to the single `=` as used in assignment (`x = 5`).

Inequalities

To ask if `x` is not equal to 5, you run:

```
x != 5
```

```
## [1] FALSE
```

To ask if x is less than 5, you run:

```
x < 5
```

```
## [1] FALSE
```

To ask if x is less than *or equal to* 5, you run:

```
x <= 5
```

```
## [1] TRUE
```

Greater than works the same way, though with the > symbol replaced.

In

Suppose you want to ask which elements of one vector are also found within another vector:

```
y = c(1,2,3)
x %in% y
```

```
## [1] FALSE
```

```
# or
y = c(4,5,6)
x %in% y
```

```
## [1] TRUE
```

```
# or
y %in% x
```

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

%in% is a very useful operator in R that is not one of the traditional Boolean operators.

And

Suppose you have two conditions, and you want to know if **both are met**. For this you would use **and** by running:

```
x > 4 & x < 6
```

```
## [1] TRUE
```

which asks if x is between 4 and 6.

Or

Suppose you have two conditions, and you want to know if **either are met**. For this you would use **or** by running:

```
x <= 5 | x > 5
```

```
## [1] TRUE
```

which asks if x is less than or equal to 5 **or** greater than 5 - you would be hard-pressed to find a real number that did not meet these conditions!

1.11 Logical Subsetting

A critical use of logical/Boolean operators is in the subsetting of data objects. You can use a logical vector (i.e., one made of only `TRUE` and `FALSE` elements) to tell R to extract only those elements corresponding to the `TRUE` records. For example:

```
# here's logical vector: TRUE everywhere condition met
dat$stream_width > 60
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
## [12] FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE
```

```
# insert it to see only the flows for the TRUE elmements
dat$flow[dat$stream_width > 60]
```

```
##  [1] 120.48 123.78  95.64  95.82 120.06 135.63 120.61 111.34 149.54 131.22
```

gives all of the `flow` values for which `stream_width` is greater than 60.

To extract all of the data from Alabama, you would run:

```
dat[dat$state == "Alabama",]
```

```
##     state stream_width   flow
## 1 Alabama        81.68 120.48
## 2 Alabama        57.76  85.90
## 3 Alabama        48.32  73.38
## 4 Alabama        31.63  46.55
## 5 Alabama        48.93  80.91
```

You will be frequently revisiting this skill throughout the workshop.

1.12 `if()`, `else`, and `ifelse()`

You can tell R to do something if the result of a question is `TRUE`. This is a typical if-then statement:

```
if (x == 5) print("x is equal to 5")
```

```
## [1] "x is equal to 5"
```

This says "if x equals 5, then print the phrase 'x is equal to 5' to the console". If the logical returns a `FALSE`, then this command does nothing. To see this, change the `==` to a `!=` and re-run:

```
if (x != 5) print("x is equal to 5")
```

We can tell R to do multiple things if the logical is `TRUE` by using curly braces:

```
if (x == 5) {
  print("x is equal to 5")
  print("you dummy, x is supposed to be 6")
}
```

```
## [1] "x is equal to 5"
## [1] "you dummy, x is supposed to be 6"
```

You can always use curly braces to extend code across multiple lines whereas it may have been intended to go on one line.

If you want R to do something if the logical is `FALSE`, you would use the `else` command:

```
if (x > 5) print("x is greater than 5") else print("x is not greater than 5")
```

```
## [1] "x is not greater than 5"
```

Or extend this same thing to multiple lines:

```
if (x > 5) {
  print("x is greater than 5")
} else {
  print("x is not greater than 5")
}
```

The `if()` function is useful, but it can only respond to one question at a time. If you supply it with a vector of length greater than 1, it will give a warning:

```
# vector from -5 to 5, excluding zero
xs = c(-5:-1, 1:5)

# attempt a logical decision
if (xs < 0) print("negative") else print("positive")
```

```
## Warning in if (xs < 0) print("negative") else print("positive"): the
## condition has length > 1 and only the first element will be used
```

```
## [1] "negative"
```

Warnings are different than errors in that something still happens, but it tells you that it might not be what you wanted, whereas an error stops R altogether. In short, this warning is telling you that you passed `if()` a logical vector with more than 1 element, and that it can only use one element so it's picking the first one. Because the first element of `xs` is -5, `xs < 0` evaluated to `TRUE`, and you got a `"negative"` printed along with the warning.

To respond to multiple questions at once, you must use `ifelse()`. This function is similar, but it combines the `if()` and `else` syntax into one useful function function:

```
ifelse(xs > 0, "positive", "negative")
```

```
##  [1] "negative" "negative" "negative" "negative" "negative" "positive"
##  [7] "positive" "positive" "positive" "positive"
```

The syntax is `ifelse(condition, do_if_TRUE, do_if_FALSE)`. You can `cbind()` the output with `xs` to verify it worked:

```
cbind(
  xs,
  ifelse(xs > 0, "positive", "negative")
)
```

```
##       xs
##  [1,] "-5" "negative"
##  [2,] "-4" "negative"
##  [3,] "-3" "negative"
##  [4,] "-2" "negative"
##  [5,] "-1" "negative"
##  [6,] "1"  "positive"
##  [7,] "2"  "positive"
```

```
##  [8,] "3"  "positive"
##  [9,] "4"  "positive"
## [10,] "5"  "positive"
```

Use `ifelse()` to create a new variable in `dat` that indicates whether a stream is big or small depending whether `stream_width` is greater or less than 50:

```
dat$size_cat = ifelse(dat$stream_width > 50, "big", "small"); head(dat)
```

```
##     state stream_width   flow size_cat
## 1 Alabama        81.68 120.48      big
## 2 Alabama        57.76  85.90      big
## 3 Alabama        48.32  73.38    small
## 4 Alabama        31.63  46.55    small
## 5 Alabama        48.93  80.91    small
## 6 Georgia        39.42  57.63    small
```

This says "make a new variable in the data frame `dat` called `size_cat` and assign each row a 'big' if `stream_width` is greater than 50 and a 'small' if less than 50".

One neat thing about `ifelse()` is that you can nest multiple statements inside another[8]. What if you wanted three categories: 'small', 'medium', and 'large'?

```
dat$size_cat_fine = ifelse(dat$stream_width <= 40, "small",
                           ifelse(dat$stream_width > 40 & dat$stream_width <= 70, "medium", "big")); hea
```

```
##     state stream_width   flow size_cat size_cat_fine
## 1 Alabama        81.68 120.48      big           big
## 2 Alabama        57.76  85.90      big        medium
## 3 Alabama        48.32  73.38    small        medium
## 4 Alabama        31.63  46.55    small         small
## 5 Alabama        48.93  80.91    small        medium
## 6 Georgia        39.42  57.63    small         small
```

If the first condition is `TRUE`, then it will give that row a "small". If not, it will start another `ifelse()` to ask if the `stream_width` is greater than 40 *and* less than or equal to 70. If so, it will give it a "medium", if not it will get a "big". Not all function nesting examples are this complex, but this is a neat example. Without `ifelse()`, you would have to use as many `if()` statements as there are elements in `dat$stream_width`.

1.13 Writing Output Files

1.13.1 `.csv` Files

Now that you have made some new variables in your data frame, you may want to save this work in the form of a new `.csv` file. To do this, you can use the `write.csv` function:

```
write.csv(dat, "updated_streams.csv", row.names = F)
```

The first argument is the data frame (or matrix) to write, the second is what you want to call it (don't forget the `.csv`!), and `row.names = F` tells R to not include the row names (because they are just numbers in this case). R puts the file in your working directory unless you tell it otherwise. To put it somewhere else, type in the path with the new file name at the end (e.g., `C:/Users/YOU/Documents/R-Book/Data/updated_streams.csv`.

[8] You can nest **ALL** R functions, by the way.

1.13.2 Saving R Objects

If all you care about is the data frame `dat` as interpreted by R (not a share-able file like the `.csv` method), then you can save the object `dat` (in its current state) then load it in to a future R session. You can save the new data frame using:

```
save(dat, file = "updated_streams")
```

Then try removing the `dat` object from your current session (`rm(dat)`) and loading it back in using:

```
rm(dat); head(dat)  # should give error
load(file = "updated_streams")
head(dat) # should show first 6 rows
```

1.14 User-Defined Functions

Sometimes you may want R to carry out a specific task, but there is no built-in function to do it. In these cases, you can write your own functions. Function writing makes R incredibly flexible, though you will only get a small taste of this topic here. You will see more examples in later Chapters, particularly in Chapter 4.

First, you must think of a name for your function (e.g., `myfun`). Then, you specify that you want the object `myfun()` to be a function by using using the `function()` function. Then, in parentheses, you specify any arguments that you want to use within the function to carry out the specific task. Open and closed curly braces specify the start and end of your function body, i.e., the code that specifies how it uses the arguments to do its job.

Here's the general syntax for specifying your own function:

```
myfun = function(arg1) {
  # function body goes here
    # use arg1 to do something

  # return something as last step
}
```

As an example, write a general function to take any number `x` to any power `y`:

```
power = function(x, y){
  x^y
}
```

After typing and running the function code (`power()` is an object that must be assigned), try using it:

```
power(x = 5, y = 3)
```

```
## [1] 125
```

Remember, you can nest or embed functions:

```
power(power(5,2),2)
```

```
## [1] 625
```

This is the equivalent of $(5^2)^2$.

Exercise 1B

In this exercise, you will be using what you learned in Chapter 1 to summarize data from a hypothetical pond experiment.

Pretend that you added nutrients to mesocosoms and counted the densities of four different zooplankton taxa. In this experiment, there were two ponds, two treatments per pond, and five replicates of each treatment. The data are located in the data file `ponds.csv` (see the instructions on acquiring and organizing the data files for this book. There is one error in the data set. After you download the data and place it in the appropriate directory, make sure you open this file and fix it *before* you bring it into R. Refer back to the information about reading in data (Section 1.8) to make sure you find the error.

Create a new R script in your working directory for this chapter called `Ex1B.R` and use that script to complete this exercise.

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Read in the data to R and assign it to an object.
2. Calculate some basic summary statistics of your data using the `summary()` function.
3. Calculate the mean chlorophyll *a* for each pond (*Hint: pond is a grouping variable*)
4. Calculate the mean number of *Chaoborus* for each treatment in each pond using `tapply()`. (*Hint: You can group by two variables with:* `tapply(dat$var, list(dat$grp1, dat$grp2), fun)`.
5. Use the more general `apply()` function to calculate the variance for each zooplankton taxa found only in pond S-28.
6. Create a new variable called `prod` in the data frame that represents the quantity of chlorophyll *a* in each replicate. If the chlorophyll *a* in the replicate is greater than 30 give it a "high", otherwise give it a "low". (*Hint: are you asking R to respond to one question or multiple questions? How should this change the strategy you use?*)

Exercise 1B Bonus

1. Use `?table` to figure out how you can use `table()` to count how many observations of high and low there were in each treatment (*Hint:* `table` *will have only two arguments.*).
2. Create a new function called `product()` that multiplies any two numbers you specify.
3. Modify your function to print a message to the console and return the value `if()` it meets a condition and to print another message and not return the value if it doesn't.

Chapter 2

Base R Plotting Basics

Chapter Overview

In this chapter, you will get familiar with the basics of using R for making plots and figures. You will learn:

- how ot make various plots including:
  - scatterplots
  - line plots
  - bar plots
  - box-and-whisker plots
  - histograms
- the basics of how to change plot features like the text displayed, the size of points, and the type of points
- the basics of multi-panel plotting
- the basics of the `par` function
- how to save your plot to an external file

R's base `{graphics}` plotting package is incredibly versatile, and as you will see, it doesn't take much to get started making professional-looking graphs. It is worth mentioning that there are other R packages[1] for plotting (e.g. `{ggplot2}` and `{lattice}`) that have nice features. They can be more complex to learn at first than the base R plotting capabilities and look a bit different.

IMPORTANT NOTE: If you did not attend the sessions corresponding to Chapter 1, you are recommended to walk through the material found in that chapter before proceeding to this material. Also note that if you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this book.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch2.R` and save it in the directory `C:/Users/YOU/Documents/R-Book/Chapter2`. Set your working directory to that location. Revisit Sections 1.2 and 1.3 for more details on these steps.

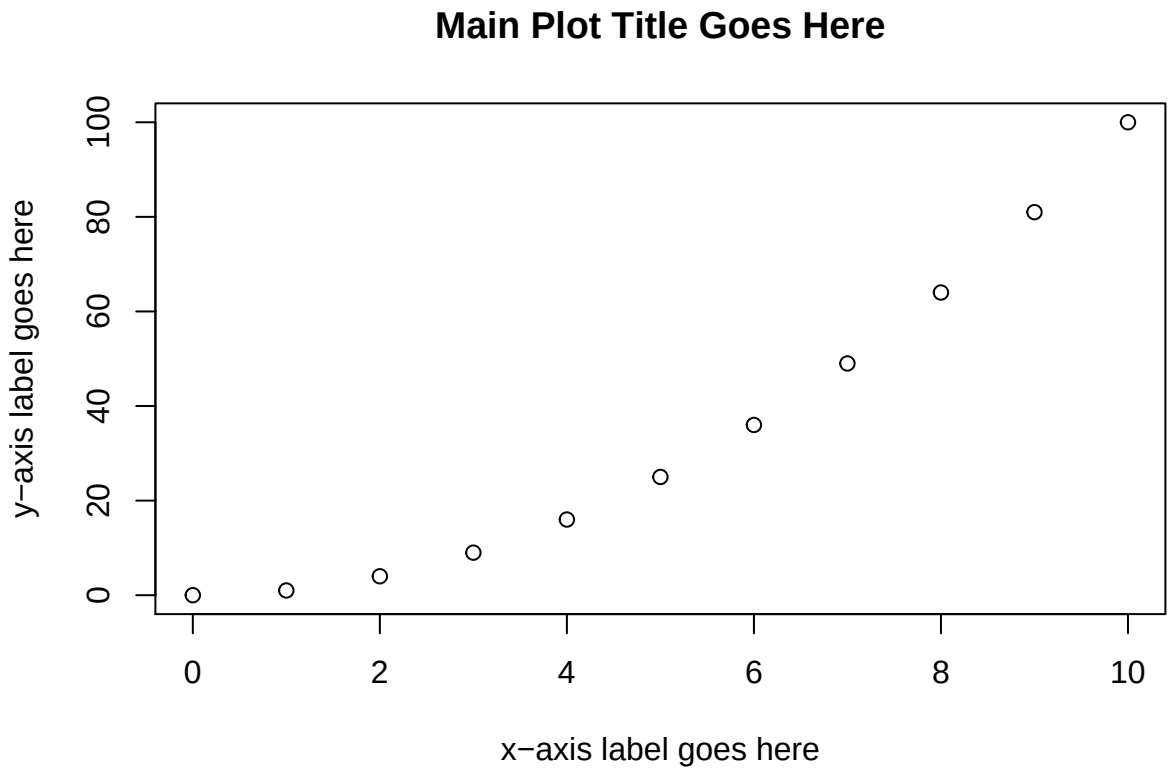
[1] An **R package** is a bunch of code that somebody has written and placed on the web for you to install, their use is first introduced in Chapter 5

2.1 R Plotting Lingo

Learning some terminology will help you get used to the base R graphics system:

- A **high-level plotting function** is one that is used to make a new graph. Examples of higher level plotting functions are the general `plot()` and the `barplot()` functions. When-a high level plotting function is executed, the currently displayed plot (if any) is written over.
- A **low-level plotting function** is one that is used to modify a plot that has already been created. You must already have made a plot using a higher level plotting function before you use lower level plotting functions. Examples include `text()`, `points()`, and `lines()`. When a low-level plotting function is executed, the output is simply added to an existing plot.
- The **graphics device** is the area that a plot shows up. RStudio has a built in graphics device in its interface (lower right by default in the "Plots" tab), or you can create a new device. R will plot on the active device, which is the most recently created device. There can only be one active device at a time.

Here is an example of a basic R plot:



There are a few components:

- The **plotting region**: all data information is displayed here.
- The **margin**: where axis labels, tick marks, and main plot titles are located
- The **outer margin**: by default, there is no outer margin. You can add one if you want to add text here or make more room around the edges.

You can change just about everything there is about this plot to suit your tastes. Duplicate this plot, but make the x-axis, y-axis, and main titles something other than the placeholders shown here:

Table 2.1: Several of the key arguments to high- and low-level plotting functions

Argument	Usage	Description
xlab	xlab = "X-AXIS"	changes the x-axis label text
ylab	ylab = "Y-AXIS"	changes the y-axis label text
main	main = "TITLE"	changes the main title text
cex	cex = 1.5	changes the size of symbols in the plotting region
pch	pch = 17	changes the symbol type
xlim	xlim = range(x)	changes the endpoints (limits) of the x-axis
ylim	ylim = c(0,1)	same as xlim, but for the y-axis
type	type = "l"	changes the way points are connected by lines
lty	lty = 2	changes the line type
lwd	lwd = 2	changes the line width
col	col = "blue"	changes the color of plotted objects

[a] cex is a multiplier: cex = 1.5 says make the points 1.5 times as large as they would be by default.
[b] There are approximately 20 different pch settings: pch = 1 is empty circles, pch = 16 is filled circles, etc.
[c] xlim and ylim both require a numeric vector of length 2 where neither of the elements may be an NA.
[d] The default is points only, type = "l" is for lines only, type = "o" is for points connected with lines, and type = "b" is for points and lines but with a small amount of separation between them.
[e] lty = 1 is solid, lty = 2 is dashed, lty = 3 is dotted, etc. You can also specific it like lty = "solid", lty = "dotted", or lty = "dotdash".
[f] works just like cex: lwd = 3 codes for a line that is 3 times as thick as it would normally be
[g] there is a whole host of colors you can pass R by name, run colors() to see for yourself

```
# plot dummy data
x = 0:10
y = x^2
plot(x = x, y = y,
     xlab = "x-axis label goes here",
     ylab = "y-axis label goes here",
     main = "Main Plot Title Goes Here")
```

Note that the first two arguments, `x` and `y`, specify the coordinates of the points (i.e., the first point is placed at coordinates `x[1]`,`y[1]`). `plot()` has *tons* of arguments (or *graphical parameters* as the help file found using `?plot` or `?par` calls them) that change how the plot looks. Note that when you want something displayed verbatim on the plotting device, you must wrap that code in `" "`, i.e., the arguments `xlab`, `ylab`, and `main` all receive a character vector of length 1 as input.

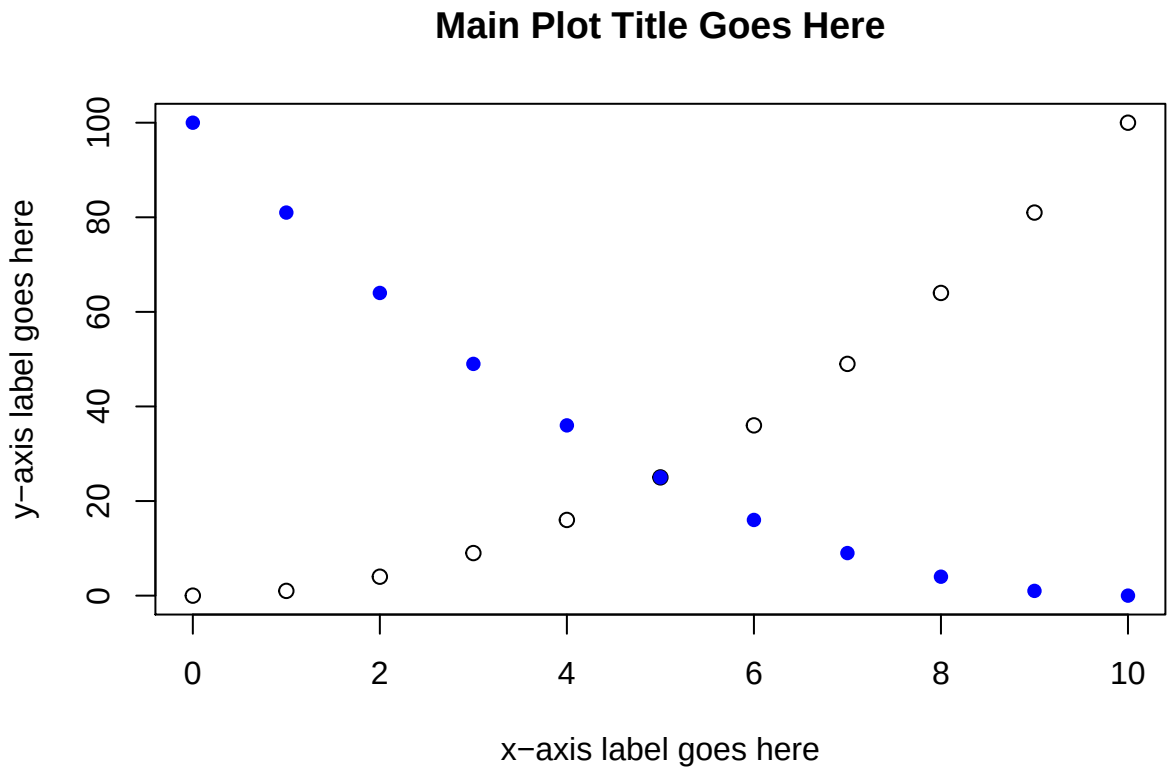
Table 2.1 shows information on just a handful of them to get you started.

You are advised to try at least some of the arguments in Table 2.1 out for yourself with your `plot(x, y)` code from above - notice how every time you run `plot()`, a whole new plot is created, not just the thing you changed. There are definitely other options: check out `?plot` or `?par` for more details.

2.2 Lower Level Plotting Functions

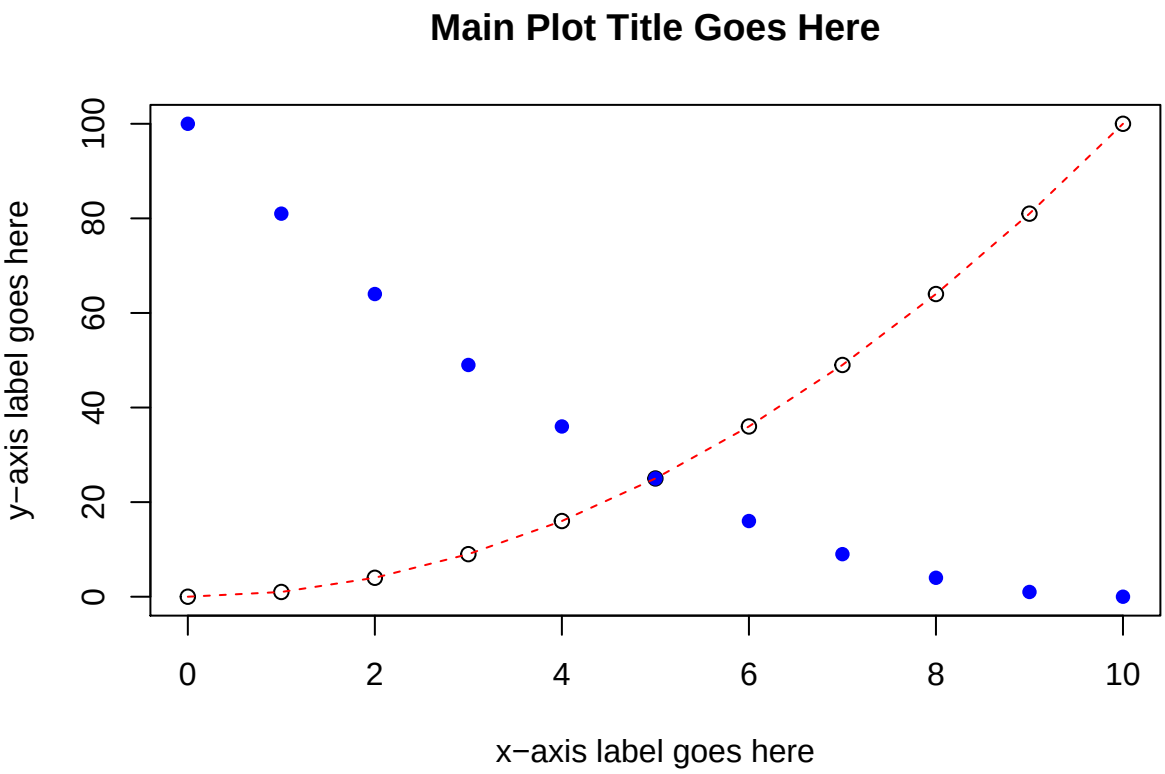
Now that you have a base plot designed to your liking, you might want to add some additional "layers" to it to represent more data or other kind of information like an additional label or text. Add some more points to your plot by putting this line right beneath your `plot(x,y)` code and run just the `points()` line (make sure your device is showing a plot first):

```
# rev() reverses a vector: so the old x[1] is x[11] now
points(x = rev(x), y = y, col = "blue", pch = 16)
```



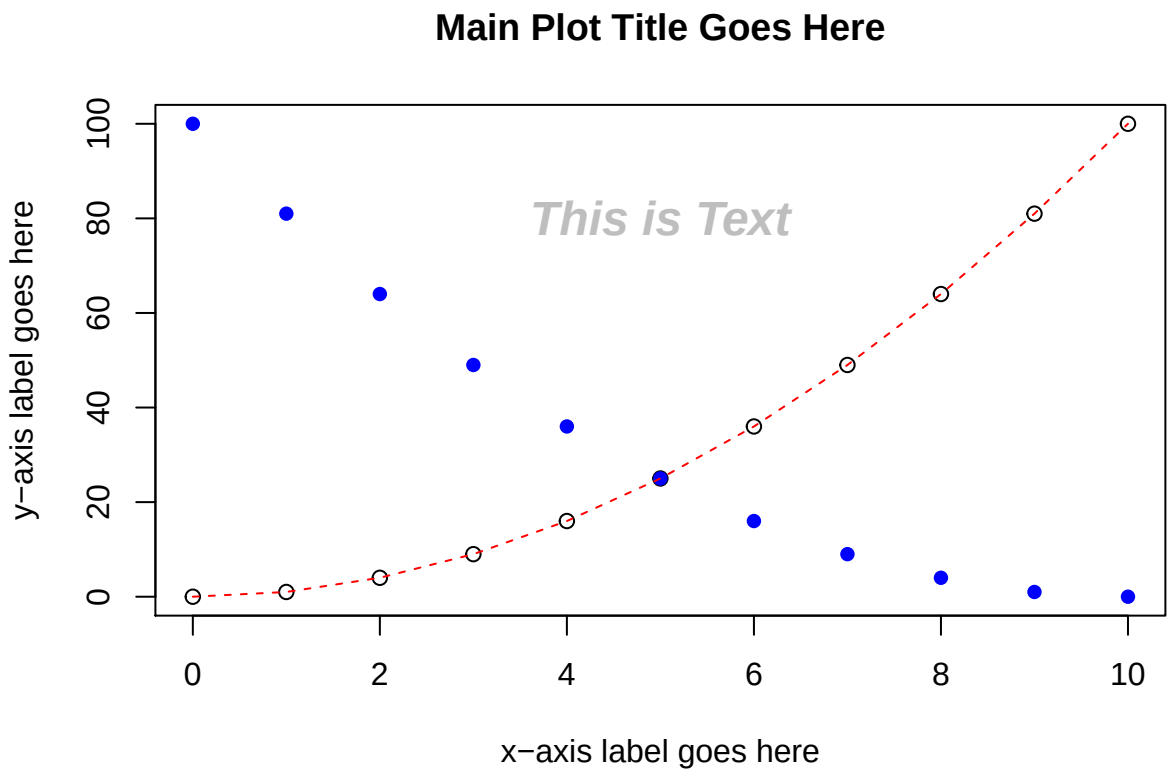
Here, `points()` acted like a low-level plotting function because it added points to a plot you already made. Many of the arguments shown in Table 2.1 can be used in both high-level and low-level plotting functions (notice how `col` and `pch` were used in `points()`). Just like `points()`, there is also `lines()`:

```
lines(x = x, y = x, lty = 2, col = "red")
```



You can add text to the plotting region:

```
text(x = 5, y = 80, "This is Text", cex = 1.5, col = "grey", font = 4)
```

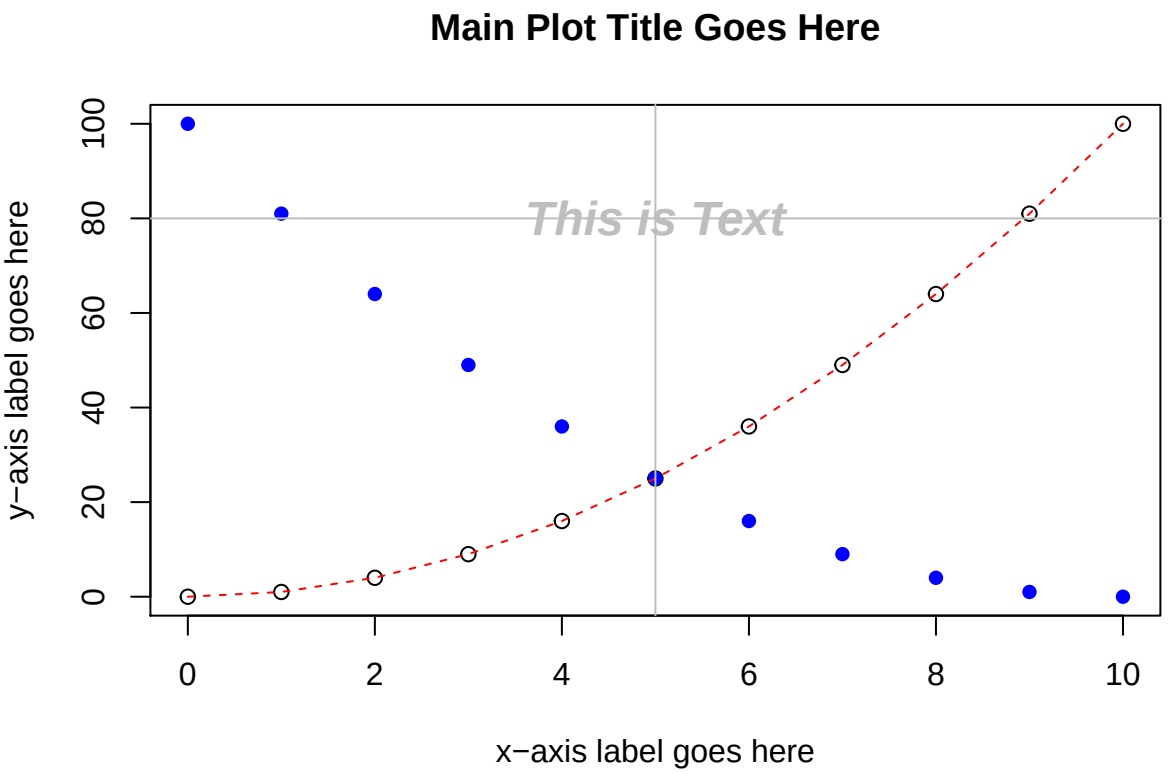


The text is centered on the coordinates you provide. You can also provide vectors of coordinates and text to write different things at once.

The easiest way to add a straight line to a plot is with `abline()`. By default it takes two arguments: `a` and `b` which are the intercept and slope, respectively, e.g., `abline(c(0,1))` will draw a 1:1 line. You can also do `abline(h = 5)` to draw a horizontal line at 5 or `abline(v = 5)` to draw a vertical line at `5`.

You can see that the text is centered on the coordinates `x = 5` and `y = 80` using `abline()`:

```
abline(h = 80, col = "grey")
abline(v = 5, col = "grey")
```



If you accidentally add a plot element that you don't want using a low-level plotting function, the only way to remove it is by re-running the high-level plotting function to start a new plot and adding only the objects you want. Try removing the "This is Text" text and the straight lines you drew with `abline()` from the plot displayed in your device.

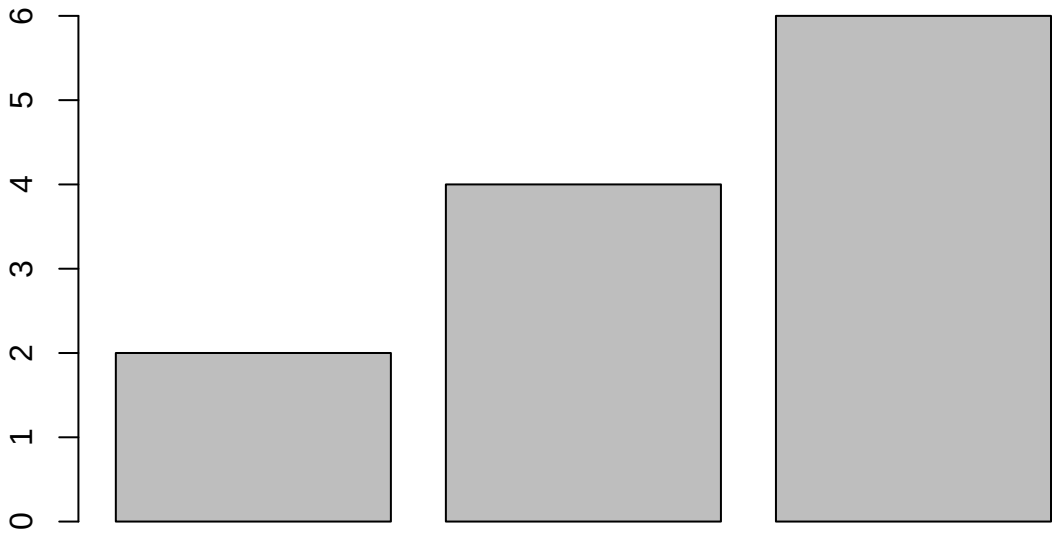
2.3 Other High-Level Plotting Functions

You have just seen the basics of making two dimensional scatter plots and line plots. You will now explore other types of graphs you can make.

2.3.1 The Bar Graph

Another very common graph is a bar graph. R has a `bargraph()` function, and again, it has lots of arguments. Here you will just make two common variations: single bars per group and multiple bars per group. Create a vector and plot it:

```
x1 = c(2,4,6)
barplot(x1)
```

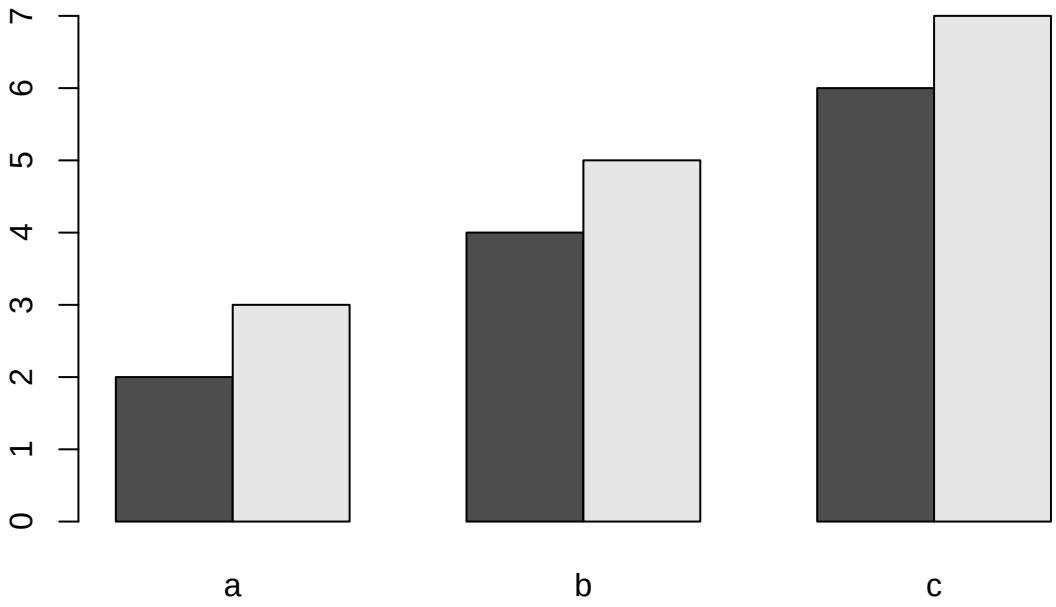


Notice that there are no group names on the bars (if `x1` had names, there would be). You can add names by using the argument `names.arg`:

```
barplot(x1, names.arg = c("a", "b", "c"))
```

Add some more information by including two bars per group. Create another vector and combine it with the old data:

```
x2 = c(3,5,7)
x3 = rbind(x1, x2)
barplot(x3, names.arg = c("a", "b", "c"), beside = T)
```



To add multiple bars per group, R needs a matrix like you just made. The **columns** store the heights of the bars that will be placed together in a group. Including the `beside = T` argument tells R to plot all groups as different bars as opposed to using a stacked bar graph.

Oftentimes, you will want to add error bars to a bar graph like this. To avoid digressing too much here, creating error bars is covered as a bonus topic (Section 2.11).

2.4 Box-and-Whisker Plots

Box-and-whisker plots are a great way to visualize the spread of your data. All you need to make a box-and-whisker plot is a grouping variable (a factor, revisit Section 1.5 if you don't remember what these are) and some continuous (i.e., numeric) data for each level of the factor. You will be using the `creel.csv` data set (see the instructions on acquiring and placing the data files in the appropriate location).

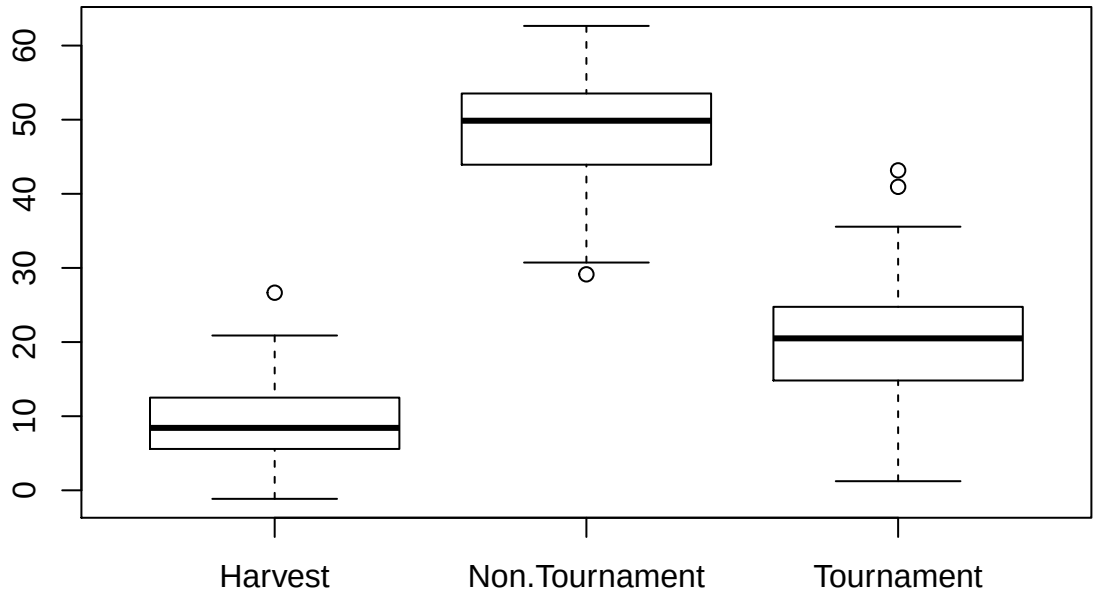
Read the data in and print a summary:

```
dat = read.csv("../Data/creel.csv")
summary(dat)
```

```
##            fishery        hours
##  Harvest       :100   Min.   :-1.150
##  Non.Tournament:100   1st Qu.: 9.936
##  Tournament    :100   Median :20.758
##                       Mean   :26.050
##                       3rd Qu.:43.896
##                       Max.   :62.649
```

This data set contains some simulated (i.e., fake) continuous and categorical data that represent 300 anglers who were creel surveyed[2]. In the data set, there are three categories (levels to the factor `fishery`) and the continuous variable is how many hours each angler fished this year. If you supply the generic `plot()` function with a continuous response (`y`) variable and a categorical predictor (`x`) variable, it will automatically assume you want to make a box-and-whisker plot:

```
plot(x = dat$fishery, y = dat$hours)
```



In the box-and-whisker plot above, the heavy line is the median, the ends of the boxes are the 25^{th} and 75^{th} percentiles and the "whiskers" are the 2.5^{th} and 97.5^{th} percentiles. Any points that are outliers (i.e., fall outside of the whiskers) will be shown as points[3].

It is worth introducing a shorthand syntax of typing the same command:

```
plot(hours ~ fishery, data = dat)
```

Instead of saying `plot(x = x.var, y = y.var)`, this expression says `plot(y.var ~ x.var)`. The `~` reads "as a function of". By specifying the `data` argument, you no longer need to indicate where the variables `hours` and `fishery` are found. Many R functions have a `data` argument that works this same way. It is sometimes preferable to plot variables with this syntax because it is often less code and is also the format of R's statistical equations[4].

[2] A creel survey is a sampling program where fishers are asked questions about their fishing behavior in order to estimate effort and harvest.

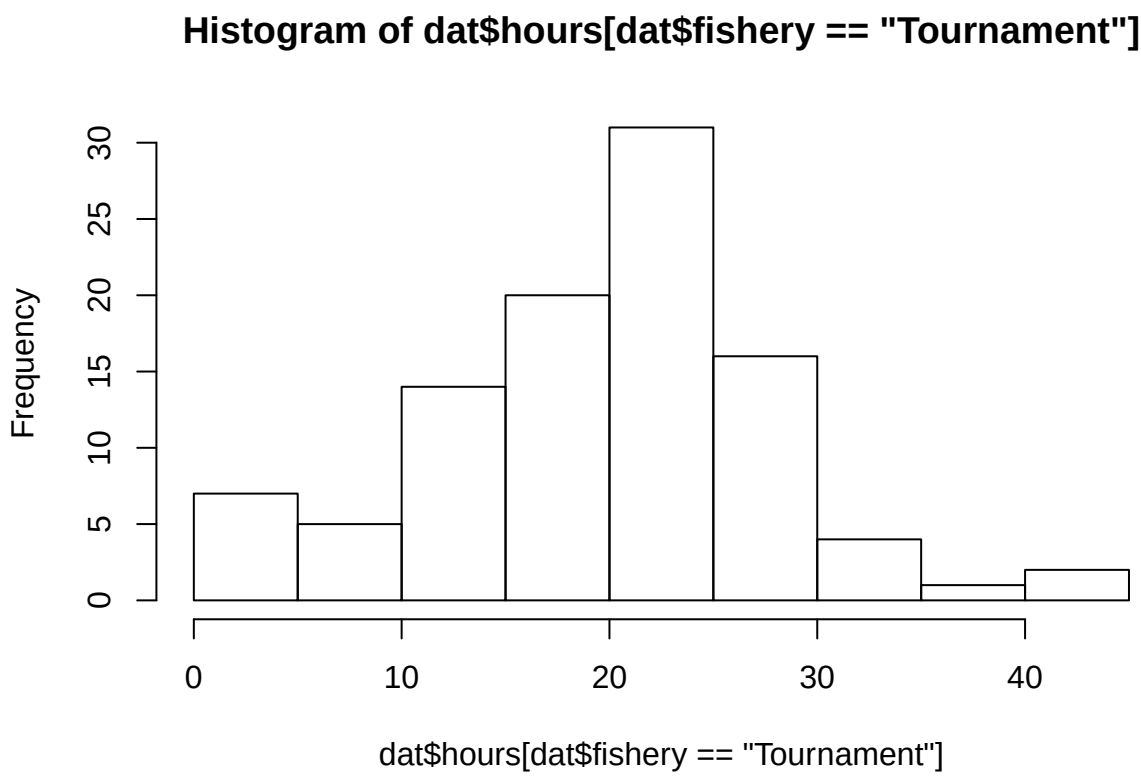
[3] Outliers can be turned off using the `outline = F` argument to the `plot()` function

[4] Which allows you to easily copy and paste the code between the model and plot functions, see Chapter 3

2.5 Histograms

Another way to show the distribution of a variable is with histograms. These figures show the relative frequencies of observations in different discrete bins. Make a histogram for the hours the surveyed tournament anglers fished this year:

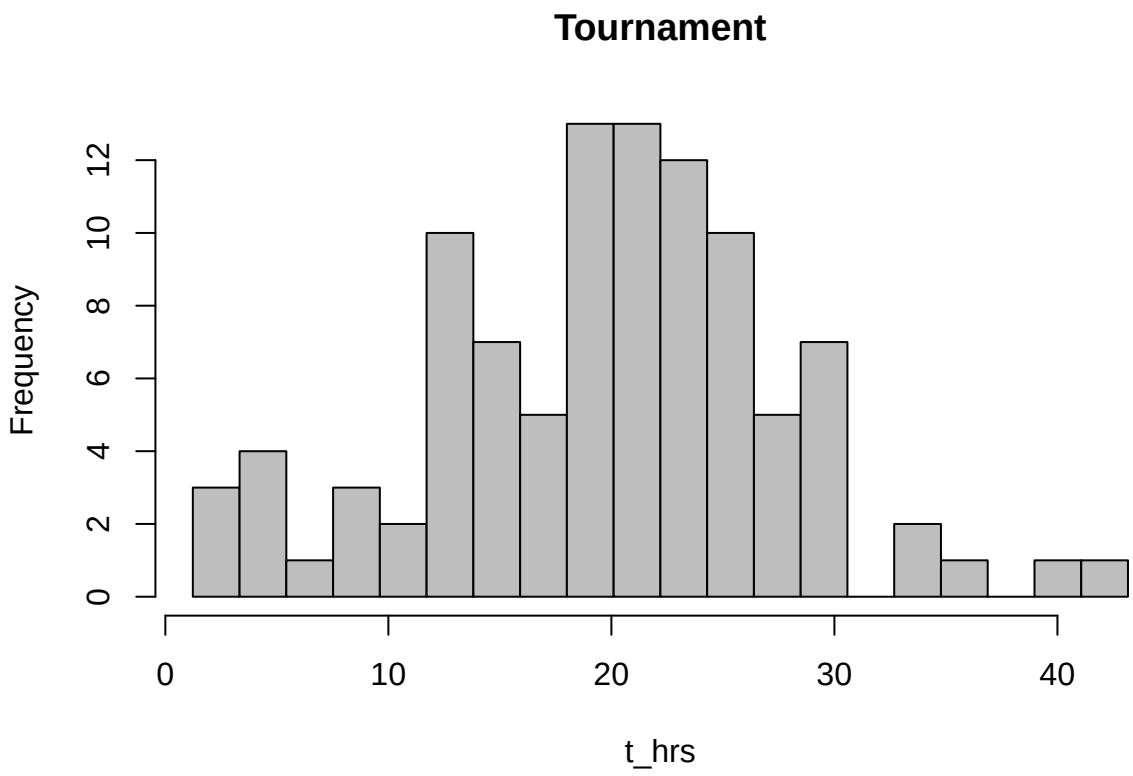
```
hist(dat$hours[dat$fishery == "Tournament"])
```



Notice the subset that extracts hours fished for tournament anglers only before plotting.

`hist()` automatically selects the number of bins based on the range and resolution of the data. You can specify how many evenly-sized bins you want to plot:

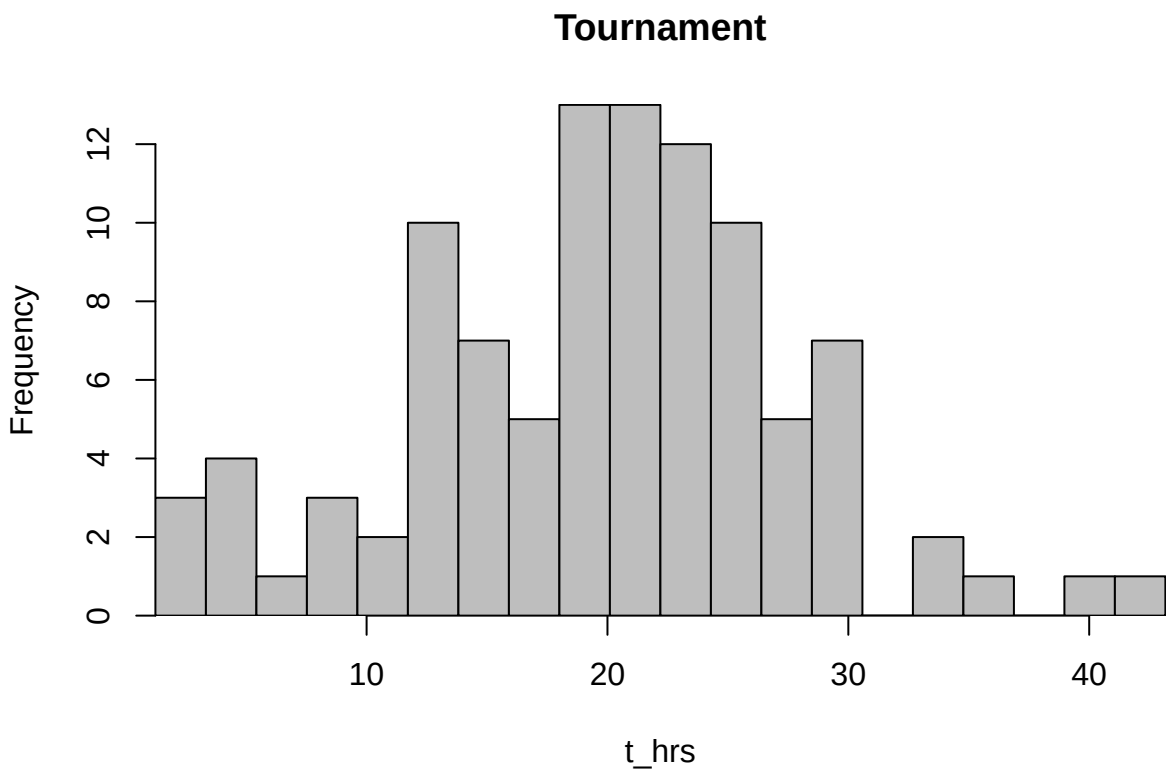
```
# extract the hours for tournament anglers
t_hrs = dat$hours[dat$fishery == "Tournament"]

# create the bin endpoints
nbins = 20
breaks = seq(from = min(t_hrs), to = max(t_hrs), length = nbins + 1)
hist(t_hrs, breaks = breaks, main = "Tournament", col = "grey")
```



2.6 The par Function

If it bothers you that the axes are "floating", you can fix this using this command:

```
par(xaxs = "i", yaxs = "i")
hist(t_hrs, breaks = breaks, main = "Tournament", col = "grey")
```



Here, you changed the graphical parameters of the graphics device by using the `par()` function. Once you change the settings in `par()`, they will remain that way until you start a new device.

The `par()` function is central to fine-tuning your graphics. Here, the `xaxs = "i"` and `yaxs = "i"` arguments essentially removed the buffer between the data and the axes. `par()` has options to change the size of the margins, add outer margins, change colors, etc. Some of the graphical parameters that can be passed to high- and low-level plotting functions (like those in Table 2.1) can also be passed `par()`. Check out the help file (`?par`) to see everything it can do. If you want to start over with fresh `par()` settings, start a new device.

2.7 New Temporary Devices

If you are using RStudio, then likely all of the plots you have made thus far have shown up in the lower right-hand corner or your RStudio window. You have been using RStudio's built-in plotting device. If you wish to open a new plotting device (maybe to put it on a separate monitor), you can use the following commands, depending on your operating system:

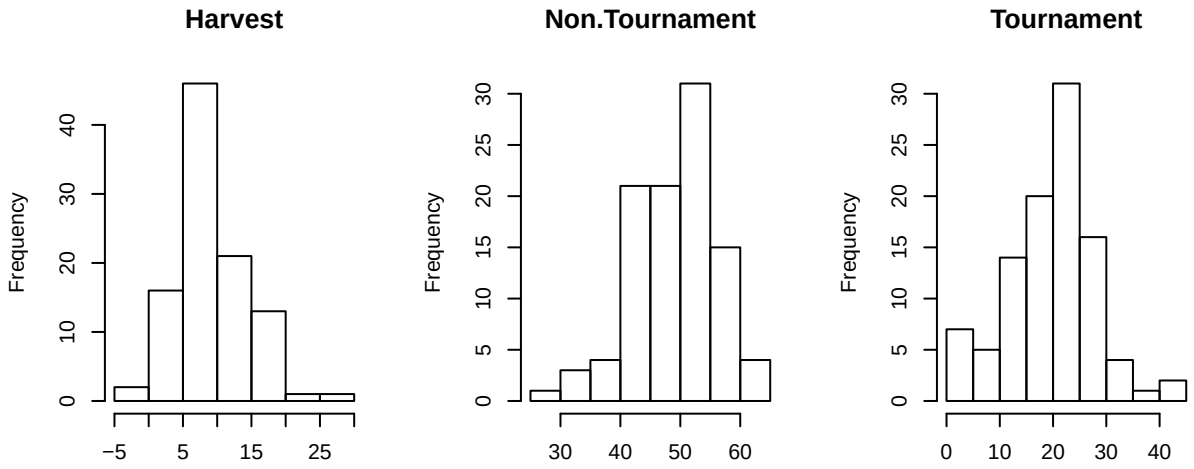
- **Windows Users** – just run `windows()` to open up a new plotting device. It will become the active device.
- **Mac Users** – similarly, you can run `quartz()` to open a new device.
- **Linux Users** – similarly, just run `x11()`.

2.8 Multi-panel Plots

Sometimes you want to display more than one plot at a time. You can make a multi-panel plot which allows for multiple plotting regions to show up simultaneously within the same plotting device. First, you need to change the layout of the plotting region. The easiest way to set up the device for multi-panel plotting is by using the `mfrow` argument in the `par()` function.

Below, the code says "set up the graphical parameters so that there is 1 row and 3 columns of plotting regions within the device". Every time you make a new plot, it will go in the next available plotting region. Make 3 histograms, each that represents a different sector of the fishery:

```
par(mfrow = c(1,3))
sapply(levels(dat$fishery), function(f) {
  hist(dat$hours[dat$fishery == f], main = f, xlab = "")
})
```



Here, `sapply()` applied a user-defined function that plots a histogram for a given fishery type to each of the fishery types separately[5], which allowed you to only need to type the `hist()` code once. There are other ways to make multi-panel plots, however, they are beyond the scope of this introductory material. See `?layout` for details. With this function you can change the size of certain plots and make them have different shapes (i.e., some squares, some rectangles, etc.), but it takes some pretty involved (though not impossible, by any means) specification of how you want the device to be split up into regions.

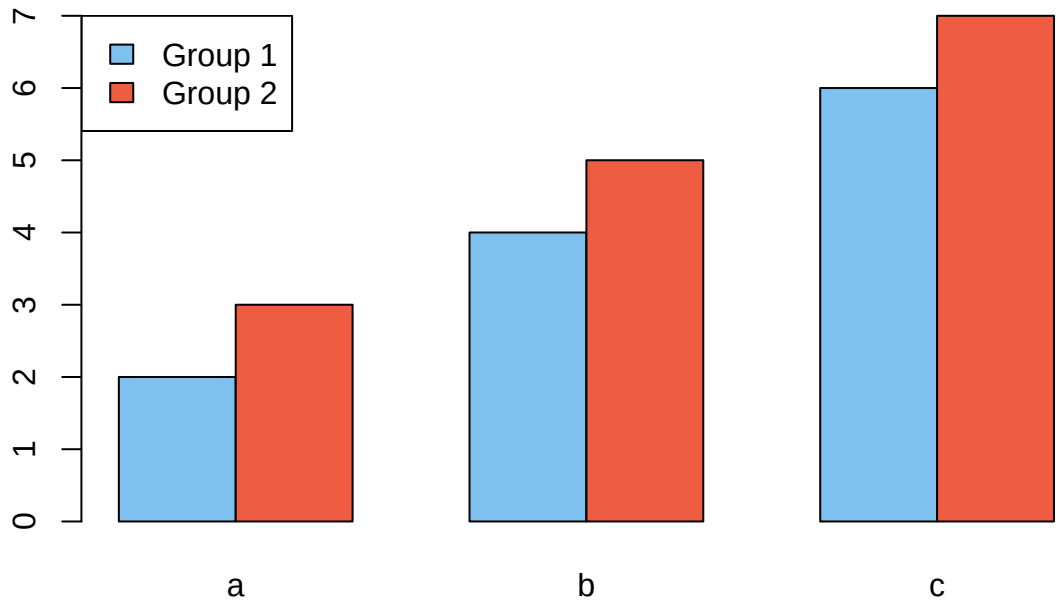
2.9 Legends

Oftentimes you will want to add a legend to plots to help people interpret what it is showing. You can add legends to R plots using the low-level plotting function `legend()`. Add a legend to the bar plot you made earlier with two groups. First, re-make the plot by running the high-level `barplot()` function, but change the colors of the bars to be shades of blue and red. Once you have the plot made, add the legend:

```
barplot(x3, beside = T,
        names.arg = c("a", "b", "c"),
        col = c("skyblue2", "tomato2"))
```

[5] `sapply()` works like `apply()`, except on vectors only so you don't need to supply it with a `1` or `2` for rows or columns

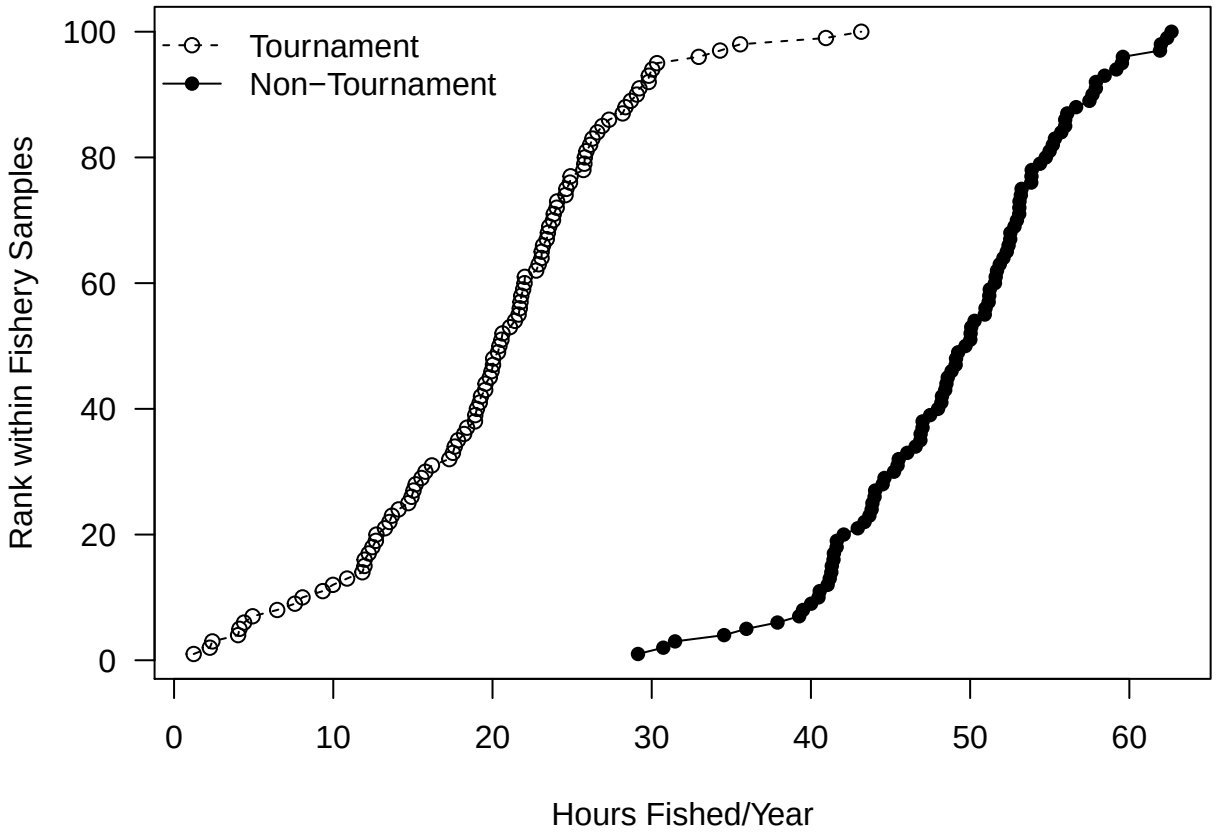
```
legend("topleft", legend = c("Group 1", "Group 2"),
       fill = c("skyblue2", "tomato2"))
```



The box can be removed using the `bty = "n"` argument and the size can be changed using `cex`. The position can be specified either with words (like above) or by using x-y coordinates.

Here is a more complex example:

```
# 1) extract and sort the hours for two fisheries from fewest hours
t_hrs = sort(dat$hours[dat$fishery == "Tournament"])
n_hrs = sort(dat$hours[dat$fishery == "Non.Tournament"])

# 2) make the plot: plot for t_hrs only, but ensure xlim covers both groups
# set the margins:
  # 4 lines of margin space on bottom and left,
  # 1 on top and right
par(mar = c(4,4,1,1))
plot(x = t_hrs, y = 1:length(t_hrs),
     type = "o", lty = 2, xlim = range(c(t_hrs, n_hrs)),
     xlab = "Hours Fished/Year", ylab = "Rank within Fishery Samples",
     las = 1)  # las = 1 says "turn the y-axis tick labels to be horizontal"
# 3) add info for the other fishery
points(x = n_hrs, y = 1:length(n_hrs), type = "o", lty = 1, pch = 16)
# 4) add the legend
legend("topleft", legend = c("Tournament", "Non-Tournament"),
       lty = c(2,1), pch = c(1, 16), bty = "n")
```



Notice that you need to be careful about the order of how you specify which `lty` and `pch` settings match up with the elements of the `legend` argument. In the `plot()` code, you specified that the `lty = 2` but didn't specify what `pch` should be (it defaults to `1`). So when you put the "Tournament" group first the the `legend` argument vector, you must be sure to use the corresponding plotting codes. The first element of the `lty` argument matches up with the first element of `legend`, and so on. Note the other plotting tricks used in the code above: changing the margins using `par(mar)` and the rotation of y-axis tick mark labels using `las = 1`.

2.10 Exporting Plots

There are two main ways to save plots. First is a quick-and-dirty method that saves the plots, but they are not high-resolution and you can't automate this process. The second method produces cleaner-looking high-resolution plots with code that can be embedded in your script, ensuring the same exact plot will be created each time the code is executed

2.10.1 Click Save

- If your plot is in the RStudio built-in graphics device: Right above the plot, click *Export > Save as Image*. Change the name, dimensions and file type.
- If your plot is in a plotting device window (opened with `windows()` or `quartz()`: Simply go to *File > Save*.

All plots will be saved in the working directory by default. You can also just copy the plot to your clipboard (*File > Copy to the clipboard > bitmap*) and paste it where you want. You should saving one of the plots

you made in this Chapter using this approach.

2.10.2 Use a function to place plot in a new file

If you are producing plots for a final report or publication, you want the output to be as clean-looking as possible and you want them to be fully reproducible so when something changes with your data or analysis in review, you can reproduce the same figure with the new results. You can save high-resolution plots through R by using the following steps:

```
# step 1: Make a pixels per inch object
ppi = 600

# step 2: Call the figure file creation function
png("TestFigure.png", h = 8 * ppi, w = 8 * ppi, res = ppi)

# step 3: Run the plot
# put all of your plotting code here (without windows())

# step 4: Close the device
dev.off()
```

A plot will be saved in your working directory containing the plot made by the code in step 3 above. The `ppi` object is pixels-per-inch. When you specify `h = 8 * ppi`, you are saying "make a plot with height equal to 8 inches". There are similar functions to make PDFs, tiff files, jpegs, etc. You should try saving one of the plots you made in this Chapter using this approach.

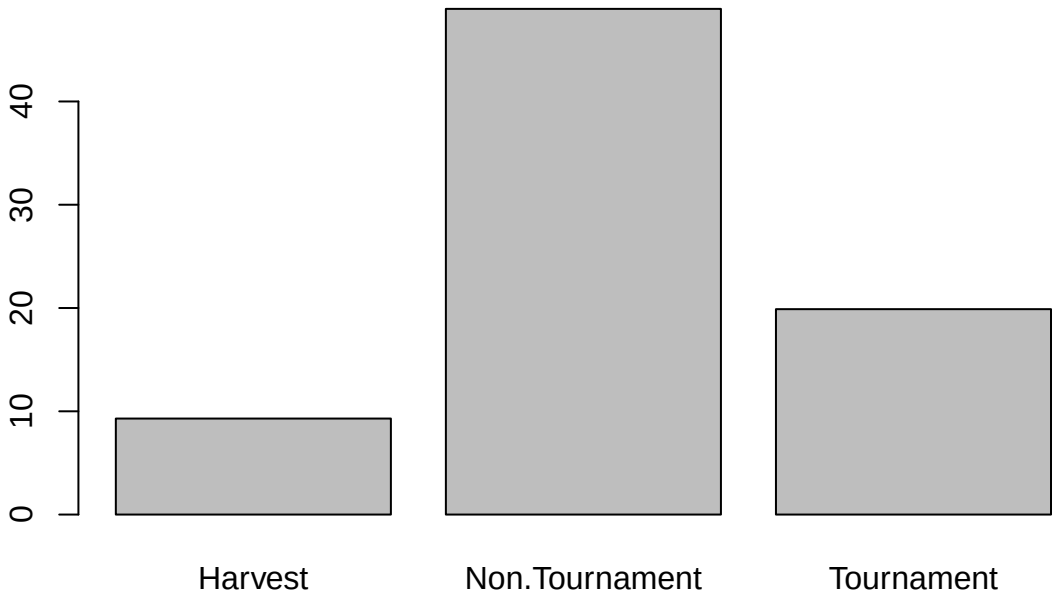
2.11 Bonus Topic: Error Bars

Rarely should you ever present estimates without some measure of uncertainty. The most common way for visualizing the uncertainty in an estimate is by using error bars, which can be added to an R plot using the lower-level function `arrows()`. To use `arrows()`, you need:

- Vectors of the `x` and `y` coordinates of the lower bound of the error bars
- Vectors of the `x` and `y` coordinates of the upper bound of the error bars

The syntax for `arrows()` is as follows: `arrows(x0, y0, x1, y1, ...)`, where `x0` and `y0` are the coordinates you are drawing "from" (e.g., lower limits) and the `x1` and `y1` are the coordinates you are drawing "to" (e.g., upper limits). The `...` represents other arguments to change how the error bars look. Calculate the mean of the different fishery sectors (if you don't remember how `tapply()` works, revisit Section 1.9) and plot them:

```
x_bar = tapply(dat$hours, dat$fishery, mean)
barplot(x_bar)
```



You wish to add error bars that represent 95% confidence intervals on the mean. You can create a 95% confidence interval using this basic formula:

$$\bar{x} \pm 1.96 * SE(\bar{x}), \tag{2.1}$$

where

$$\bar{x} = \frac{1}{n}\sum_{i}^{n} x_i, \tag{2.2}$$

and

$$SE(\bar{x}) = \frac{\sqrt{\frac{\sum_i^n (x_i - \bar{x})^2}{n-1}}}{\sqrt{n}} \tag{2.3}$$

Begin by creating a function to calculate the standard error ($SE(\bar{x})$):

```
calc_se = function(x) {
  sqrt(sum((x - mean(x))^2)/(length(x)-1))/sqrt(length(x))
}
```

Then calculate the standard errors for each fishery type:

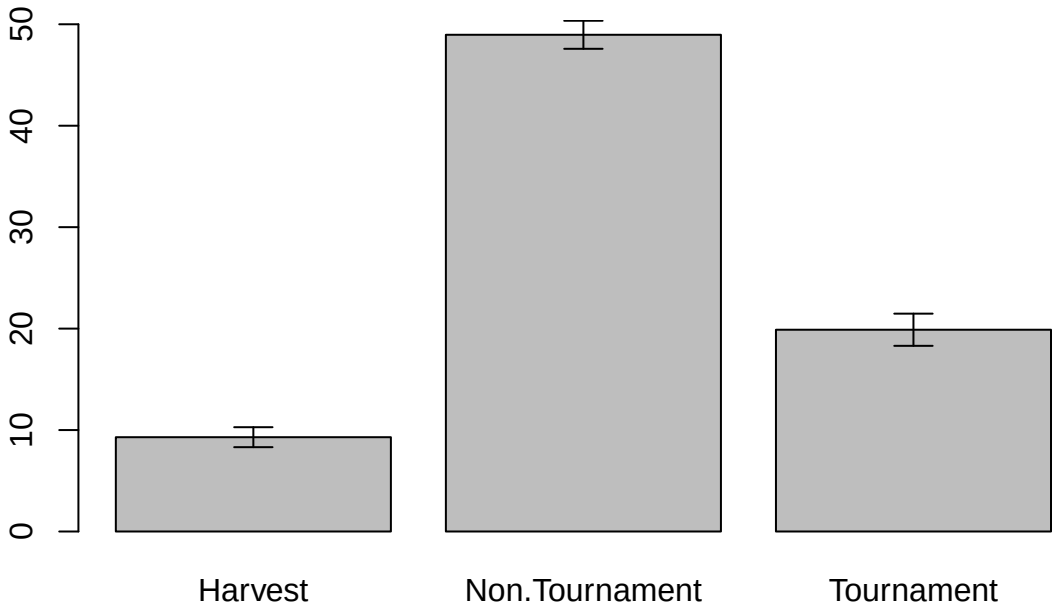
```
se = tapply(dat$hours, dat$fishery, calc_se)
```

Then calculate the lower and upper limits of your bars:

```
lwr = x_bar - 1.96 * se
upr = x_bar + 1.96 * se
```

Then draw them on using the `arrows()` function:

```
mp = barplot(x_bar, ylim = range(c(0, upr)))
arrows(x0 = mp, y0 = lwr, x1 = mp, y1 = upr, length = 0.1, angle = 90, code = 3)
```



Notice four things:

1. The use of `mp` to specify the `x` coordinates. If you do `mp = barplot(...)`, `mp` will contain the `x` coordinates of the midpoint of each bar.
2. `x0` and `x1` are the same: you wish to have vertical bars, so these must be the same while `y1` and `y2` differ.
3. The use of `ylim = range(c(0, upr))`: you want the y-axis to show the full range of all the error bars.
4. The three arguments at the end of `arrows`:
   - `length = 0.1`: the length of the arrow heads, fiddle with this until you like it.
   - `angle = 90`: the angle of the arrow heads, you want 90 here for the error bars.
   - `code = 3`: indicates that arrow heads should be drawn on both ends of the arrow.

Exercise 2

For this exercise, you will be making a few of plots and changing how they look to suit your taste. You will use a real dataset (`sockeye.csv`, see the instructions regarding acquiring the data files) this time from a sockeye salmon (*Oncorhynchus nerka*) population from the Columbia/Snake River system, and the data set was obtained from Kline and Flagg (2014). This population spawns in Redfish Lake in Idaho, which feeds into the Salmon River which is a tributary of the Snake River. In order to reach the lake, the sockeye salmon must successfully pass through a total eight dams that have fish passage mechanisms in place. The Redfish Lake population is one of the most endangered sockeye populations in the U.S. and travels farther (1,448 km), higher (1,996 m), and is the southernmost population of all sockeye populations in the world (Kline and Flagg, 2014). Given this uniqueness, a captive breeding program was initiated in 1991 to conserve the genes from this population. These data came from both hatchery-raised and wild fish and include average female spawner weight (g), fecundity (number of eggs), egg size (eggs/g), and % survival to the eyed-egg stage.

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Create a new R script called `Ex2.R` and save it in the `Chapter2` directory. Read in the data set `sockeye.csv`. Produce a basic summary of the data and take note of the data classes, missing values (`NA`), and the relative ranges for each variable.
2. Make a histogram of fish weights for only hatchery-origin fish. Set `breaks = 10` so you can see the distribution more clearly.
3. Make a scatter plot of the fecundity of females as a function of their body weight for wild fish only. Use whichever plotting character (`pch`) and color (`col`) you wish. Change the main title and axes labels to reflect what they mean. Change the x-axis limits to be 600 to 3000 and the y-axis limits to be 0 to 3500. (*Hint: The `NA`s will not cause a problem. R will only use points where there are paired records for both `x` and `y` and ignore otherwise*).
4. Add points that do the same thing but for hatchery fish. Use a different plotting character and a different color.
5. Add a legend to the plot to differentiate between the two types of fish.

6. Make a multi-panel plot in a new window with box-and-whisker plots that compare (1) spawner weight, (2) fecundity, and (3) egg size between hatchery and wild fish. (*Hint: each comparison will be on its own panel*). Change the titles of each plot to reflect what you are comparing.
7. Save the plot as a .png file in your working directory with a file name of your choosing.

EXERCISE 2 BONUS

1. Make a bar plot comparing the mean survival to eyed-egg stage for each type of fish (hatchery and wild). Add error bars that represent 95% confidence intervals.
2. Change the names of each bar, the main plot title, and the y-axis title.

3. Adjust the margins so there are 2 lines on the bottom, 5 on the left, 2 on the top, and 1 on the right.

Chapter 3

Basic Statistics

Chapter Overview

In this chapter, you will get familiar with the basics of using R for the purpose it was designed: statisitical analysis. You will learn how to:

- fit and interpret the output from various general linear models:
  - simple linear regression models
  - T-tests (also ANOVA)
  - ANCOVA models
  - Interactions
- conduct basic model selection
- fit basic GLMs: the logistic regression model
- Bonus topic: fitting non-linear regression models using `nls()`

R's built-in statistical modeling framework is pretty intuitive and comprehensive. R has gained popularity as a statistics software and is commonly used both in academia and governmental resource agencies. This popularity is likely a result of its power, flexibility, intuitive nature, and price (free!). For many students, this chapter may be the one that is most immediately useful.

IMPORTANT NOTE: If you did not attend the sessions corresponding to Chapters 1 or 2, you are recommended to walk through the material found in those chapters before proceeding to this material. Also note that if you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this book.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch3.R` and save it in the directory `C:/Users/YOU/Documents/R-Book/Chapter3`. Set your working directory to that location. Revisit the material in Sections 1.2 and 1.3 for more details on these steps.

3.1 The General Linear Model

Much of this chapter will focus on the general linear model, so it is important to become familiar with it. The general linear model is a family of models that allows you to determine the relationship (if any) between some continuous response variable (y) and some predictor variable(s) (x_n) and is often written as:

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_j x_{ij} + ... + \beta_n x_{in} + \varepsilon_i, \varepsilon_i \sim N(0, \sigma), \tag{3.1}$$

where the subscript i represents an individual observation. The predictor variable(s) can be either categorical (i.e., grouping variables used in ANOVA, t-test, etc.), continuous (regression), or a combination of categorical and continuous (ANCOVA). The main focus is to estimate the coefficients (β), and in some cases it is to determine if their values are "significantly" different from the value assumed by some null hypothesis.

The model makes several assumptions about the residuals[1] to obtain estimates of the coefficients. For reliable inference, the residuals must:

- be independent
- be normally-distributed
- have constant variance across range of the x-axis

In R, the general linear model is fitted using the `lm()` function. Here's the basic syntax is `lm(y ~ x, data = dat)`[2]; it says: "fit a model with `y` as the response variable and `x` as the sole predictor variable, look for the variables `x` and `y` in a data frame called `dat`".

3.1.1 Simple Linear Regression

Read the data found in `sockeye.csv` (see in the instructions for help with acquiring the data) into R. This is the same data set you used in Exercise 2 - revisit this section for more details on the meaning of the variables. These are real data and are presented in Kline and Flagg (2014).

```
dat = read.csv("../Data/sockeye.csv")
head(dat)
```

```
##   year  type weight fecund egg_size survival
## 1 1991 hatch     NA     NA       NA       NA
## 2 1992 hatch     NA     NA       NA       NA
## 3 1993 hatch   1801   2182    12.25    46.58
## 4 1994 hatch   1681   2134     7.92    50.98
## 5 1995 hatch   2630   1576    21.61    68.06
## 6 1996 hatch   2165   2171     8.74    63.43
```

To fit a regression model using `lm()`, both `x` and `y` must be continous (numeric) variables. In the data set `dat`, two such variables are the `weight` and `fecund`. Fit a regression model where you link the average fecundity (number of eggs) of an individual year to the average weight (in grams) in that year. Ignore for now that the fish come from two sources: hatchery and wild origin.

```
fit1 = lm(fecund ~ weight, data = dat)
```

If you run just the `fit1` object, you will see the model you ran along with the coefficient estimates of the intercept (β_0) and the slope (β_1):

```
fit1
```

```
##
## Call:
## lm(formula = fecund ~ weight, data = dat)
##
## Coefficients:
## (Intercept)       weight
##   1874.6496       0.2104
```

[1] The residuals (ε_i) are the difference between the data point y_i and the model prediction $\hat{y}_i$: $\varepsilon_i = y_i - \hat{y}_i$

[2] This should look familar from Section 2.4

This model looks like this:

$$y_i = \beta_0 + \beta_1 x_{i1} + \varepsilon_i, \varepsilon_i \sim N(0, \sigma), \tag{3.2}$$

where x_{i1} is `weight`. The coefficients are interpretted as:

- β_0: the y-intercept (mean `fecund` at zero `weight`)
- β_1: the slope (change in `fecund` for one unit increase in `weight`)

For more information about the model fit, you can use the `summary()` function:

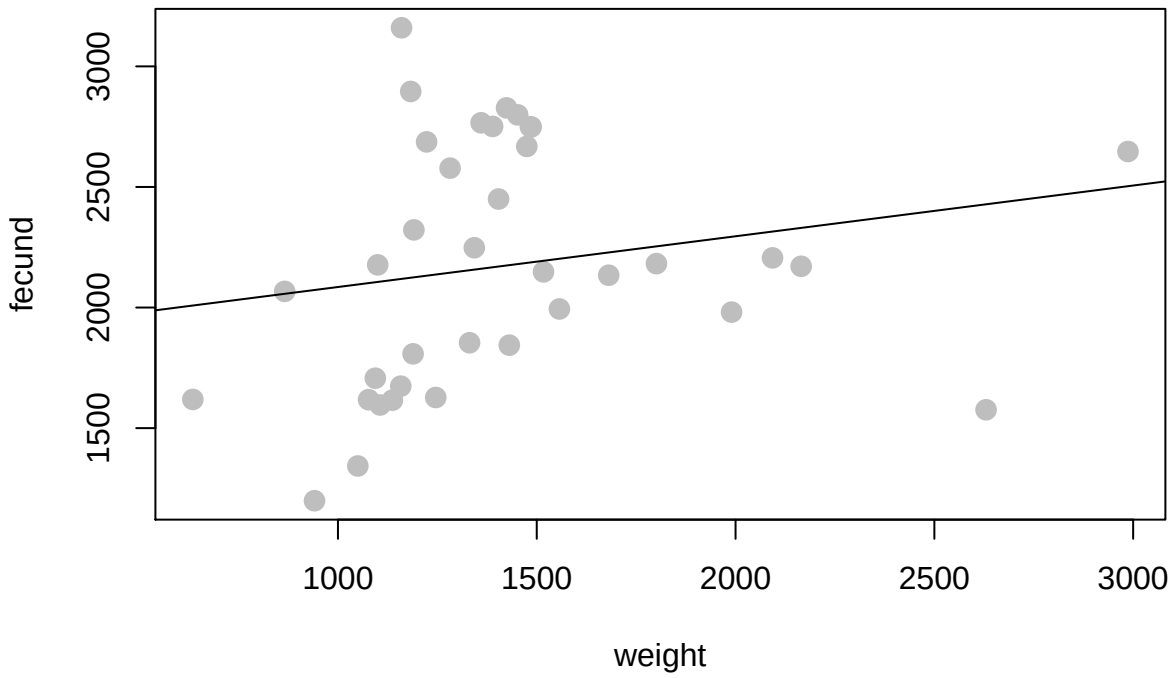
```
summary(fit1)
```

```
##
## Call:
## lm(formula = fecund ~ weight, data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -873.67 -389.28  -71.65  482.96 1041.24
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 1874.6496   269.4369   6.958 4.33e-08 ***
## weight         0.2104     0.1803   1.167    0.251
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 500.6 on 35 degrees of freedom
##   (7 observations deleted due to missingness)
## Multiple R-squared:  0.03745,    Adjusted R-squared:  0.009945
## F-statistic: 1.362 on 1 and 35 DF,  p-value: 0.2511
```

Again the coefficient estimates are shown, but now you see the uncertainty on the parameter estimates (standard errors), the test statistic, and the p-value testing the null hypothesis that each coefficient has a zero value. Here you can see that the p-value does not support rejection of the null hypothesis that the slope is zero. You can see the residual standard error (variability of data around the fitted line and the estimate of σ), the R^2 value (the proportion of variation in `fecund` explained by variation in `weight`), and the p-value of the overall model.

You can easily see the model fit by using the `abline()` function. Make a new plot and add the fitted regression line:

```
plot(fecund ~ weight, data = dat, col = "grey", pch = 16, cex = 1.5)
abline(fit1)
```



It fits, but not very well. It seems there are two groups: one with data points mostly above the line and one with data points mostly below the line. You'll now run a new model to get at this.

3.1.2 ANOVA: Categorical predictors

ANOVA models attempt to determine if the means of different groups are different. You can fit them in the same basic `lm()` framework. But first, notice that:

```
class(dat$type); levels(dat$type)
```

```
## [1] "factor"
```

```
## [1] "hatch" "wild"
```

tells you the `type` variable is a factor. It has levels of `"hatch"` and `"wild"` which indicate the origin of the adult spawning fish sampled each year. If you pass `lm()` a predictor variable with a factor class, R will automatically fit it as an ANOVA model. See Section 1.5 for more details on factors. Factors have an explicit ordering of the levels. By default, this ordering happens alphabetically: if your factor has levels `"a"`, `"b"`, and `"c"`, they will be assigned the order of `1`, `2` and `3`, respectively. You can always see how R is ordering your factor by doing something similar to this:

```
pairs = cbind(
  as.character(dat$type),
  as.numeric(dat$type)
)

head(pairs); tail(pairs)
```

```
##       [,1]    [,2]
## [1,] "hatch" "1"
## [2,] "hatch" "1"
## [3,] "hatch" "1"
## [4,] "hatch" "1"
## [5,] "hatch" "1"
## [6,] "hatch" "1"
```

```
##       [,1]   [,2]
## [39,] "wild" "2"
## [40,] "wild" "2"
## [41,] "wild" "2"
## [42,] "wild" "2"
## [43,] "wild" "2"
## [44,] "wild" "2"
```

The functions `as.character` and `as.numeric` are coersion functions: they attempt to change the way something is interpretted. Notice that the level `"hatch"` is assigned the order `1` because it comes before `"wild"` alphebetically. The first level is termed the **reference level** because it is the group that all other levels are compared to when fitting a model. You can change the reference level using `dat$type_rlvl = relevel(dat$type, ref = "wild")`.

You are now ready to fit the ANOVA model, which will measure the size of the difference in the mean `fecund` between different levels of the factor `type`:

```
fit2 = lm(fecund ~ type, data = dat)
```

Think of this model as being written as:

$$y_i = \beta_0 + \beta_1 x_{i1} + \varepsilon_i \tag{3.3}$$

and assume that $x_{i1} = 0$ if observation i is a fish from the `"hatch"` level and $x_{i1} = 1$ if observation i is a fish from the `"wild"` level. Note that Equations (3.2) and (3.3) are the same, the only thing that differs is the coding of the variable x_{i1}. In the ANOVA case:

- β_0 (the intercept) is the mean `fecund` for the `"hatch"` level and
- β_1 is the difference in mean `fecund`: `"wild"` - `"hatch"`.

So when you run `coef(fit2)` to extract the coefficient estimates and get:

```
## (Intercept)    typewild
##   1846.2500    713.3971
```

you see that the mean fecundity of hatchery fish is about 1846 eggs and that the average wild fish has about 713 more eggs than the average hatchery fish across all years. The fact that the p-value associated with the `typewild` coefficient when you run `summary(fit2)` is less than 0.05 indicates that there is statistical evidence that the difference in means is not zero.

Verify your interpretation of the coefficients:

```
m = tapply(dat$fecund, dat$type, mean, na.rm = T)

# b0:
m[1]
```

```
##   hatch
## 1846.25
```

```
# b1:
m[2] - m[1]
```

```
##     wild
## 713.3971
```

3.1.3 ANCOVA: Continuous and categorical predictors

Now that you have seen that hatchery and wild fish tend to separate along the fecundity axis (as evidenced by the ANOVA results above), you would like to include this in your original regression model. You will fit two regression lines within the same model: one for hatchery fish and one for wild fish. This model is called an ANCOVA model and looks like this:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \varepsilon_i \tag{3.4}$$

If x_{i1} is `type` coded with 0's and 1's as in Section 3.1.2 and x_{i2} is `weight`, then the coefficients are interpretted as:

- β_0: the y-intercept of the `"hatch"` level (the reference level)
- β_1: the difference in mean `fecund` at the same weight: `"wild" - "hatch"`
- β_2: the slope of the `fecund` *vs.* `weight` relationship (this model assumes the lines have common slopes, i.e., that the lines are parallel)

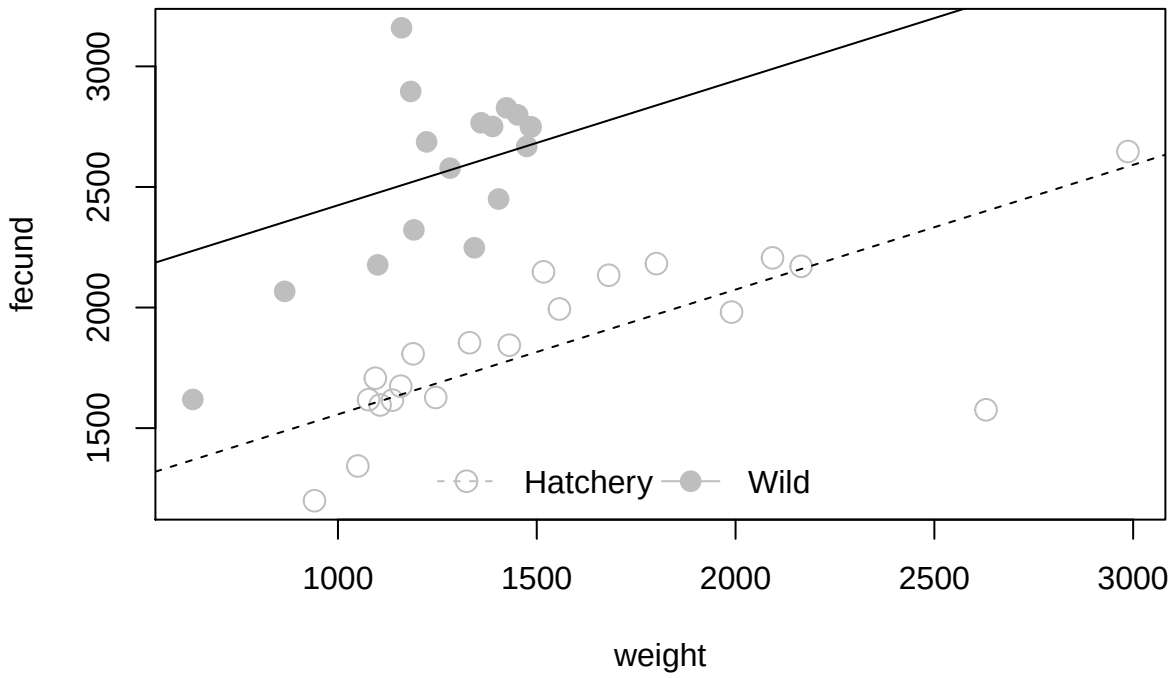
You can fit this model and extract the coefficents table from the summary:

```
fit3 = lm(fecund ~ type + weight, data = dat)
summary(fit3)$coef
```

```
##                 Estimate  Std. Error  t value     Pr(>|t|)
## (Intercept) 1039.6417645 175.0992184 5.937444 1.038268e-06
## typewild     866.9909998  96.2473387 9.007948 1.578239e-10
## weight         0.5173716   0.1051039 4.922477 2.164460e-05
```

And you can plot the fit. Study this code to make sure you know what each is doing. Use what you know about the meanings of the three coefficients to decipher the two `abline()` commands. Remember that `abline()` takes takes two arguments: `a` is the intercept and `b` is the slope.

```
plot(fecund ~ weight, data = dat, col = "grey",
     pch = ifelse(dat$type == "hatch", 1, 16), cex = 1.5)
abline(coef(fit3)[c(1,3)], lty = 2)
abline(sum(coef(fit3)[c(1,2)]), coef(fit3)[3])
legend("bottom", legend = c("Hatchery", "Wild"), pch = c(1,16), lty = c(2,1),
       col = "grey", pt.cex = 1.5, bty = "n", horiz = T)
```



3.1.4 Interactions

Above, you have included an additional predictor variable (and parameter) in your model to help explain variation in the `fecund` variable. However, you have assumed that the effect of weight on fecundity is common between hatchery and wild fish (note the parallel lines in the figure above). You may have reason to believe that the effect of weight depends on the origin of the fish, e.g., wild fish may tend to accumulate more eggs than hatchery fish for the same increase in weight. Cases where the magnitude of the effect depends on the value of another predictor variable are known as "interactions". You can write the interactive ANCOVA model like this:

$$y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \beta_3 x_{i1} x_{i2} + \varepsilon_i \tag{3.5}$$

If x_{i1} is `type` coded with 0's and 1's as in Section 3.1.2 and x_{i2} is `weight`, then the coefficients are interpretted as:

- β_0: the y-intercept of the `"hatch"` level (the reference level)
- β_1: the difference in y-intercept between the `"wild"` level and the `"hatch"` level.
- β_2: the slope of the `"hatch"` level
- β_3: the difference in slope between the `"wild"` level and the `"hatch"` level.

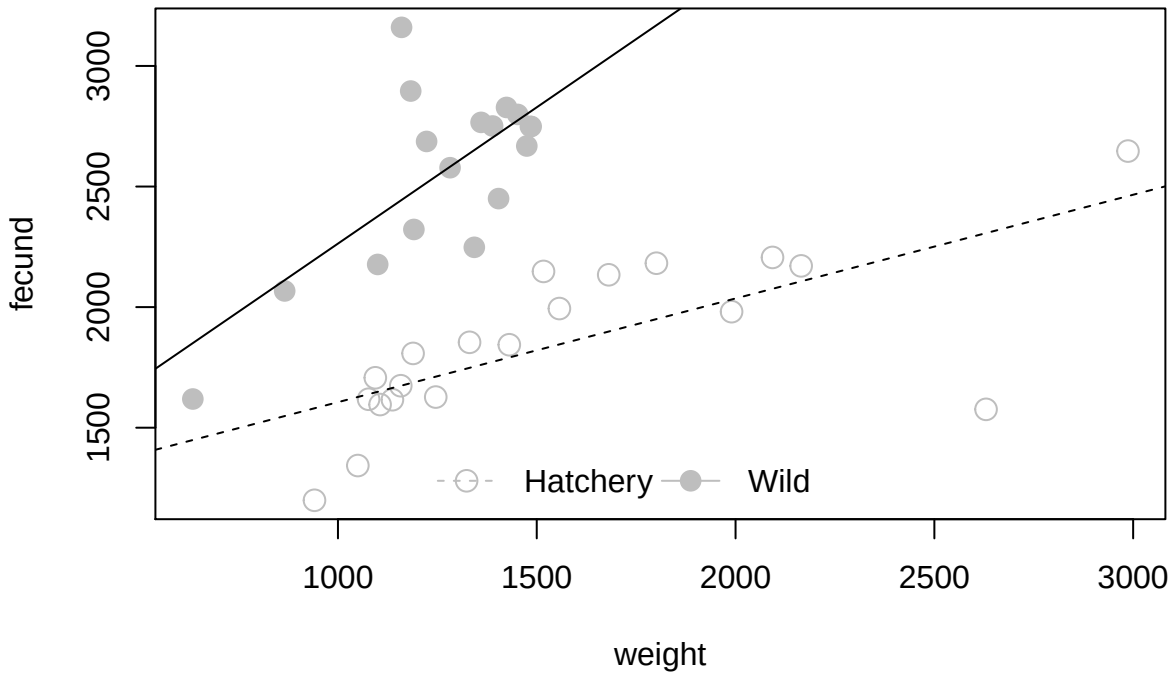
You can fit this model:

```
fit4 = lm(fecund ~ type + weight + type:weight, data = dat)

# or
# fit4 = lm(fecund ~ type * weight, data = dat)
```

The first option above is more clear in its statement, but both do the same thing.

Plot the fit. Study these lines to make sure you know what each is doing. Use what you know about the meanings of the four coefficients to decipher the two `abline()` commands.

```
plot(fecund ~ weight, data = dat, col = "grey",
     pch = ifelse(dat$type == "hatch", 1, 16), cex = 1.5)
abline(coef(fit4)[c(1,3)], lty = 2)
abline(sum(coef(fit4)[c(1,2)]), sum(coef(fit4)[c(3,4)]))
legend("bottom", legend = c("Hatchery", "Wild"), pch = c(1,16), lty = c(2,1),
       col = "grey", pt.cex = 1.5, bty = "n", horiz = T)
```



Based on the coefficients table:

```
summary(fit4)$coef
```

```
##                    Estimate Std. Error    t value     Pr(>|t|)
## (Intercept)     1175.7190847 174.545223  6.7358995 1.125255e-07
## typewild         -42.8721082 398.980751 -0.1074541 9.150794e-01
## weight             0.4300894   0.105592  4.0731253 2.732580e-04
## typewild:weight    0.7003389   0.299104  2.3414560 2.539681e-02
```

It seems that fish of the different origins have approximately the same intercept, but that their slopes are quite different.

3.1.5 AIC Model Selection

You have now fitted four different models, each that makes different claims about how you can predict the fecundity of a given sockeye salmon at Redfish Lake. If you are interested in determining *which* of these models you should use for prediction, you need to use **model selection**. Model selection attempts to find the model that is likely to have the smallest out-of-sample prediction error (i.e., future predictions will be close to what actually happens). One model selection metric is the AIC[3]. Lower AIC values mean the model should have better predictive performance. Obtain a simple AIC table from your fitted model objects and sort the table by increasing values of AIC:

```
tab = AIC(fit1, fit2, fit3, fit4)
tab[order(tab$AIC),]
```

```
##      df      AIC
## fit4  5 522.0968
## fit3  4 525.7834
## fit2  3 543.6914
## fit1  3 568.9166
```

In general, AIC values that are different by more than 2 units are interpretted as having importantly different predictive performance. Based on this very quick-and-dirty analysis, it seems that in predicting future fecundity, you would want to use the interactive ANCOVA model.

3.2 The Generalized Linear Model

The models you fitted above were called "general linear models". They all made the assumption that the residuals (ε_i) are normally-distributed and that the response variable and the predictor variables are linearly related. Oftentimes data and analyses do not follow this assumption. For these cases you should often move to the broader family of statistical models known as **generalized linear models**[4].

3.2.1 Logistic Regression

One example is in the case of **binary** data. Binary data have two outcomes, e.g., success/failure, lived/died, male/female, spawned/gravid, happy/sad, etc. If you wish to predict how the probability of one outcome over the other changes depending on some other variable, then you need to use the **logistic regression model**, which is written as:

$$logit(p_i) = \beta_0 + \beta_1 x_{i1} + ... + \beta_j x_{ij} + ... + \beta_n x_{in}, y_i \sim Bernoulli(p_i) \tag{3.6}$$

Where p_i is the probability of success for trial i ($y_i = 1$) at the values of the predictor variables x_{ij}. The $logit(p_i)$ is the **link function** that links the linear parameter scale to the data scale. It constrains the value of p_i to be between 0 and 1 regardless of the values of the β coefficients. The logit link function does this:

$$logit(p_i) = log\left(\frac{p_i}{1 - p_i}\right) \tag{3.7}$$

which is the natural logarithm of the **odds**, a measure of how likely the event is to happen relative to it not happening. Make an R function to calculate the logit transformation:

[3] Akaike's Information Criterion. An excellent overview of AIC with ecological applications is given in Anderson et al. (2000)
[4] General linear models are a member of this family

```
logit = function(p) {
  log(p/(1 - p))
}
```

If you have the result of `logit(p[i])` (which is given by the β coefficients and the x_{ij} data in Equation (3.6)) and need to get `p[i]`, you can apply the inverse logit function:

$$expit(lp_i) = \frac{e^{lp_i}}{1 + e^{lp_i}} \tag{3.8}$$

where $lp_i = logit(p_i)$. Make a function for the inverse logit transformation:

```
expit = function(lp) {  # lp stands for logit(p)
  exp(lp)/(1 + exp(lp))
}
```

Because of the logit link function, the coefficients have different interpretations than in the general linear models you fitted in the previous section: they are expressed in terms of **log odds**.

Fit a logistic regression model to the sockeye salmon data. None of the variables of interest are binary, but you can create one. Look at the variable `dat$survival`. This is the average % survival of all eggs laid that make it to the "eye-egg" stage. Create a new variable `binary` which takes on a 0 if `dat$survival` is less than 70% and a 1 otherwise:

```
dat$binary = ifelse(dat$survival < 70, 0, 1)
```

This will be your response variable and your model will estimate how the probability of `binary` being a `1` changes (or doesn't) depending on the value of other variables.

Analogous to the simple linear regression model (Section 3.1.1), estimate how p changes with `weight`:

```
fit1 = glm(binary ~ weight, data = dat, family = binomial)
summary(fit1)$coef
```

```
##                 Estimate Std. Error   z value   Pr(>|z|)
## (Intercept)  4.363441330 1.76943946  2.466002 0.01366306
## weight      -0.002819271 0.00125243 -2.251040 0.02438303
```

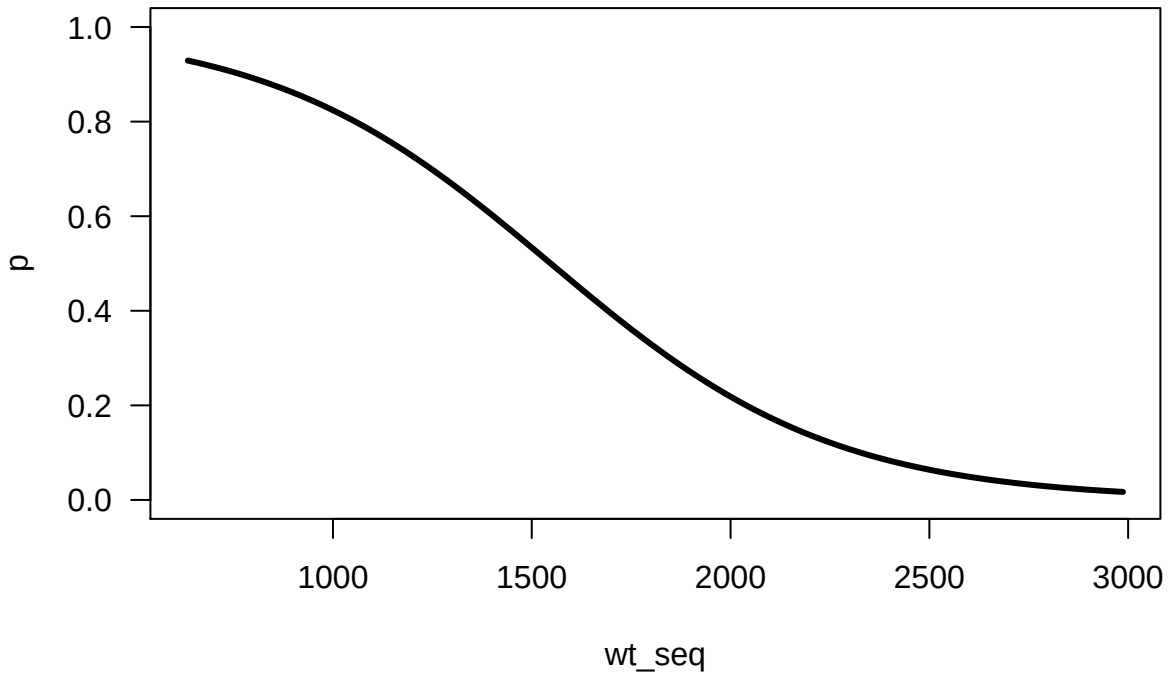
The coefficients are interpretted as (remember, "success" is defined as having at least 70% egg survival to the stage of interest):

- β_0: the log odds of success for a fish with zero weight (which is not all that important, let alone difficult to interpret). It can be transformed into more interpretable quantities:
  - e^{β_0} is the odds of success for fish with zero weight and
  - $expit(e^{\beta_0})$ is the probability of success for fish with zero weight.
- β_1: the additive effect of fish weight on the log odds of success. More interpretable expressions are:
  - e^{β_1} is the ratio of the odds of success at two consective weights (e.g., 1500 and 1501) and Claims about e^{β_1} are made as "for every one gram increase in weight, success became * e^{β_1} times as likely to happen".
- You can predict the probability of success at any weight using $expit(\beta_0 + \beta_1 weight)$

You can plot the fitted model:

```
# create a sequence of weights to predict at
wt_seq = seq(min(dat$weight, na.rm = T),
             max(dat$weight, na.rm = t),
             length = 100)

# extract the coefficients and get p
```

```
p = expit(coef(fit1)[1] + coef(fit1)[2] * wt_seq)

# plot the relationship
plot(p ~ wt_seq, type = "l", lwd = 3, ylim = c(0,1), las = 1)
```



Fit another model comparing the probability of success between hatchery and wild fish (analogous to the ANOVA model in Section 3.1.2):

```
fit2 = glm(binary ~ type, data = dat, family = binomial)
summary(fit2)$coef
```

```
##                Estimate Std. Error    z value  Pr(>|z|)
## (Intercept) -0.2006707  0.4494666 -0.4464641 0.6552620
## typewild     1.3793257  0.7272845  1.8965421 0.0578884
```

An easier way to obtain the predicted probability is by using the `predict` function:

```
predict(fit2,
        newdata = data.frame(type = c("hatch", "wild")),
        type = "response")
```

```
##         1         2
## 0.4500000 0.7647059
```

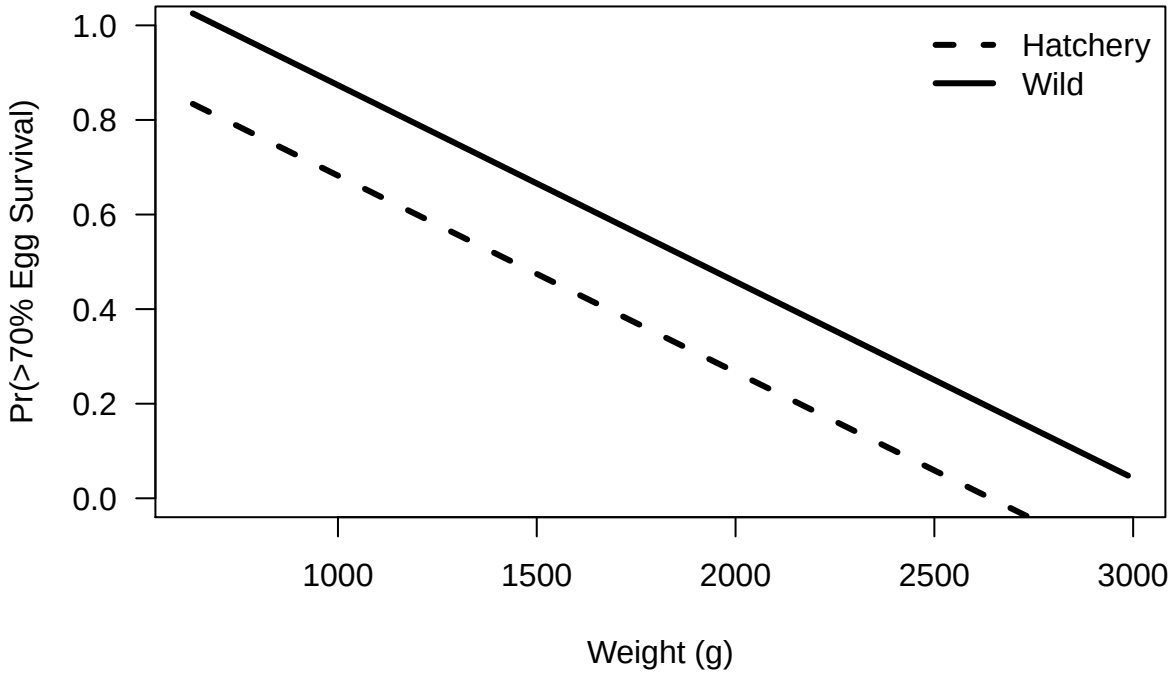
This plugs in the two possible values of the predictor variable and asks for the fitted probabilities.

Incorporate the origin type into your original model:

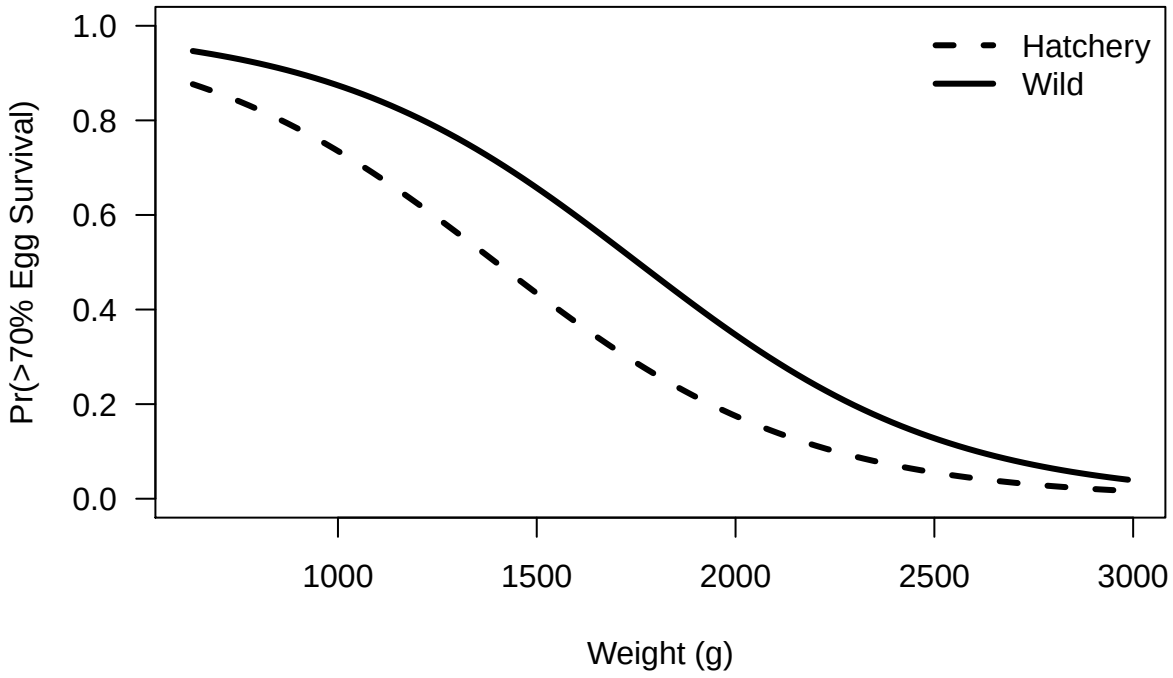
```
fit3 = glm(binary ~ type + weight, data = dat)
```

and obtain/plot the fitted probabilities for each group:

```
p_hatch = predict(
  fit3, newdata = data.frame(type = "hatch", weight = wt_seq),
  type = "response"
)
p_wild = predict(
  fit3, newdata = data.frame(type = "wild", weight = wt_seq),
  type = "response"
)

plot(p_wild ~ wt_seq, type = "l", lwd = 3, lty = 1,
     ylim = c(0,1), las =1,
     xlab = "Weight (g)", ylab = "Pr(>70% Egg Survival)"
     )
lines(p_hatch ~ wt_seq, lwd = 3, lty = 2)
legend("topright", legend = c("Hatchery", "Wild"),
       lty = c(2,1), lwd = 3, bty = "n")
```



Look for an interaction (all the code is the same except use `glm(binary ~ type * weight)` instead of `glm(binary ~ type + weight)` and change everything to `fit4` instead of `fit3`).



3.2.2 AIC Model Selection

You may have noticed that you just did the same analysis with `binary` as the response instead of `fecund`. Perform an AIC analysis to determine which model is likely to be best for prediction:

```
tab = AIC(fit1, fit2, fit3, fit4)
tab[order(tab$AIC),]
```

```
##      df      AIC
## fit1  2 45.40720
## fit4  3 46.00622
## fit2  2 50.07577
## fit3  4 50.42090
```

Oddly enough, the two best models are the simplest one and the most complex one, with `fit1` being the best, but not by a large margin.

3.3 Probability Distributions

A probability distribution is a way of representing the probability of an event or value of a parameter and they are central to statistical theory. Some of the most commonly used distributions are summarized in Table 3.1, along with the suffixes of the functions in R that correspond to each distribution[5].

[5]For an excellent and ecologically-focused description of probability distributions, checkout Ben Bolker's book, *Ecological Models and Data in R*. There is a free proof version online: https://ms.mcmaster.ca/~bolker/emdbook/book.pdf

Table 3.1: A brief description of probability distributions commonly used in ecological problems, including the function suffix in R.

Distribution	Description	R Suffix
Continuous		
Normal	Models the relative frequency of outcomes that are symmetric around a mean, can be negative	`-norm()`
Lognormal	Models the relative frequency of outcomes that are normally-distributed on the log-scale	`-lnorm()`
Uniform	Models values that are between two endpoints and that all occur with the same frequency	`-unif()`
Beta	Models values that are between 0 and 1	`-beta()`
Discrete		
Binomial	Models the number of successes from a given number of trials when there are only two possible outcomes and all trials have the same probability of success	`-binom()`
Multinomial	The same as the binomial distribution, but when there are more than two possible outcomes	`-multinom()`
Poisson	Used for count data in cases where the variance and mean are roughly equal	`-pois()`

In R, there are four different ways to use each of these distribution functions (each has a separate prefix):

- **The probability density (or mass) function** (`d-`): the height of the probability distribution function at some given value of the random variable.
- **The cumulative density function** (`p-`): what is the sum of the probability densities for all random variables below the input argument `q`.
- **The quantile function** (`-q`): what value of the random variable do `p`% fall below?
- **The random deviates function** (`-r`): generates random variables from the distribution in proportion to their probability density.

Suppose that x represents the length of individual age 6 largemouth bass in your private fishing pond. Assume that $x \sim N(\mu = 500, \sigma = 50)$[6]. Here is the usage of each of the distribution functions and a plot illustrating them:

```
# parameters
mu = 500; sig = 50

# a sequence of possible random variables (fish lengths)
lengths = seq(200, 700, length = 100)

# a sequence of possible cumulative probabilities
cprobs = seq(0, 1, length = 100)

densty = dnorm(x = lengths, mean = mu, sd = sig)  # takes specific lengths
cuprob = pnorm(q = lengths, mean = mu, sd = sig)  # takes specific lengths
quants = qnorm(p = cprobs, mean = mu, sd = sig)   # takes specific probabilities
random = rnorm(n = 1e4, mean = mu, sd = sig)      # takes a number of random deviates to make
```

[6] English: x is a normal random variable with mean equal to 500 and standard deviation equal to 50

```
# set up plotting region: see ?par for more details
# notice the tricks to clean up the plot
par(
  mfrow = c(2,2),     # set up 2x2 regions
  mar = c(3,3,3,1),   # set narrower margins
  xaxs = "i",         # remove "x-buffer"
  yaxs = "i",         # remove "y-buffer"
  mgp = c(2,0.4,0),   # bring in axis titles ([1]) and tick labels ([2])
  tcl = -0.25         # shorten tick marks
)

plot(densty ~ lengths, type = "l", lwd = 3, main = "dnorm()",
     xlab = "Fish Length (mm)", ylab = "Density", las = 1,
     yaxt = "n") # turns off y-axis
axis(side = 2, at = c(0.002, 0.006), labels = c(0.002, 0.006), las = 2)
plot(cuprob ~ lengths, type = "l", lwd = 3, main = "pnorm()",
     xlab = "Fish Length (mm)", ylab = "Cumulative Probability", las = 1)
plot(quants ~ cprobs, type = "l", lwd = 3, main = "qnorm()",
     xlab = "P", ylab = "P Quantile Length (mm)", las = 1)
hist(random, breaks = 50, col = "grey", main = "rnorm()",
     xlab = "Fish Length (mm)", ylab = "Frequency", las = 1)
box() # add borders to the histogram
```

Notice that `pnorm()` and `qnorm()` are inverses of one another: if you put the output of one into the output of the other, you get the original input back:

```
qnorm(pnorm(0))
```

```
## [1] 0
```

`pnorm(0)` asks R to find the probability that x is less than zero for the standard normal distribution ($N(0,1)$ - this is the default if you don't specify `mean` and `sig`). `qnorm(pnorm(0))` asks R to find the value of x that `pnorm(0)` * 100% of the possible values fall below. If the nesting is confusing, this line is the same as:

```
p = pnorm(0)
qnorm(p)
```

3.4 Bonus Topic: Non-linear Regression

You fitted linear and logistic regression models in Sections 3.1.1 and 3.2.1, however, R allows you to fit non-linear regression models as well.

First, read the data into R:

```
dat = read.csv("../Data/feeding.csv"); summary(dat)
```

```
##       prey            cons
##  Min.   : 1.00   Min.   : 1.00
##  1st Qu.:11.25   1st Qu.: 8.00
##  Median :26.00   Median :11.00
##  Mean   :25.08   Mean   : 9.92
##  3rd Qu.:37.75   3rd Qu.:13.00
##  Max.   :49.00   Max.   :15.00
```

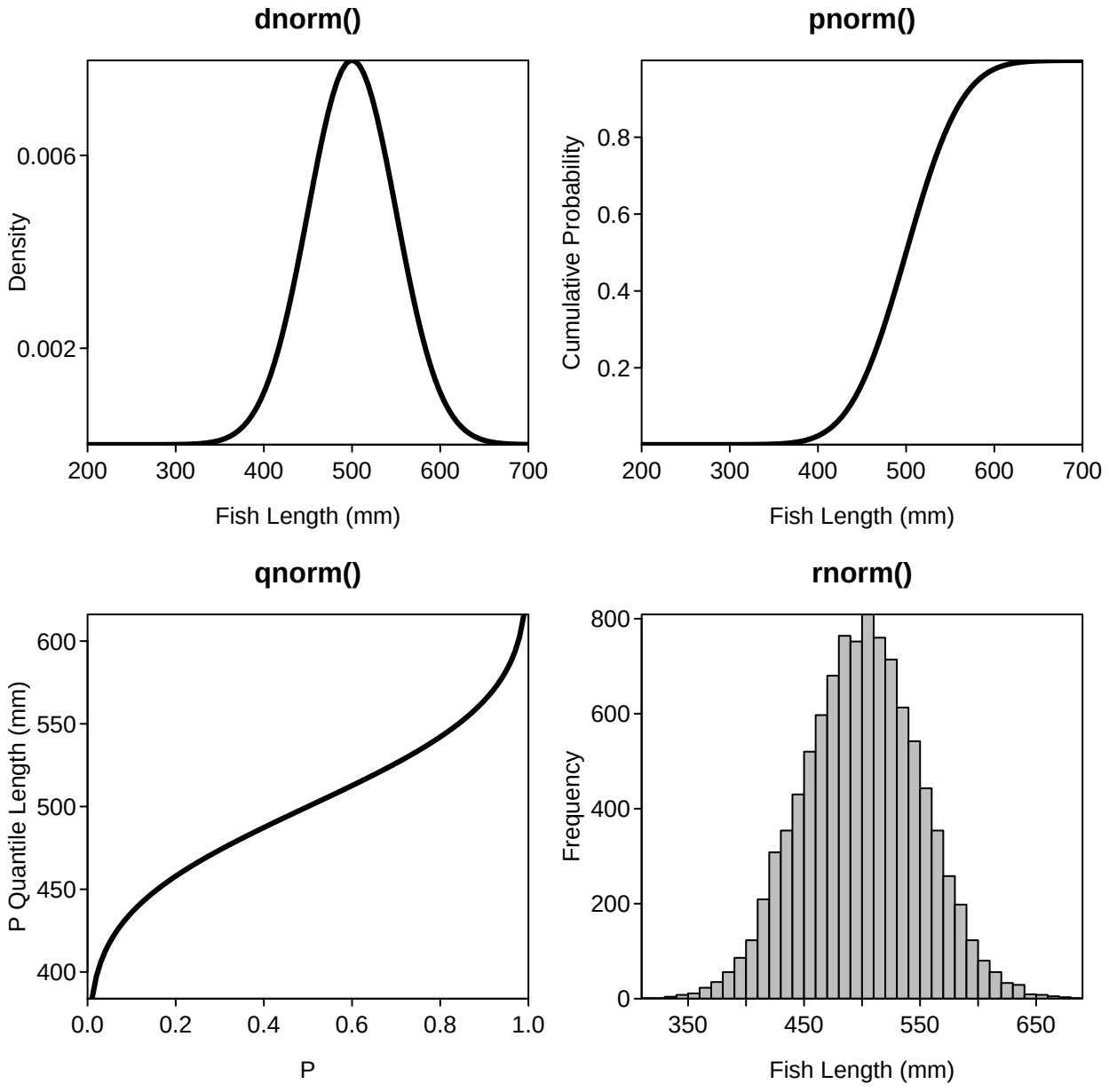
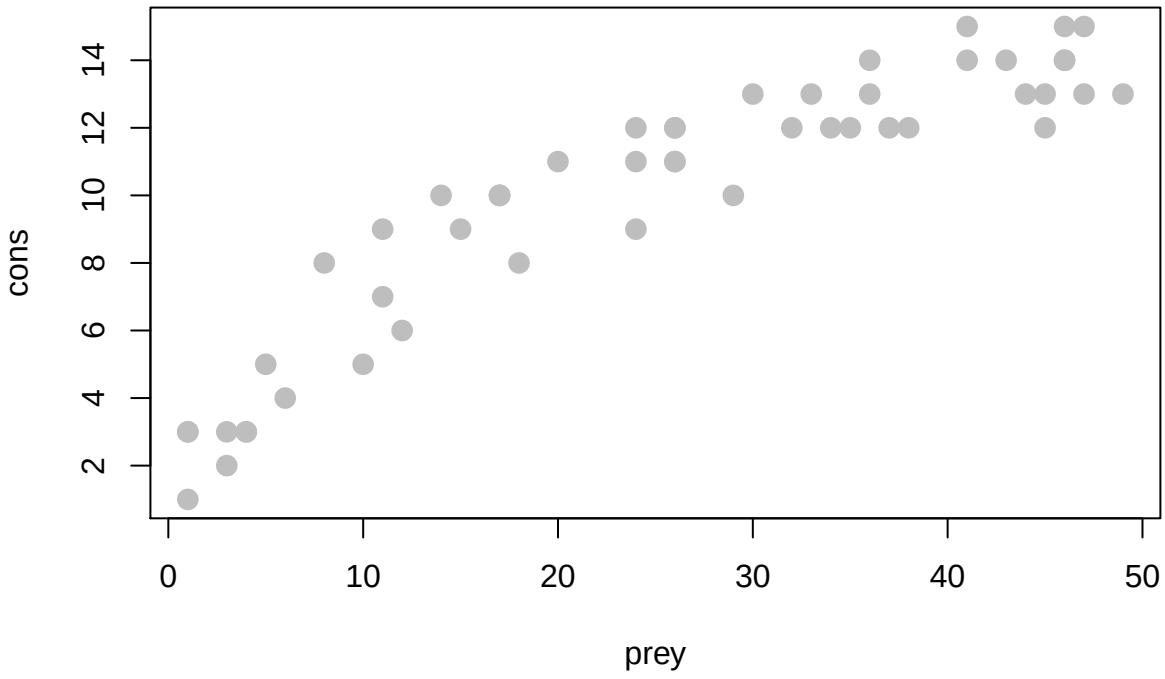


Figure 3.1: The four '-norm' functions with input (x-axis) and output (y-axis) displayed.

These are hypothetical data from an experiment in which you were interested in quantifying the functional feeding response[7] of a fish predator on zooplankton in an aquarium. You experimentally manipulated the prey density (`prey`) and counted how many prey items were consumed (`cons`).

Plot the data:

```
plot(cons ~ prey, data = dat, cex = 1.5, pch = 16, col = "grey")
```



You can see a distinct non-linearity to the relationship. The Holling Type II functional response[8] has this functional form:

$$y_i = \frac{ax_i}{1 + ahx_i} + \varepsilon_i, \varepsilon_i \sim N(0, \sigma) \tag{3.9}$$

where x_i is `prey` and y_i is `cons`.

You can fit this model in R using the `nls()` function:

```
fit = nls(cons ~ (a * prey)/(1 + a * h * prey), data = dat,
          start = c(a = 3, h = 0.1))
```

In general, it behaves very similarly to the `lm()` function, however there are a few differences:

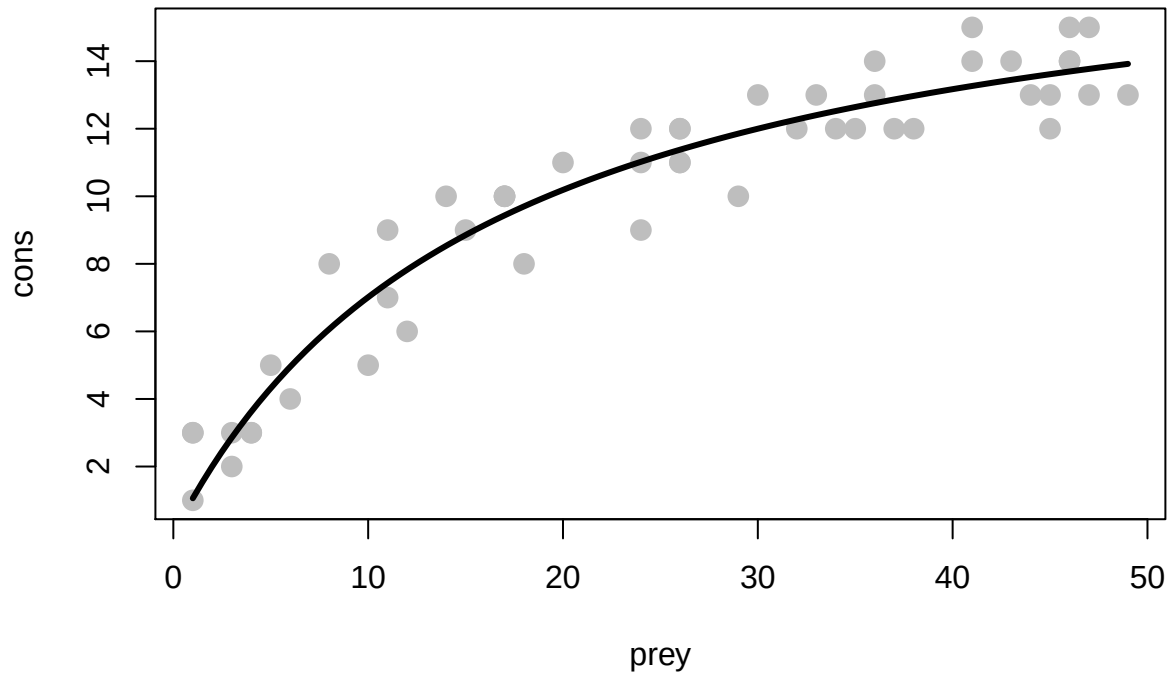
- You need to specify the functional form of the curve you are attempting to fit. In using `lm()`, the terms are all additive (e.g., `type + weight`), but in using `nls()`, this is not the case. For example, note the use of division.

[7] A functional response is the number of prey consumed by a predator at various prey densities
[8] This function rises quickly at low prey densities, but saturates at high densities

- You may need to provide starting values for the parameters (coefficients) you are estimating. This is because `nls()` will use a search algorithm to find the parameters of the best fit line, and it may need to have a reasonable idea of where to start looking for it to work properly.
- You cannot plot the fit using `abline()` anymore, because you have more parameters than just a slope and intercept, and the relationship between x and y is no longer linear.

Despite these differences, you can obtain similar output as from `lm()` by using the `summary()`, `coef()`, and `predict()` functions. Draw the fitted line over top of the data:

```
prey_seq = seq(min(dat$prey), max(dat$prey), length = 100)
cons_seq = predict(fit, newdata = data.frame(prey = prey_seq))

plot(cons ~ prey, data = dat, cex = 1.5, pch = 16, col = "grey")
lines(cons_seq ~ prey_seq, lwd = 3)
```



Exercise 3

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

1. Make the same graphic as in Figure 3.1 with at least one of the other distributions listed in Table 3.1 (other than the multinomial - being a multivariate distribution, it wouldn't work well with this code). Try thinking of a variable from your work that meets the uses of each distribution in Table 3.1 (or one

that's not listed). If you run into trouble, check out the help file for that distribution[9].

[9]Executing `?rnorm` or any other of the `-norm()` functions will take you to a page with info on all four function types for that distribution

Chapter 4

Monte Carlo Methods

Chapter Overview

Simulation modeling is one of the primary reasons to move away from spreadsheet-type programs (like Microsoft Excel) and into a program like R. R allows you to replicate the same (possibly complex and detailed) calculations over and over with slightly different random values. You can then summarize and plot the results of these replicated calculations all within the same program. Analyses of this type are **Monte Carlo methods**: they randomly sample from a set of quanities for the purpose of generating and summarizing a distribution of some statistic related to the sampled quantities. If this concept is confusing, hopefully this chapter will clarify.

In this chapter, you will learn the basic skills needed for simulation (i.e., Monte Carlo) modeling in R including:

- introduce randomness to a model
- repeat calculations many times
- summarization of many values from a distribution
- more advanced function writing

IMPORTANT NOTE: If you did not attend the sessions corresponding to Chapters 1 or 2 or 3, you are recommended to walk through the material found in those chapters before proceeding to this material. Remember that if you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this book.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch4.R` and save it in the directory `C:/Users/YOU/Documents/R-Book/Chapter4`. Set your working directory to that location. Revisit the material in Sections 1.2 and 1.3 for more details on these steps.

Layout of This Chapter

This chapter is divided into two main sections:

- **Required Material** (Sections 4.1 and 4.5) which is necessary to understand the examples in this chapter and the subsequent chapters

- **Example Cases** (Sections 4.6 and 4.7) which apply the skills learned in the required material. In the workshop session, you will walkthrough 2-3 of these example cases at the choice of the group of the participants. If you are interested in simulation modeling, you are suggested to work through all of the example cases, as slightly different tricks will be shown in the different examples.

4.1 Introducing Randomness

A critical part of simulation modeling is the use of random processes. A **random process** is one that generates a different outcome according to some rules each time it is executed. They are tightly linked to the concept of **uncertainty**: you are unsure about the outcome the next time the process is executed. There are two basic ways to introduce randomness in R: **random deviates** and **resampling**.

4.1.1 Random deviates

In Section 3.3, you learned about using probability distributions in R. One of the uses was the `r-` family of distribution functions. These functions create random numbers following a random process specified by a probability distribution.

Consider animal survival as an example. At the end of each year, each individual alive at the start can either live or die. There are two outcomes here, and suppose each animal has an 80% chance of surviving. The number of individuals that survive is the result of a **binomial random process** in which there were n individuals alive at the start of this year and p is the probability that any one individual survives to the next year. You can execute one binomial random process where $p = 0.8$ and $n = 100$ like this:

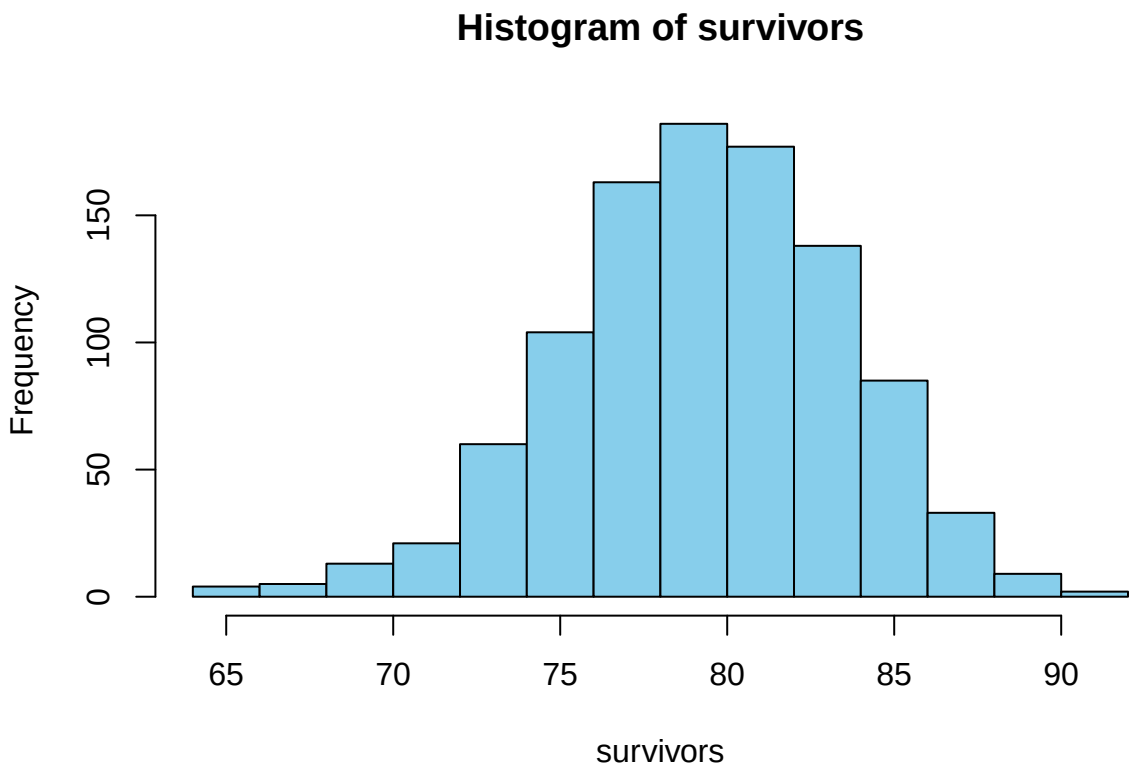
```
rbinom(n = 1, size = 100, prob = 0.8)
```

```
## [1] 76
```

The result you get will almost certainly be different from the one printed here. That is the random component.

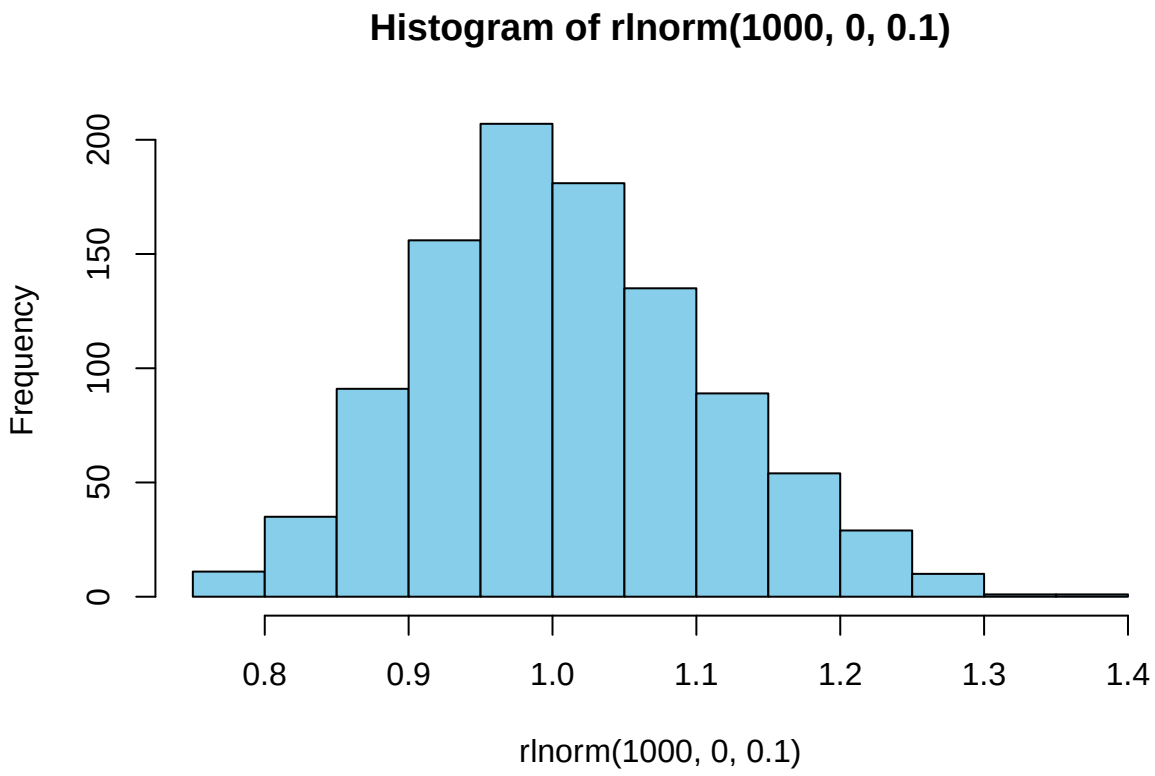
You can execute many such binomial processes by changing the `n` argument. Plot the distribution of expected surviving individuals:

```
survivors = rbinom(1000, 100, 0.8)
hist(survivors, col = "skyblue")
```



Another random process is the **lognormal process**: it generates random numbers such that the log of the values are normally-distributed with mean equal to `logmean` and standard deviation equal to `logsd`:

```
hist(rlnorm(1000, 0, 0.1), col = "skyblue")
```



There are many random processes you can use in R. Checkout Table 3.1 for more examples as well as the help files for each individual function for more details.

4.1.2 Resampling

Using random deviates works great for creating new random numbers, but what if you already have a set of numbers that you wish to introduce randomness to? For this, you can use **resampling techniques**. In R, the `sample` function is used to sample `size` elements from the vector `x`:

```
sample(x = 1:10, size = 5)
```

```
## [1] 2 4 9 6 1
```

You can sample with replacement (where it is possible to sample the same element two or more times):

```
sample(x = c("a", "b", "c"), size = 10, replace = T)
```

```
##  [1] "b" "b" "c" "c" "c" "b" "c" "a" "c" "b"
```

You can set probabilities on the sampling of different elements[1]:

```
sample(x = c("live", "die"), size = 10, replace = T,
       prob = c(0.8, 0.2))
```

```
##  [1] "die"  "live" "live" "live" "die"  "live" "live" "live" "live" "live"
```

Notice that this is the same as the binomial random process above, but with only 10 trials and the printing of the outcomes rather than the number of successes.

[1] If `prob` doesn't sum to 1, then it will be rescaled: `prob = prob/sum(prob)`

4.2 Reproducing randomness

For reproducibility purposes, you may wish to get the same exact random numbers each time you run your script. To do this, you need to set the **random seed**, which is the starting point of the random number generator your computer uses. If you run these two lines of code, you should get the same result as printed here:

```
set.seed(1234)
rnorm(1)
```

```
## [1] -1.207066
```

4.3 Replication

To use Monte Carlo methods, you need to be able to replicate some random process many times. There are two main ways this is commonly done: either with`replicate` or with `for` loops.

4.3.1 `replicate`

The `replicate` function executes some expression many times and returns the output from each execution. Say we have a vector `x`, which represents 30 observations of fish length (mm):

```
x = rnorm(30, 500, 30)
```

We wish to build the sampling distribution of the mean length "by hand". We can sample randomly from it, calculate the mean, then repeat this process many times:

```
means = replicate(n = 1000, expr = {
  x_i = sample(x, length(x), replace = T)
  mean(x_i)
})
```

If we take `mean(means)` and `sd(means)`, that should be very similar to `mean(x)` and `se(x)`. Create the `se` function (also shown in Section 2.11) and prove this to yourself:

```
se = function(x) sd(x)/sqrt(length(x))
mean(means); mean(x)
```

```
## [1] 493.8096
```

```
## [1] 493.4166
```

```
sd(means); se(x)
```

```
## [1] 4.899929
```

```
## [1] 5.044153
```

4.3.2 The `for` loop

In programming, a *loop* is a command that does something over and over until it reaches some point that you specify. R has a few types of loops: `repeat`, `while`, and `for`, to name a few. `for` loops are among the most common in simulation modeling. A `for` loop repeats some action for however many times you tell it for each value in some vector. The syntax is:

```
for (var in seq) {
  expression(var)
}
```

The loop calculates the expression for values of `var` for each element in the vector `seq`. For example:

```
for (i in 1:5) {
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
```

The `print` command will be executed 5 times: once for each value of `i`. It is the same as:

```
i = 1; print(i^2); i = 2; print(i^2); i = 3; print(i^2); i = 4; print(i^2); i = 5; print(i^2)
```

If you remove the `print()` function, see what happens:

```
for (i in 1:5) {
  i^2
}
```

Nothing is printed to the console. R did the calculation, but did not show you or store the result Often, you'll need to store the results of the calculation in a **container object**:

```
results = numeric(5)
```

This makes an empty numeric vector of length 5 that are all 0's. You can store the output of your loop calculations in `results`:

```
for (i in 1:5) {
  results[i]=i^2
}
results
```

```
## [1]  1  4  9 16 25
```

When `i^2` is calculated, it will be placed in the element `results[i]`. This was a trivial example, because you should do things like this using R's vectorized calculation framework: `(1:5)^2` (see Section 1.6).

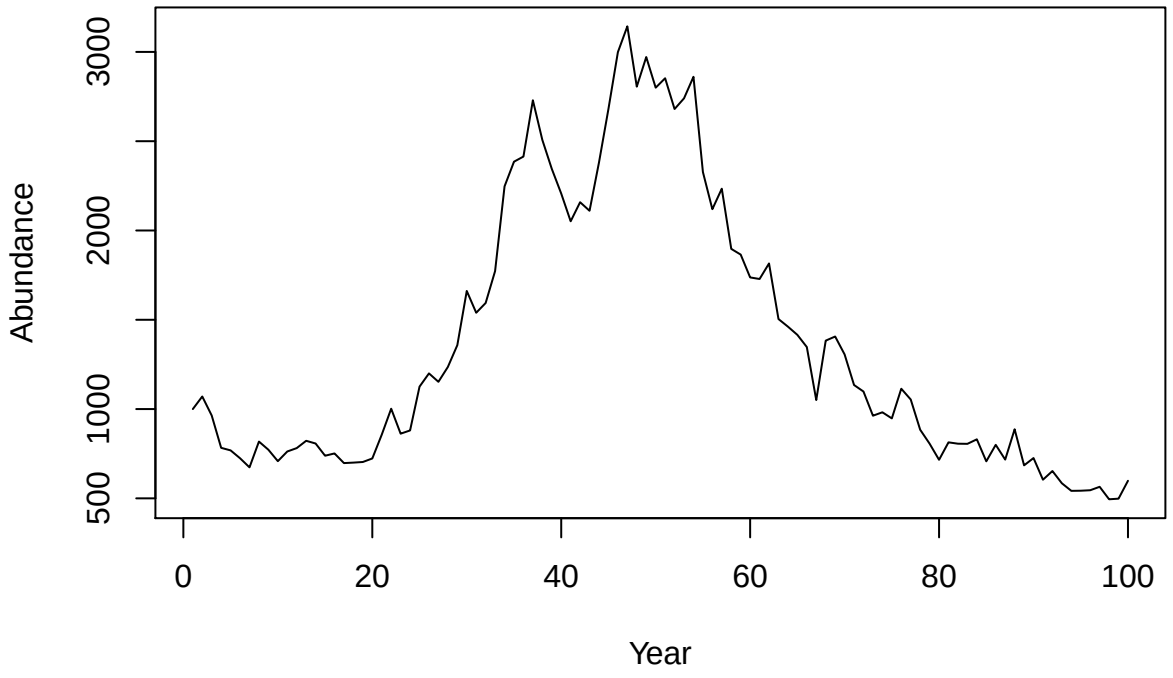
However, there are times where it is advantageous to use a loop. Particularly in cases where:

1. the calculations in one element are determined from the value in previous elements, such as in time series models
2. the calculations have multiple steps
3. you wish to store multiple results
4. you wish to track the progress of your calculations

As an illustration for item (1) above, build a (very) basic population model. At the start of the first year, the population abundance is 1000 individuals and grows by an average factor of 1.1 per year (reproduction and death processes result in a growth rate of 10%) before harvest. The growth rate varies randomly, however. Each year, the 1.1 growth factor has variability introduced by small changes in survival and reproductive process. Model these variations as lognormal random variables. After production, 8% of the population is harvested. Simulate and plot the abundance at the end of the year for 100 years:

```
nt = 100         # number of years
N = NULL        # container for abundance
```

```
N[1] = 1000    # first end-of-year abundance

for (t in 2:nt) {
  # N this year is N last year * growth *
    # randomness * fraction that survive harvest
  N[t] = (N[t-1] * 1.1 * rlnorm(1, 0, 0.1)) * (1 - 0.08)
}

# plot
plot(N, type = "l", pch = 15, xlab = "Year", ylab = "Abundance")
```



Examples of the other three utilities are shown in the example cases.

4.4 Function Writing

In Monte Carlo analyses, it is often useful to wrap code into functions. This makes them easy to be replicated and have the settings adjusted. As an example, turn the population model shown above into a function:

```
pop_sim = function(nt, grow, sd_grow, U, plot = F) {
  N = NULL
  N[1] = 1000

  for (t in 2:nt) {
    N[t] = (N[t-1] * grow * rlnorm(1, 0, sd_grow)) * (1 - U)
  }
```

```
  if (plot) {
    plot(N, type = "l", pch = 15, xlab = "Year", ylab = "Abundance")
  }

  N
}
```

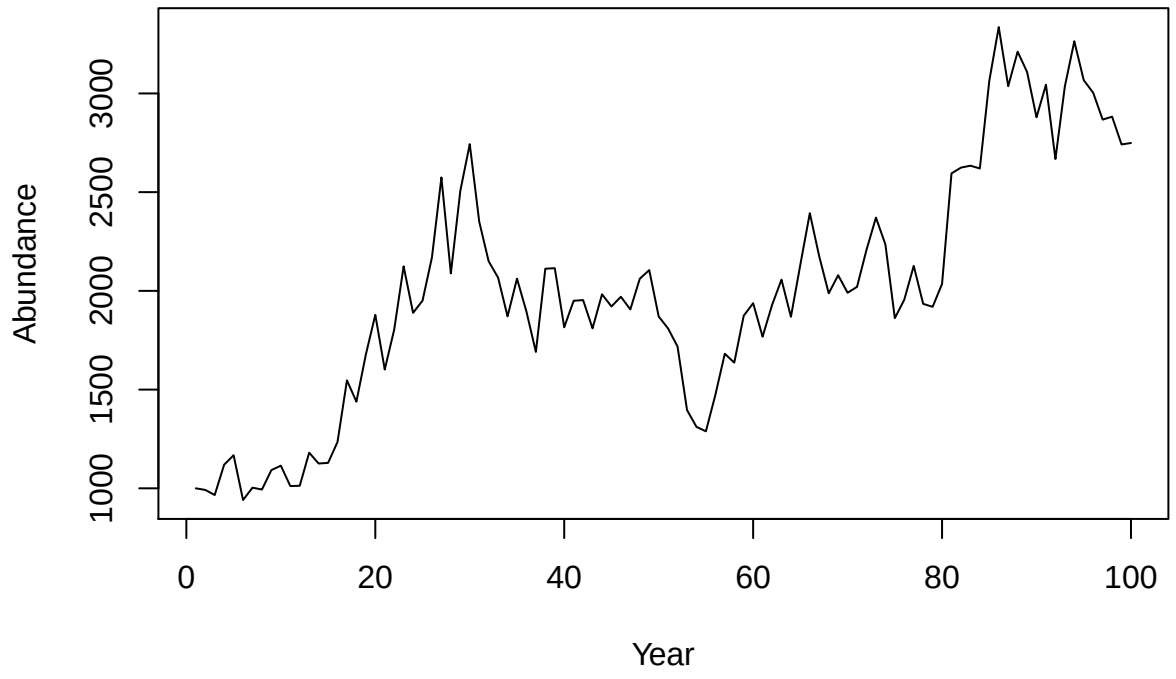
This function takes five inputs:

- `nt`: the number of years,
- `grow`: the population growth rate,
- `sd_grow`: the amount of annual variability in the growth rate
- `U`: the annual exploitation rate
- `plot`: whether you wish to have a plot created. It has a default setting of `FALSE`: if you don't specify `plot = T` when you call `pop_sim`, you won't see a plot made.

It returns one output: the vector of population abundance.

Use your function once using the same settings as before:

```
pop_sim(100, 1.1, 0.1, 0.08, T)
```




```
##   [1] 1000.0000  991.9189  966.1673 1119.8166 1167.4864  940.7572 1003.4028
##   [8]  993.7741 1092.3542 1114.6248 1011.6523 1012.8149 1180.6373 1125.6400
##  [15] 1128.9152 1234.7466 1546.7473 1438.8662 1677.2920 1878.7559 1601.6186
##  [22] 1803.7180 2124.3262 1888.6512 1950.4458 2170.6249 2574.6142 2088.5542
##  [29] 2506.9833 2743.3092 2350.9056 2150.1135 2067.0640 1870.8924 2061.8156
##  [36] 1895.1914 1690.9365 2112.3612 2114.7401 1815.7680 1949.8661 1953.3604
```

```
##  [43] 1810.1937 1982.6345 1920.9718 1970.1317 1905.9889 2061.5250 2105.1941
##  [50] 1870.4381 1809.3494 1717.6727 1396.3874 1311.3019 1288.7084 1472.9669
##  [57] 1681.8080 1636.4883 1874.0338 1937.6544 1767.9563 1927.9457 2056.8504
##  [64] 1868.7910 2133.6614 2393.3918 2174.1558 1987.7674 2079.1235 1990.4753
##  [71] 2020.4988 2207.8347 2371.1186 2235.4894 1861.8581 1955.1694 2126.6954
##  [78] 1934.2892 1919.1523 2035.2788 2595.0493 2624.3191 2634.0353 2619.8963
##  [85] 3063.5877 3335.9375 3036.8140 3211.4930 3108.6992 2879.1979 3044.2377
##  [92] 2667.8093 3036.1608 3264.0875 3067.2281 3003.3483 2867.4153 2882.2673
##  [99] 2741.6617 2748.9291
```

Now, you wish to replicate exectuing this function 1000 times. Use the `replicate` function to do this:

```
out = replicate(n = 1000, expr = pop_sim(100, 1.1, 0.1, 0.08, F))
```

If you do `dim(out)`, you'll see that rows are stored as years (there are 100 of them) and columns are stored as replicates (there are 1000 of them). Notice how wrapping the code in the function made the `replicate` call easy.

Here are some advantages of wrapping code like this into a function:

- If you do the same task over and over, you don't need to type all of the code to perform the task, just the function call.
- If you need to change the way the function behaves (mechanically in the function body), you only need to change it one place: in the function definition.
- You can easily change the settings of the code (e.g., whether you want to see the plot) in one place
- Function writing **can** lead to shorter scripts
- Function writing **can** lead to more readable code (if people know what your functions do)

4.5 Summarization

After replicating a calculation many times, you will need to summarize the results. Here are several examples using the `out` matrix.

4.5.1 Central Tendency

You can calculate the mean abundance each year across your iterations using the `apply` function (Section 1.9):

```
N_mean = apply(out, 1, mean)
N_mean[1:10]
```

```
##  [1] 1000.000 1017.511 1033.436 1054.619 1069.542 1086.413 1107.854
##  [8] 1129.619 1145.288 1161.254
```

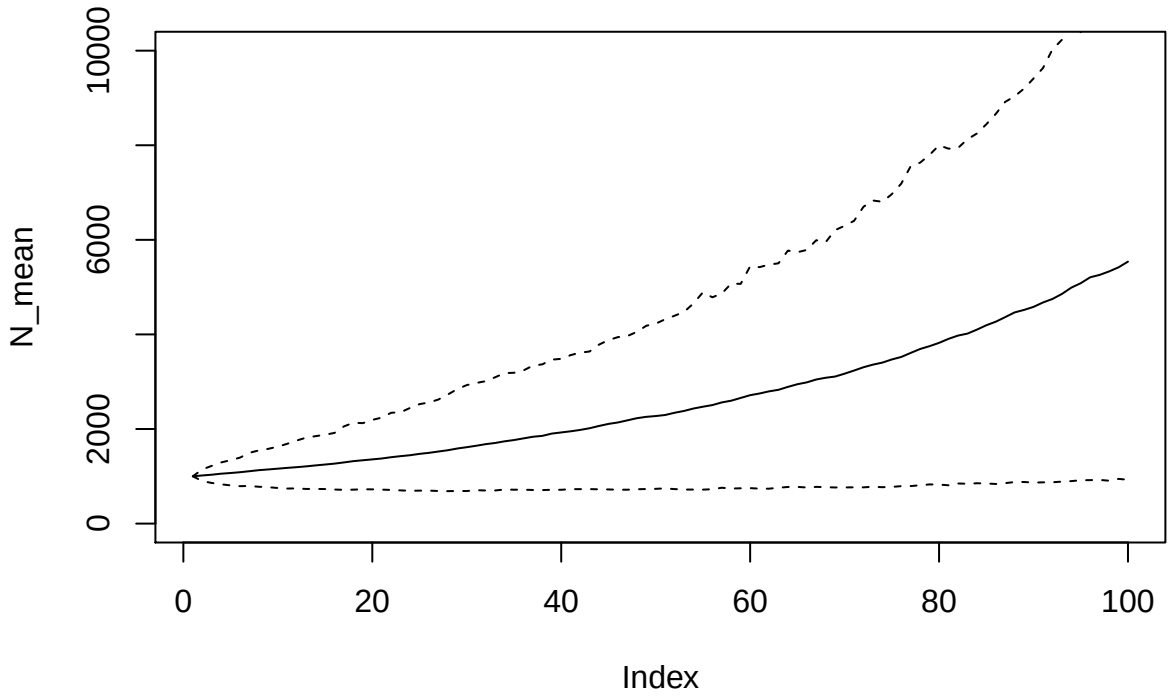
You could do the same thing using `median` rather than `mean`. Mode is more difficult to calculate in R, if you need to get the mode, try to Google it[2].

4.5.2 Variability

One of the primary reasons to conduct a Monte Carlo analysis is to obtain estimates of variability. You can summarize the variability easily using the `quantile` function:

[2]Google is an R programmer's best friend. There is a massive online community for R, and if you have a question on something, it has almost certainly been asked somewhere on the web.

```
N_quants = apply(out, 1, function(x) quantile(x, c(0.1, 0.9)))

plot(N_mean, type = "l", ylim = c(0, 10000))
lines(N_quants[1,], lty = 2)
lines(N_quants[2,], lty = 2)
```



Notice how a user-defined function was passed to `apply`. The range within the two dashed lines represents the range that encompassed the central 80% of the random abundances each year.

4.5.3 Frequencies

Often you will want to count how many times something happened. In some cases, the fraction of times something happened can be interpretted as a probability.

The `table` function is very useful for counting occurences of events. Suppose you are interested in how many of your iterations resulted in fewer than 1000 individuals at year 10:

```
out10 = ifelse(out[10,] < 1000, "less10", "greater10")
table(out10)
```

```
## out10
## greater10    less10
##       643       357
```

Suppose you are also interested in how many of your iterations resulted in fewer than 1100 individuals at year 20:

```
out20 = ifelse(out[20,] < 1100, "less20", "greater20")
table(out20)
```

```
## out20
## greater20    less20
##       592       408
```

Now suppose you are interested in how these two metrics are related:

```
table(out10, out20)
```

```
##            out20
## out10       greater20 less20
##   greater10       502    141
##   less10           90    267
```

As an example in iterpretting this output, most often populations that were greater than 1000 at year 10 were also greater than 1100 at year 20. If a population was less than 1000 at year 10, it was more likely to be less than 1100 at year 20 than to be greater than it.

You can turn these into probabilities (if you believe your model represents reality) by dividing each cell by the total number of iterations:

```
round(table(out10, out20)/1000, 2)
```

```
##            out20
## out10       greater20 less20
##   greater10      0.50   0.14
##   less10         0.09   0.27
```

4.6 Simulation-Based Examples

4.6.1 Test `rnorm`

In this example, you will verify that the function `rnorm` works the same way that `qnorm` and `pnorm` work. Hopefully it will also reinforce the way the random, quantile, and cumulative distribution functions work in R.

First, specify the mean and standard deviation for this example:

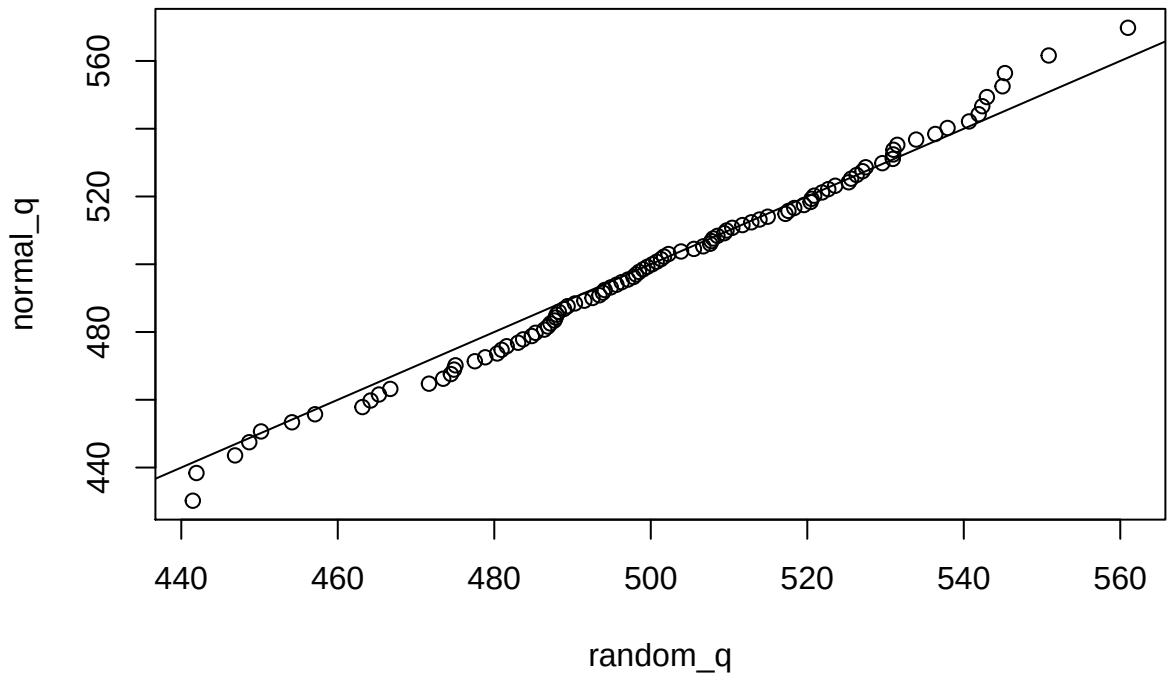
```
mu = 500; sig = 30
```

Now make up `n` (any number of your choosing, something greater than 10) random deviates from this normal distribution:

```
random = rnorm(100, mu, sig)
```

Test the quantiles (obtain the values that `p` * 100% of the quantities fall below, both for random numbers and from the `qnorm` function):

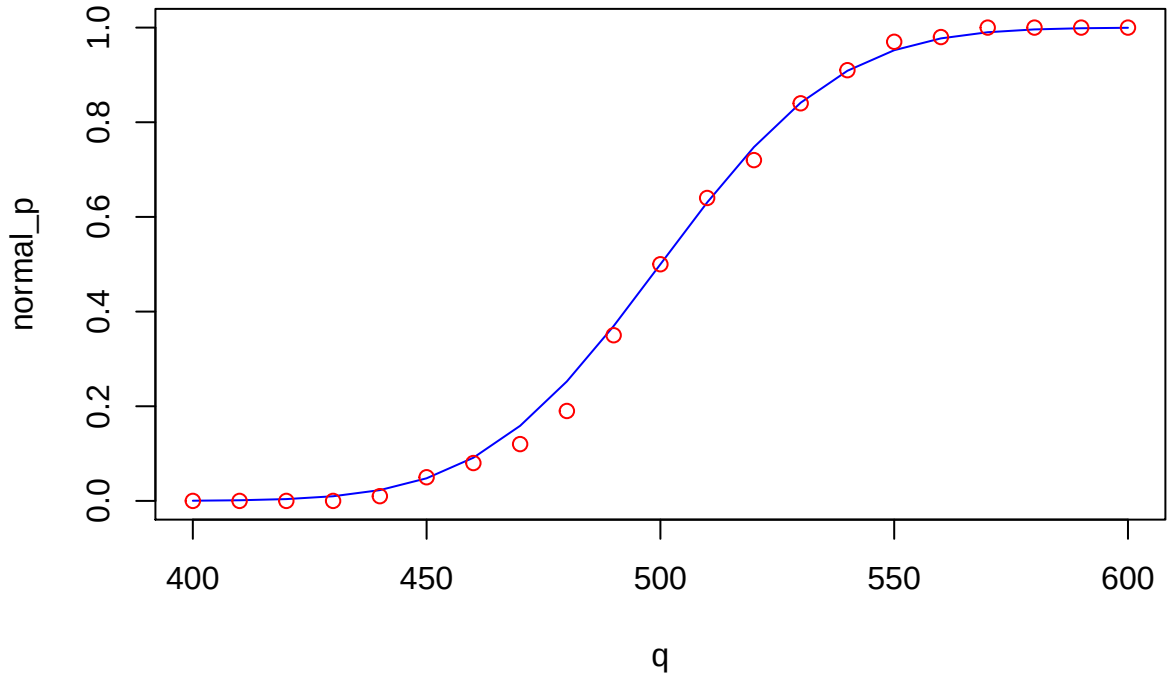
```
p = seq(0.01, 0.99, 0.01)
random_q = quantile(random, p)
normal_q = qnorm(p, mu, sig)
plot(normal_q ~ random_q); abline(c(0,1))
```



The fact that all the quantiles fall around the 1:1 line sugggests the `n` random samples samples are indeed from a normal distribution. Any variabilities you see are due to sampling errors. If you increase `n` to `n = 1e6` (one million), you'll see no deviations. This is called a **q-q plot**, and is frequently used to assess the fit of data to a distribution.

Now test the random values in their agreement with the `pnorm` function. Plot the cumulative density functions for the truely normal curve and that approximated by the random deviates:

```
q = seq(400, 600, 10)
random_cdf = ecdf(random)
random_p = random_cdf(q)
normal_p = pnorm(q, mu, sig)
plot(normal_p ~ q, type = "l", col = "blue")
points(random_p ~ q, col = "red")
```



The `ecdf` function obtains the empirical cumulative density function (which is just `pnorm` for a sample). It allows you to plug in any random variable and obtain the probability of having one less than it.

4.6.2 Stochastic Power Analysis

A **power analysis** is one where the analyst wishes to determine how much power they will have to detect an effect. Having high power means ensuring you do not falsely reject a true hypothesis (e.g., claiming that there is not effect based on a p-value greater than 0.05).

You can conduct a power analysis using stochastic simulation (i.e., a Monte Carlo analysis). Here, you will write a power analysis to determine how likely are you to be able to correctly identify what you deem to be a biologically-meaningful difference in survival between two tagging procedures.

You know one tagging procedure has approximately a 10% mortality rate (10% of tagged fish die within the first 12 hours as result of the tagging process). Another, cheaper and less labor-intensive method has been proposed but before implementing it, your agency wishes to determine if it will have a meaningful impact on the reliability of the study or the efficiency of the tagging crew to tag individuals that will be alive long enough to be useful. You and your colleagues determine that if the mortality rate reaches 25%, then gains in time and cost efficiency would be offset by needing to tag more fish (because more will die). You have decided to perform a small-scale study to determine if the new method affects mortality enough to result in 25% or more mortality. The study will tag `n` individuals using each method (new and old) and track the fraction that survived after 12 hours. Before performing the study however, you deem it important to determine how large `n` needs to be to answer this question. You decide to use a stochastic power analysis based on what you've learned in this book to help your research group. The small-scale study can tag a total of at most 100 fish with the currently available resources. Could you tag fewer than 100 total individuals and still have a high probability of correctly identifying an effect of this size?

The stochastic power analysis approach works like this (this is called **psuedocode**):

1. Simulate data under the reality that the difference is real with `n` observations per treatment, where `n < 30/2`
2. Fit the model that will be used when the real data are collected
3. Determine if the effect was detected with a significant p-value
4. Replicate steps 1 - 3 many times
5. Replicate step 4 while varying `n` over the interval from 10 to 50
6. Determine what fraction of the p-values were deemed significant at each `n`

Step 2 will require fitting a generalized linear model, for a review revisit Section 3.2 (specifically Section 3.2.1 on logistic regression).

First, create a function that will generate data, fit the model, and determine if the p-value is significant (steps 1-3 above):

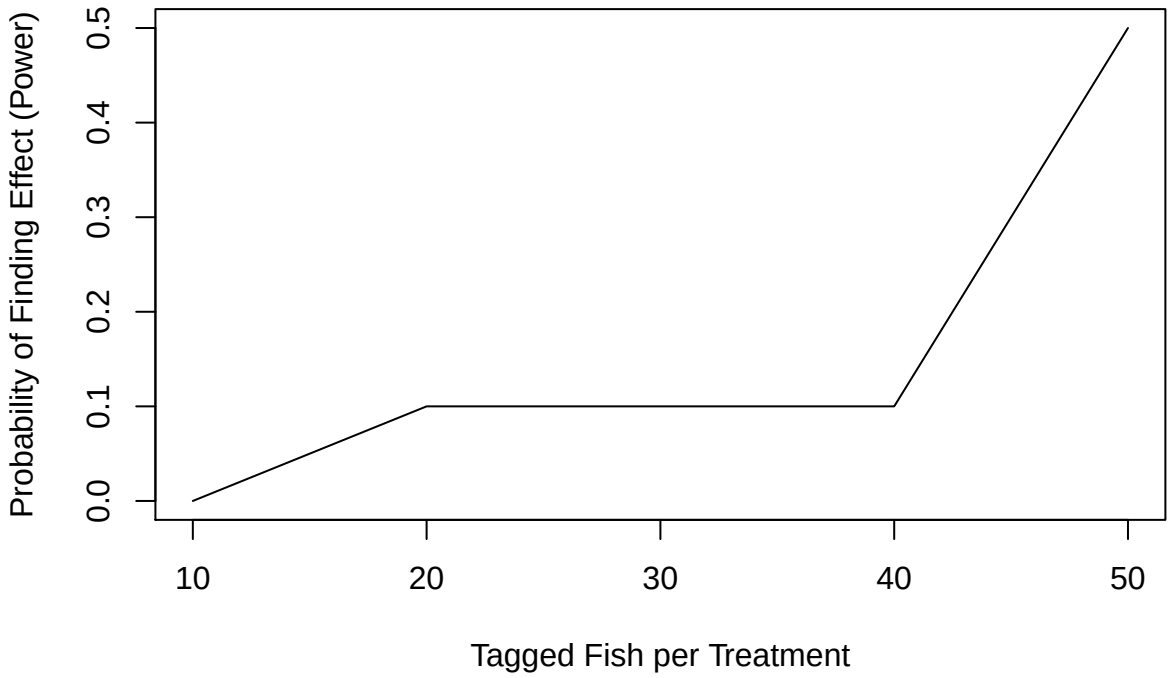
```
sim_fit = function(n, p_old = 0.10, p_new = 0.25) {

  ### step 1: create the data ###
  # generate random response data
  dead_old = rbinom(n, size = 1, prob = p_old)
  dead_new = rbinom(n, size = 1, prob = p_new)
  # create the predictor variable
  method = rep(c("old", "new"), each = n)
  # create a data.frame to pass to glm
  df = data.frame(dead = c(dead_old, dead_new), method = method)
  # relevel so old is the reference
  df$method = relevel(df$method, ref = "old")

  ### step 2: fit the model ###
  fit = glm(dead ~ method, data = df, family = binomial)

  ### step 3: determine if a sig. p-value was found ###
  # extract the p-value
  pval = summary(fit)$coef[2,4]
  # determine if it was found to be significant
  pval < 0.05
}
```

Next, for steps 4 and 5, set up a **nested `for` loop**. This will have two loops: one that loops over sample sizes for step 5 and one that loops over replicates of each sample size (step 4):

```
I = 10  # the number of replicates at each sample size
n_try = seq(10, 50, 10)  # the test sample sizes
N = length(n_try)        # count them
# container:
out = matrix(NA, I, N) # matrix with I rows and N columns
for (n in 1:N) {
  for (i in 1:I) {
    out[i,n] = sim_fit(n = n_try[n])
  }
}
```

You now have a matrix of `TRUE` and `FALSE` elements that indicates whether a significant difference was found at the $\alpha = 0.05$ level if the effect was truely as large as you care about. You can obtain the proportion of all the replicates at each sample size that resulted in a significant difference using the `mean` function with `apply`:

```
plot(apply(out, 2, mean) ~ n_try, type = "l",
     xlab = "Tagged Fish per Treatment",
     ylab = "Probability of Finding Effect (Power)")
```

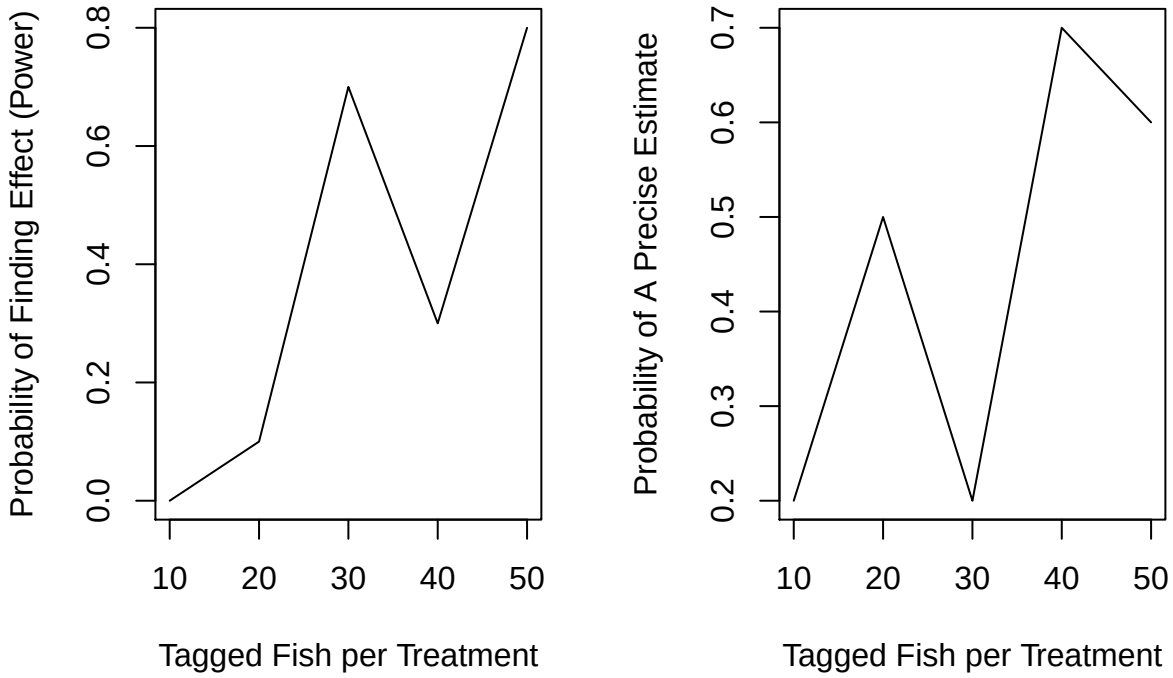


Even if you tagged 100 fish total, you would only have a 50% chance of saying the effect (which truely is there!) is present under the null hypothesis testing framework.

Suppose you and your colleagues aren't relying on p-values in this case, and are purely interested in how precisely the **effect size** would be estimated. Adapt your function to determine how frequently it is you would be able to estimate the true mortality probability of the new method within +/- 5% based on the point estimate only (the estimate for the tagging mortality of the new method must be between 0.2 and 0.3 for a successful study). Change your function to calculate this additional metric and re-run the analysis:

```
sim_fit = function(n, p_old = 0.10, p_new = 0.25) {
  # create the data
  dead_old = rbinom(n, size = 1, prob = p_old)
  dead_new = rbinom(n, size = 1, prob = p_new)
  # create the predictor variable
  method = rep(c("old", "new"), each = n)
  # create a data.frame to pass to glm
  df = data.frame(dead = c(dead_old, dead_new), method = method)
  # relevel so old is the reference
  df$method = relevel(df$method, ref = "old")
  # fit the model
  fit = glm(dead ~ method, data = df, family = binomial)
  # extract the p-value
  pval = summary(fit)$coef[2,4]
```

```
  # determine if it was found to be significant
  sig_pval = pval < 0.05
  # obtain the estimated mortality rate for the new method
  p_new_est = predict(fit, data.frame(method = c("new")),
                      type = "response")

  # determine if it is +/- 5% from the true value
  prc_est = p_new_est >= (p_new - 0.05) & p_new_est <= (p_new + 0.05)
  # return a vector with these two elements
  c(sig_pval = sig_pval, prc_est = unname(prc_est))
}

# run the analysis
I = 10  # the number of replicates at each sample size
n_try = seq(10, 50, 10)  # the test sample sizes
N = length(n_try)      # count them

# containers:
out_sig = matrix(NA, I, N) # matrix with I rows and N columns
out_prc = matrix(NA, I, N) # matrix with I rows and N columns
for (n in 1:N) {
  for (i in 1:I) {
    tmp = sim_fit(n = n_try[n])      # run sim
    out_sig[i,n] = tmp["sig_pval"]  # extract and store significance metric
    out_prc[i,n] = tmp["prc_est"]   # extract and store precision metric
  }
}

par(mfrow = c(1,2))
plot(apply(out_sig, 2, mean) ~ n_try, type = "l",
     xlab = "Tagged Fish per Treatment",
     ylab = "Probability of Finding Effect (Power)")
plot(apply(out_prc, 2, mean) ~ n_try, type = "l",
     xlab = "Tagged Fish per Treatment",
     ylab = "Probability of A Precise Estimate")
```



It seems that even if you tagged 50 fish per treatment, you would have a 60% chance of estimating that the mortality rate is between 0.2 and 0.3 if it was truly 0.25.

You and your colleagues consider these results and determine that you will need to somehow acquire more funds to tag more fish in the small-scale study in order to a high level of confidence in the results.

4.6.3 Harvest Policy Analysis

In this example, you will simulate population dynamics under a more realistic model than in Sections 4.3.2 and 4.4 for the purpose of evaluating different harvest policies.

Suppose you are a fisheries research biologist, and a commercial fishery for pink salmon (*Oncorhynchus gorbuscha*) takes place in your district. For the past 10 years, it has been fished with an exploitation rate of 40% (40% of the fish that return each year have been harvested, exploitation rate is abbreviated by U), resulting in an average annual harvest of 8.5 million fish. The management plan is up for evaluation this year, and your supervisor has asked you to prepare an analysis that determines if more harvest could be sustained if a different exploitation rate were to be used in the future.

Based on historical data, your best understanding implies that the stock is driven by Ricker spawner-recruit dynamics. That is, the total number of fish that return this year (recruits) are a function of the total number of spawners that produced and fertilized the eggs from which this year's recruits were produced. The Ricker model can be written this way:

$$R_t = \alpha S_{t-1} e^{-\beta S_{t-1} + \varepsilon_t}, \varepsilon_t \sim N(0, \sigma) \tag{4.1}$$

where α is a parameter representing the maximum recruits per spawner (obtained at very low spawner

abundances) and β is a measure of the strength of density-dependent mortality. Notice that the error term is in the exponent, which makes e^{ε_t} lognormal.

You have estimates of the parameters:

- $\alpha = 6$
- $\beta = 1 \times 10^{-7}$
- $\sigma = 0.4$

You decide that you can build a policy analysis by simulating the stock forward through time under different exploitation rates to determine if its reasonable to expect a different exploitation rate to provide more harvest than what is currently being extracted.

First, write a function for your population model. Your function must:

1. take the parameters, dimensions (number of years), and the policy variable (U) as input arguments
2. simulate the population using Ricker dynamics
3. calculate and return the sum of the yields over the number future years you simulated.

```
# Step #1: name function and give it some arguments
ricker_sim = function(ny, params, U) {
  # extract the parameters out by name:
  alpha = params["alpha"]
  beta = params["beta"]
  sigma = params["sigma"]
  # create containers:
  # yep, you can do this
  R = S = H = NULL
  # initialize the population in the first year
    # start the population at being fished at 40%
    # with lognormal error
  R[1] = log(alpha * (1 - 0.4))/(beta * (1 - 0.4)) * exp(rnorm(1, 0, sigma))
  S[1] = R[1] * (1 - U)
  H[1] = R[1] * U

  # carry simulation forward through time
  for (y in 2:ny) {
    # use the ricker function with random lognormal white noise
    R[y] = S[y-1] * alpha * exp(-beta * S[y-1] + rnorm(1, 0, sigma))
    #harvest and spawners are the same as before
    S[y] = R[y] * (1 - U)
    H[y] = R[y] * U
  }
  # wrap output in a list object
  list(
    mean_H = mean(H),
    mean_S = mean(S)
    )
}
```

Use the function once:

```
params = c(alpha = 6, beta = 1e-7, sigma = 0.4)
out = ricker_sim(U = 0.4, ny = 20, params = params)
#average annual harvest (in millions)
round(out$mean_S/1e6, digits = 2)
```

```
## [1] 13.54
```

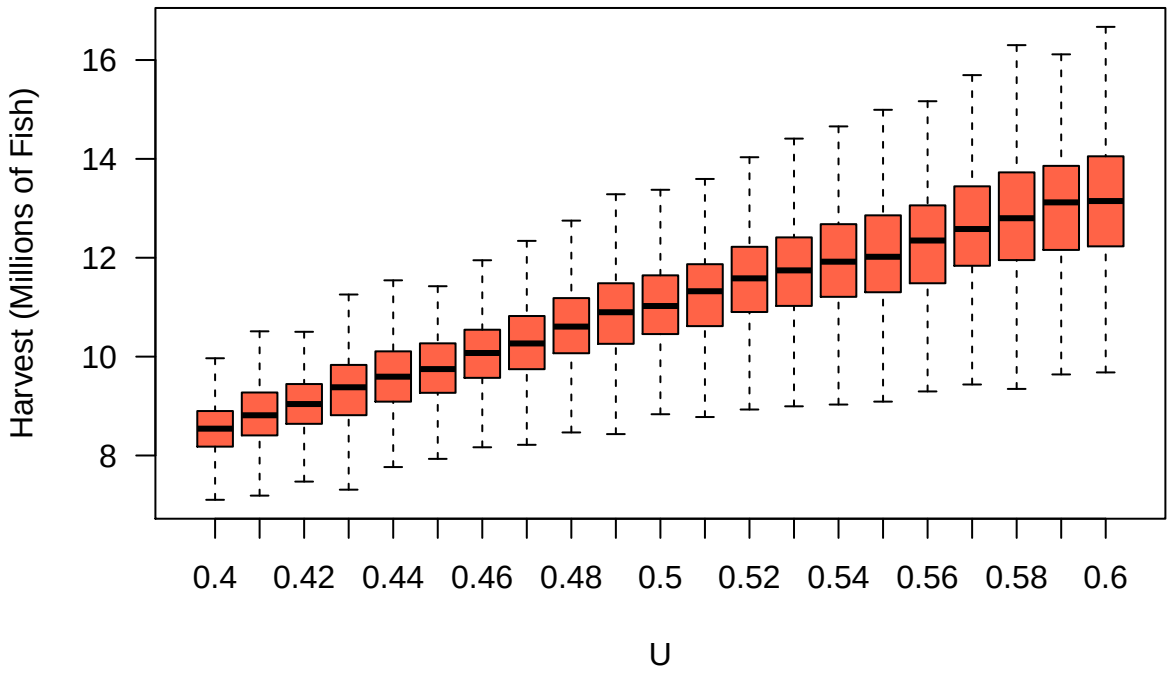
If you completed the stochastic power analysis example (Section 4.6.2), you might see where this is going. You are going to replicate applying a fixed policy many times to a random system. This is the Monte Carlo part of the analysis. The policy part is that you will compare the output from several candidate exploitation rates to inform a decision about which is best. This time, set up your analysis using `sapply` and `replicate` instead of performing a nested `for` loop as in previous examples:

```
U_try = seq(0.4, 0.6, 0.01)
n_rep = 500
H_out = sapply(U_try, function(u) {
  replicate(n = n_rep, expr = {
    ricker_sim(U = u, ny = 20, params = params)$mean_H/1e6
  })
})
```

The nested `replicate` and `sapply` method is a bit cleaner than a nested `for` loop, but you have less control over the format of the output.

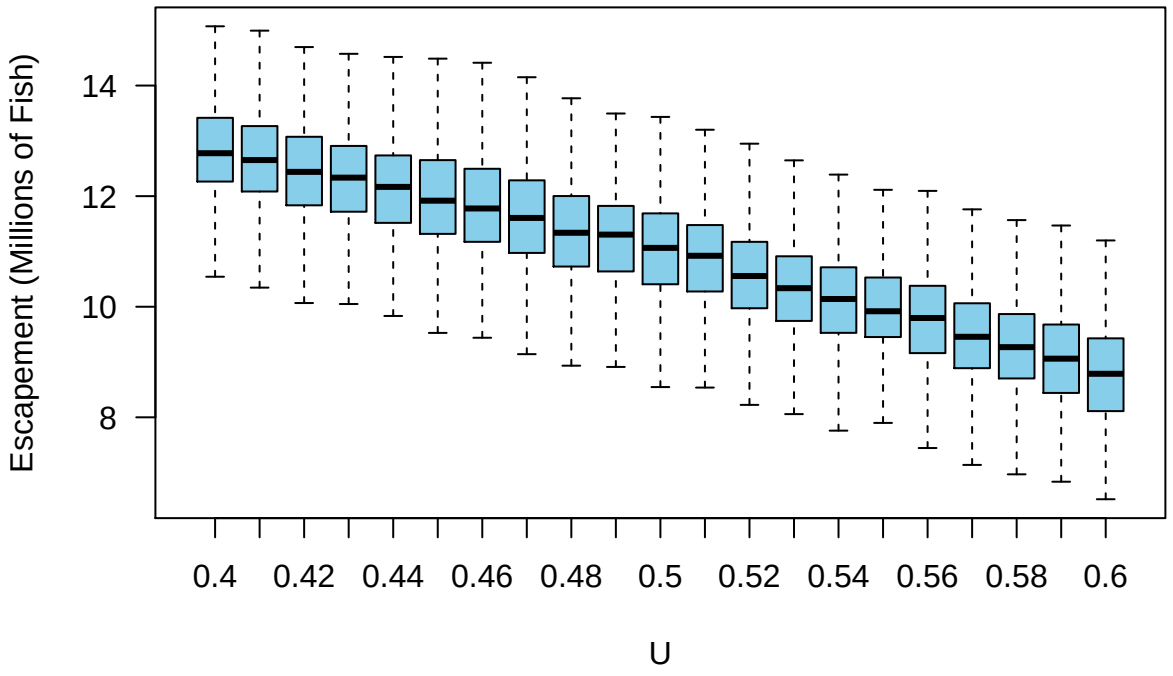
Plot the output of your simulations using a boxplot. To make things easier, give `H_out` column names representing the exploitation rate:

```
colnames(H_out) = U_try
boxplot(H_out, outline = F,
        xlab = "U", ylab = "Harvest (Millions of Fish)",
        col = "tomato", las = 1)
```



It appears the stock could produce more harvest than its current 8.5 million fish per year if it was fished harder. However, your supervisors also do not want to see the escapement drop below three-quarters of what it has been in recent history (75% of approximately 13 million fish). They ask you to obtain the expected

average annual escapement as well as harvest. You can simply re-run the code above, but extracting `S_mean` rather than `H_mean`. Call this output `S_out` and plot it just like harvest (if your curious, this blue color is `col = "skyblue"`):



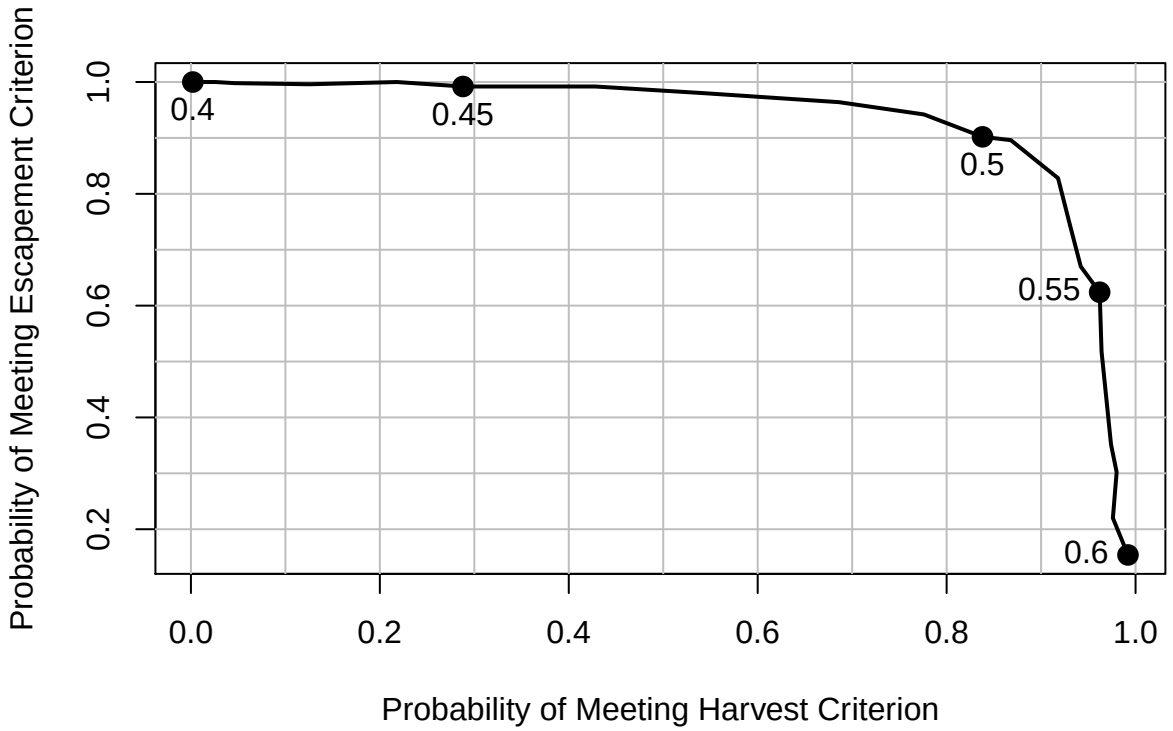
After seeing this information, your supervisor realizes they are faced with a trade-off: the stock could produce more with high exploitation rates, but they are concerned about pushing the stock too low for sustainability reasons. They tell you to determine the probability the average escapement would not be pushed below 75% of 13 million at each exploitation rate, as well as the probability that the average annual harvests will be at least 20% greater than they are currently (approximately 8.5 million fish). Given your output, this is easy:

```
# determine if each element meets escapement criterion
Smeet = S_out > (0.75 * 13)
# determine if each element meets harvest criterion
Hmeet = H_out > (1.2 * 8.5)
# calculate the probability of each occuring at a given exploitation rate
  # remember, mean of a logical vector calculate the proportion of TRUEs
p_Smeet = apply(Smeet, 2, mean)
p_Hmeet = apply(Hmeet, 2, mean)
```

You plot this for your supervisor as follows:

```
# the U levels to highlight on plot
plot_U = seq(0.4, 0.6, 0.05)
# create an empty plot
plot(p_Smeet ~ p_Hmeet, type = "n",
     xlab = "Probability of Meeting Harvest Criterion",
     ylab = "Probability of Meeting Escapement Criterion")
# add gridlines
```

```
abline(v = seq(0, 1, 0.1), col = "grey")
abline(h = seq(0, 1, 0.1), col = "grey")
#draw on the tradeoff curve
lines(p_Smeet ~ p_Hmeet, type = "l", lwd = 2)
# add points and text for particular U policies
points(p_Smeet[U_try %in% plot_U] ~ p_Hmeet[U_try %in% plot_U],
       pch = 16, cex = 1.5)
text(p_Smeet[U_try %in% plot_U] ~ p_Hmeet[U_try %in% plot_U],
     labels = U_try[U_try %in% plot_U], pos = c(1,1,1,2,2))
```



Equipped with this analysis, your supervisor plans to go to the policy-makers with the recommendation of adjusting the exploitation rate policy to use $U = 0.5$, because they think it balances the trade-off. Notice how if the status quo was maintained, your model suggests you would have complete certainty of staying where you are now: escapement will remain above 75% of its current level with a 100% chance, but you would have no chance of improving harvests to greater than 20% of their current level. Small increases in the exploitation rate (e.g., from 0.4 to 0.45) have a reasonably large gain in harvest performance, but hardly any losses for the escapement criterion. Your supervisor is willing to live with a 90% chance that the escapement will stay where they desire in order to gain a >80% chance of obtaining the desired amount of increases in harvest.

The utility of using Monte Carlo methods in this example is the ability to calculate the probability of some event you are interested in. There are analytical (i.e., not simulation-based) solutions to predict the annual harvest and escapement from a fixed U from a population with parameters α and β, but by incorporating randomness, you were able to obtain the relative weights of outcomes other than the expectation under the deterministic Ricker model, thereby allowing the assignment of probabilities to meeting the two criteria.

4.7 Resampling-Based Examples

4.7.1 The Bootstrap

Oftentimes, you'll have a fitted model that you wish to propagate the uncertainty from to some derived quantity. Consider the case of the **von Bertalanffy growth model**. This is a non-linear model used to predict the size of an organism (weight or length) based on its age. The model can be written for a non-linear regression model (see Section 3.4) as:

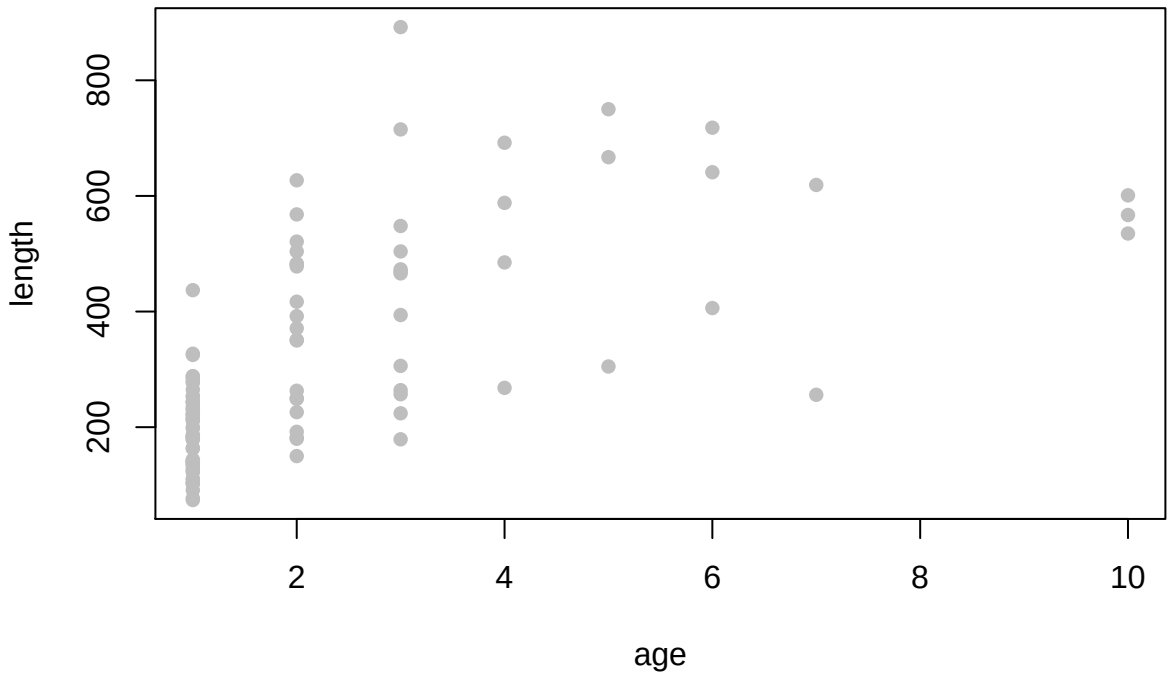
$$L_i = L_\infty \left(1 - e^{-k(age_i - t_0)}\right) + \varepsilon_i, \varepsilon_i \sim N(0, \sigma) \tag{4.2}$$

where L_i and age_i are the observed length and age of individual i, respectively, and L_∞, k, and t_0 are parameters to be estimated. The interpretations of the parameters are as follows:

- L_∞: the maximum average length acheived
- k: a growth coefficient linked to metabolic rate. It specifies the rate of increase in length as the fish ages early in life
- t_0: the theoretical age when length equals zero (the x-intercept).

Use the data set `growth.csv` for this example (see the instructions on aquiring data files). Read in and plot the data:

```
dat = read.csv("../Data/growth.csv")
plot(length ~ age, data = dat, pch = 16, col = "grey")
```



Due to a large amount of variability in individual growth rates, the relationship looks pretty noisy. Notice how you have mostly young fish in your sample: this is characteristic of "random" sampling of fish populations.

Suppose you would like to obtain the probability that an average-sized fish of each age is sexually mature. You know that fish of this species mature at approximately 450 mm, and you simply need to determine the fraction of all fish at each age are greater than 450 mm. However, you don't have any observations for some ages (e.g., age 8), so you cannot simply calculate this fraction based on your raw data. You need to fit the von Bertalanffy growth model, then carry the statistical uncertainty from the fitted model forward to the predicted length-at-age. This would be difficult to obtain using only the coefficient estimates and their standard errors, because of the non-linear relationship between the x and y variables.

Enter the **bootstrap**, which is a Monte Carlo analysis using an observed data set and a model. The **pseudocode** for a bootstrap analysis is:

1. Resample from the originial data (with replacement)
2. Fit a model of interest
3. Derive some quantity of interest from the fitted model
4. Repeat steps 1 - 3 many times
5. Summarize the randomized quantities from step 4

In this example, you will apply a bootstrap approach to obtain the distribution of expected fish lengths at each age, then use these distributions to quantify the probability that an averaged-sized fish of each age is mature (i.e., greater than 450 mm).

You will write a function for each of steps 1 - 3 above. The first is to resample the data:

```
randomize = function(dat) {
  # number of observed pairs
  n = nrow(dat)
  # sample the rows to determine which will be kept
  keep = sample(x = 1:n, size = n, replace = T)
  # retreive these rows from the data
  dat[keep,]
}
```

Notice the use of `replace = T` here: without this, there would be no bootstrap. You would just sample the same observations over and over, their order in the rows would just be shuffled. Next, write a function to fit the model (revisit Section 3.4 for more details on `nls`):

```
fit_vonB = function(dat) {
  nls(length ~ linf * (1 - exp(-k * (age - t0))),
      data = dat,
      start = c(linf = 600, k = 0.3, t0 = -0.2)
      )
}
```

This function will return a fitted model object when executed. Next, write a function to predict mean length-at-age:

```
# create a vector of ages
ages = min(dat$age):max(dat$age)
pred_vonB = function(fit) {
  # extract the coefficients
  ests = coef(fit)
  # predict length-at-age
  ests["linf"] * (1 - exp(-ests["k"] * (ages - ests["t0"])))
}
```

Notice your function will use the object `ages` even though it was not defined in the function. This has to do with **lexical scoping** and **environments**, which are beyond the scope of this introductional material[3].

[3] If you'd like more details, see Hadley Wickham's page on it: http://adv-r.had.co.nz/Functions.html#lexical-scoping

Basically, if an object with the same name as one defined in the function exists outside of the funciton, the function will use the one that is defined in it. If there is no object defined in the function with that name, it will look outside of the function for that object.

Now, use these three functions to perform one iteration:

```
pred_vonB(fit = fit_vonB(dat = randomize(dat = dat)))
```

You can wrap this inside of a `replicate` call to perform step 4 above:

```
set.seed(2)
out = replicate(n = 100, expr = {
  pred_vonB(fit = fit_vonB(dat = randomize(dat = dat)))
})

dim(out)
```

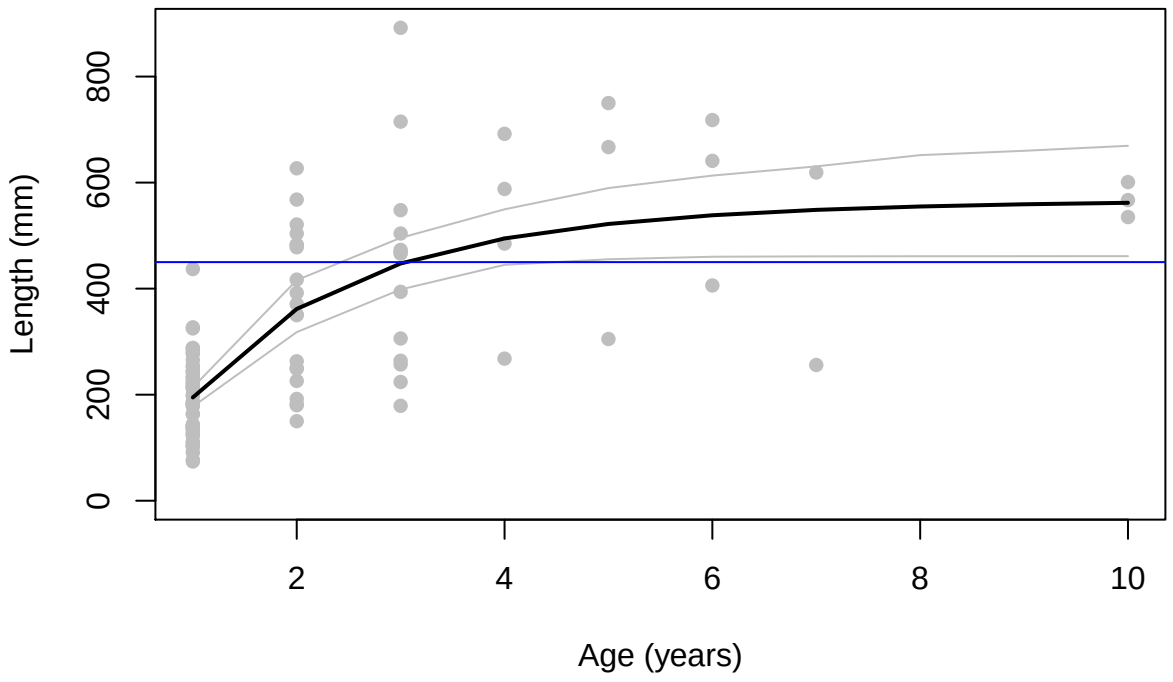
```
## [1]  10 100
```

It appears the rows are different ages and the columns are different bootstrapped iterations. Summarize the random lengths at each age:

```
summ = apply(out, 1, function(x) c(mean = mean(x), quantile(x, c(0.025, 0.975))))
```

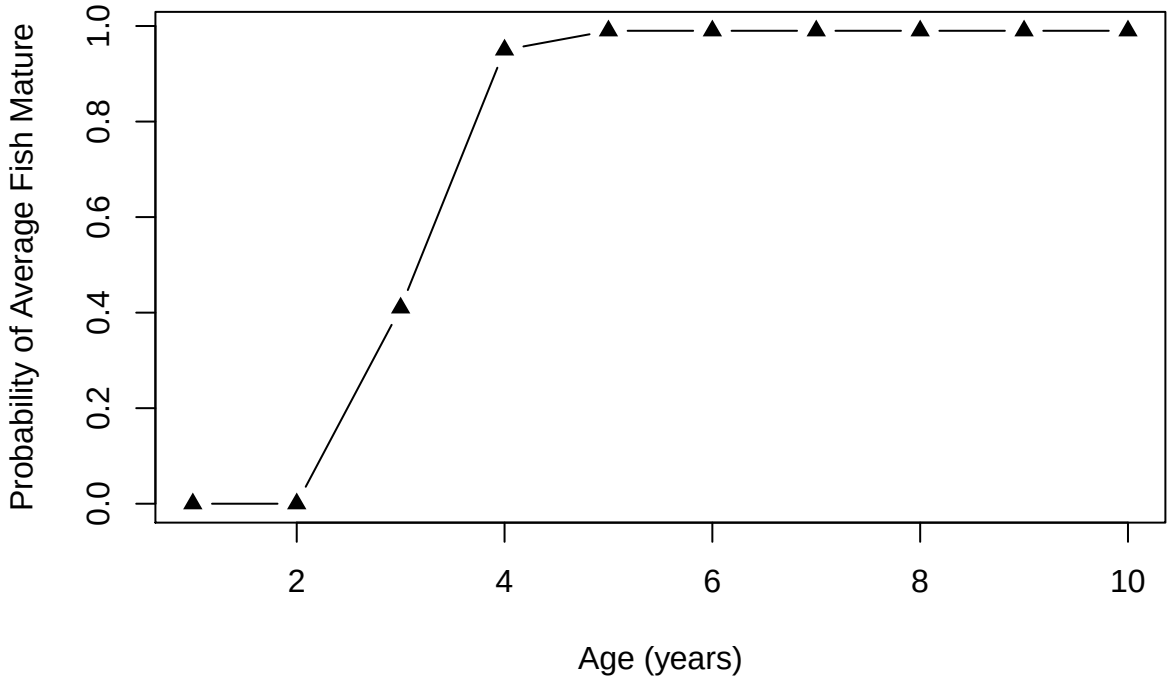
Plot the data, the summarized ranges of mean lengths, and the length at which all fish are assumed to be mature (450 mm)

```
plot(length ~ age, data = dat, col = "grey", pch = 16,
     ylim = c(0, max(dat$length, summ["97.5%",])),
     ylab = "Length (mm)", xlab = "Age (years)")
lines(summ["mean",] ~ ages, lwd = 2)
lines(summ["2.5%",] ~ ages, col = "grey")
lines(summ["97.5%",] ~ ages, col = "grey")
abline(h = 450, col = "blue")
```



Obtain the fraction of iterations that resulted in the mean length-at-age being greater than 450 mm. This is interpretted as the probability that the average-sized fish of each age is mature:

```
p_mat = apply(out, 1, function(x) mean(x > 450))
plot(p_mat ~ ages, type = "b", pch = 17,
     xlab = "Age (years)", ylab = "Probability of Average Fish Mature")
```



This **maturity schedule** can be used by fishery managers in attempting to decide which ages should be allowed to be harvested and which should be allowed to grow more[4]. Because each age has an associated expected length, managers can use what they know about the size selectivity of various gear types to set policies that attempt to target some ages more than others.

4.7.2 Permutation Test

In the previous example (Section 4.7.1), you learned about the bootstrap. A related Monte Carlo analysis is the **permutation test**. This is a non-parametric statistical test to determine if there is a statistically-significant difference in the mean of some quantity between two populations. It is used in cases where the assumptions of a generalized linear model may not be met, but a p-value is still required.

The **pseudocode** for the permutation test is:

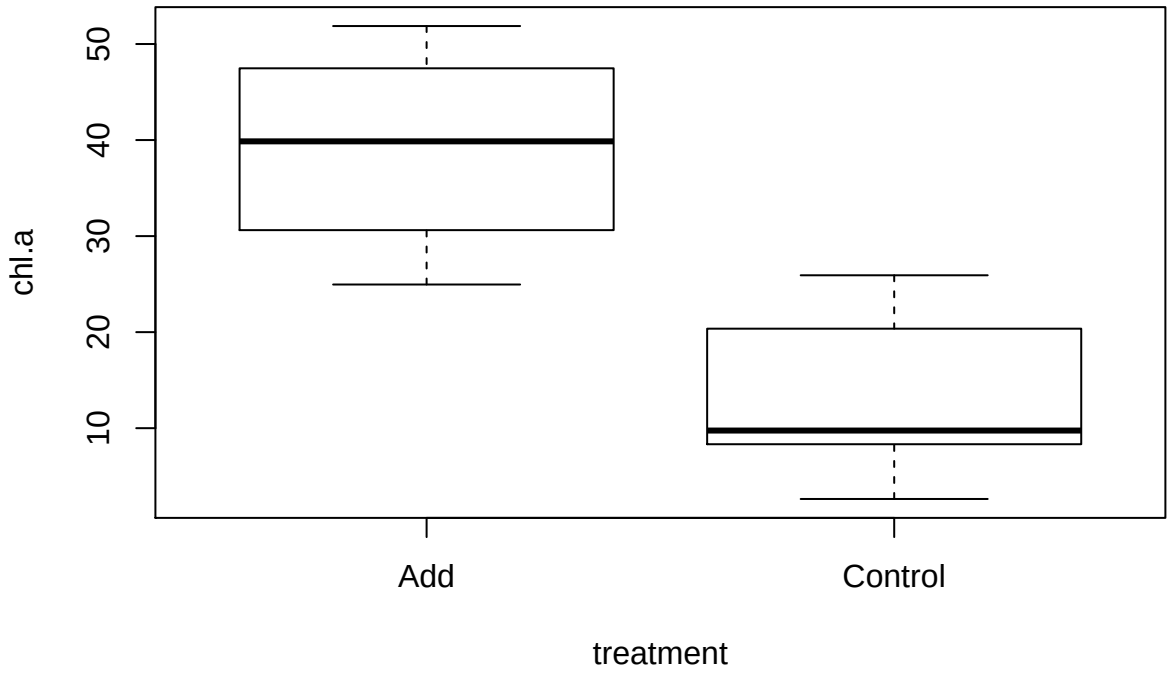
1. Calculate the difference between means based on the original data set
2. Shuffle the group assignments randomly among the observations
3. Calculate the difference between the randomly-assigned groups
4. Repeat steps 2 - 3 many times. This builds the **null distribution**: the distribution of the test statistic (the difference) assuming the null hypothesis that there is not difference in means is true
5. Determine what fraction of the absolute differences were larger than the original difference. This constitutes a **two-tailed** p-value. One-tailed tests can also be derived using the same steps 1 - 4, which is left as an exercise.

Use the data set `ponds.csv` for this example (see the instructions on aquiring data files). This is the same data set used for Exercise 1B, revisit that exercise for details on this hypothetical data set. Read in and plot

[4] possibly in a **yield-per-recruit** analysis

the data:

```
dat = read.csv("ponds.csv")
plot(chl.a ~ treatment, data = dat)
```



It appears as though there is a relatively strong signal indicating a difference. Use the permutation test to determine if it is statistically significant. Step 1 from the pseudocode is to caluclate the observed difference between groups:

```
Dobs = mean(dat$chl.a[dat$treatment == "Add"]) - mean(dat$chl.a[dat$treatment == "Control"])
Dobs
```

```
## [1] 26.166
```

Write a function to perform one iteration of steps 2 - 3 from the pseudocode:

```
# x is the group: Add or Control
# y is chl.a
perm = function(x, y) {
  # turn x to a character, easier to deal with
  x = as.character(x)
  # shuffle the x values:
  x_shuff = sample(x)
  # calculate the mean of each group:
  x_bar_add = mean(y[x_shuff == "Add"])
  x_bar_ctl = mean(y[x_shuff == "Control"])
  # calculate the difference:
  x_bar_add - x_bar_ctl
}
```

Use your function once:

```
perm(x = dat$treatment, y = dat$chl.a)
```

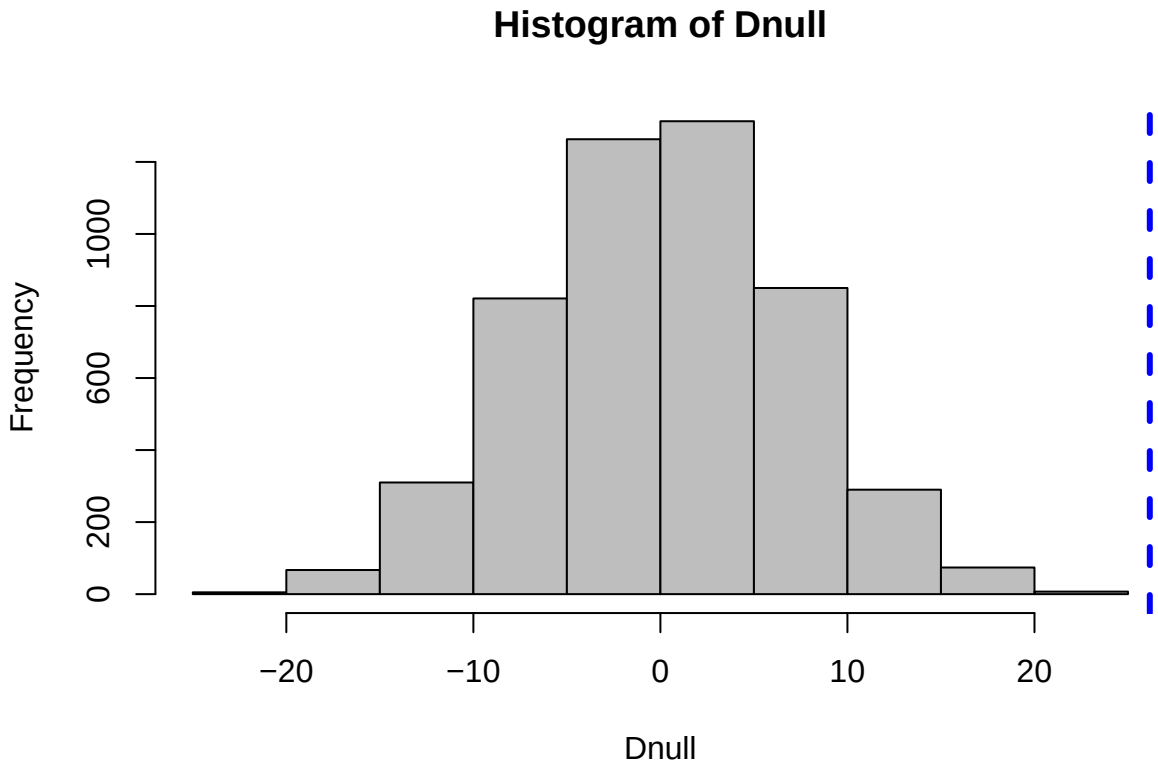
```
## [1] -11.15
```

Perform step 4 from the pseudocode by replicating the `perm` function many times:

```
Dnull = replicate(n = 5000, expr = perm(x = dat$treatment, y = dat$chl.a))
```

Plot the distribution of the null test statistic and draw a line where the originally-observed difference falls:

```
hist(Dnull, col = "grey")
abline(v = Dobs, col = "blue", lwd = 3, lty = 2)
```



Notice the null distribution is centered on zero: this is because the null hypothesis is that there is no difference. The observation (blue line) falls way in the upper tail of the null distribution, indicating it is unlikely an effect that large was observed by random chance. The two-tailed p-value can be calculated as:

```
mean(abs(Dnull) >= Dobs)
```

```
## [1] 0
```

Very few (or zero) of the random data sets resulted in a difference greater than what was observed, indicating there is statisitcal support to the hypothesis that there is a non-zero difference between the two nutrient treatments.

4.8 Exercise 4

In these exercises, you will be adapting the code written in this chapter to investigate slightly different questions. You should create a new R script `Ex4.R` in your working directory for these exercises so your chapter code is left unchanged. Exercise 4A is based solely on the required material and Exercises 4B - 4F are based on the example cases. You should work through each example before attempting each of the later exercises.

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are truly stumped.*

Exercise 4A: Required Material Only

These questions are based on the material in Sections 4.1 - 4.5 only.

1. Simulate flipping an unfair coin (probability of heads = 0.6) 100 times using `rbinom`. Count the number of heads and tails.
2. Simulate flipping the same unfair coin 100 times, but suing `sample` instead. Determine what fraction of the flips resulted in heads.
3. Simulate rolling a fair 6-sided die 100 times using `sample`. Determine what fraction of the rolls resulted in an even number.
4. Simulate rolling the same die 100 times, but use the function `rmultinom` instead. Look at the help file for details on how to use this function. Determin what fraction of the rolls resulted in an odd number.

Solutions

Exercise 4B: Test `rnorm`

These questions will require you to adapt the code written in Section 4.6.1

1. Adapt this example to investigate another univariate probability distribution, like `-lnorm`, `-pois`, or `-beta`. See the help files (e.g., `?rpois`) for details on how to use each function.

Solutions

Exercise 4C: Stochastic Power Analysis

These questions will require you to adapt the code written in Section 4.6.2

1. What sample size `n` do you need to have a power of 0.8 of detecting a significant difference between the two tagging methods?
2. How do the inferences from the power analysis change if you are interested in `p_new = 0.4` instead of `p_new = 0.25`? Do you need to tag more or fewer fish in this case?
3. Your analysis takes a bit of time to run so you are interested in tracking its progress. Add a progress message to your nested `for` loop that will print the sample size currently being analyzed:

```
for (n in 1:N) {
  cat("\r", "Sample Size = ", n_try[n])
  for (i in 1:I) {
    ...
  }
}
```

Solutions

Exercise 4D: Harvest Policy Analysis

These questions will require you to adapt the code written in Section 4.6.3

1. Add an argument to `ricker_sim` that will give the user an option to create a plot that shows the time series of recruitment, harvest, and escapement all on the same plot. Set the default to be to not plot the result, in case you forget to turn it off before performing the Monte Carlo analysis.
2. Add a _error handler** to `ricker_sim` that will cause the function to return an error `if` the names of the vector passed to the `param` argument aren't what the function is expecting. You can use stop("Error Message Goes Here") to have your function stop and return an error.
3. How do the results of the trade-off analysis differ if the process error was larger (a larger value of σ)?
4. Add implementation error to the harvest policy. That is, if the target exploitation rate is U, make the real exploitation rate in year y be: $U_y \sim Beta(a, b)$, where $a = 100U$ and $b = 100(1 - U)$. You can make there be more implementation error by inserting a smaller number other than 100 here. How does this affect the trade-off analysis?

Solutions

Exercise 4E: The Bootstrap

These questions will require you to adapt the code written in Section 4.7.1

1. Replicate the bootstrap analysis but adapted for the linear regression example in Section 3.1.1. Stop at the step where you summarize the 95% interval range.
2. Compare the 95% bootstrap confidence intervals to the intervals you get by running the `predict` function on the original data set with the argument `interval = "confidence"` set.

Solutions

Exercise 4F: Permutation Tests

These questions will require you to adapt the code written in Section 4.7.2

1. Adapt the code to perform a permutation test for the difference in each of the zooplankton densities between treatments. Don't forget to fix the missing value in the `chao` variable. See Exercise 2 for more details on this.
2. Adapt the code to perform a permutation test for another data set used in this book where there are observations of both a categorical variable and a continuous variable. The the data sets `sockeye.csv`, `growth.csv`, or `creel.csv` should be good starting points.
3. Add a calculation of the p-value for a one-tailed test (i.e., that the difference in means is greater or less than zero). Steps 1 - 4 are the same: all you need is `Dnull` and `Dobs`. Don't be afraid to Google this if you are confused.

Solutions

Chapter 5

Large Data Manipulation

Chapter Overview

Any data analyst will tell you that oftentimes the most difficult part of an analysis is simply getting the data in the proper format. Real data sets are messy: they have missing values, variables are often stored in multiple columns that would be better stored as rows, the same factor level may be coded as two or more different types[1], and the required data are in several separate files. You get the point: sometimes significant data-wrangling may be required before you perform an analysis.

In this chapter, you will learn tricks to do more advanced data manipulation in R using two R **packages**:

- `{reshape2}`: used for changing data between formats (wide and long)
- `{dplyr}`: used for large data manipulations with a consistent and readable format.

You will be turning daily observations of harvest and escapement (fish that are not harvested) into annual totals for the purpose of fitting a **spawner-recruit analysis** on a hypothetical pink salmon (*Oncorhynchus gorbuscha*) data set. More details and context on these terms and topics will be provided later.

IMPORTANT NOTE: If you did not attend the sessions corresponding to Chapters 1 or 2, you are recommended to walk through the material found in those chapters before proceeding to this material. Additionally, you will find the material in Section 3.4 helpful for the end of this chapter. Remember that if you are confused about a topic, you can use **CTRL + F** to find previous cases where that topic has been discussed in this book.

Before You Begin

You should create a new directory and R script for your work in this Chapter. Create a new R script called `Ch5.R` and save it in the directory `C:/Users/YOU/Documents/R-Book/Chapter5`. Set your working directory to that location. Revisit the material in Sections 1.2 and 1.3 for more details on these steps.

5.1 Aquiring and Loading Packages

An **R package** is a bunch of code, documentation, and data that someone has written and bundled into a consistent format that allows other R users to install and use in their R sessions. R packages make R

[1] `"hatch"`, "`Hatch`", and `"HATCH"` are all treated differently in R

incredibly flexible and extensible. If you are trying to do a specialized analysis that is not included in the base R distribution, it is likely that someone has already written a package that will allow you to do it!

You will be using some new packages that you likely don't already have on your computer, so you will need to install them first:

```
install.packages("dplyr")
install.packages("reshape2")
```

This requires an internet connection. You will see some text display in the console telling you the packages are being installed. This only needs to be done once for each version of R you have. If you install a new version of R, you will likely need to update your packages (i.e., re-install them).

Now that the packages are on your computer, you will need to load them into the current session:

```
library(dplyr)
library(reshape2)
```

These messages are telling you that there are functions in the `{dplyr}` package that have the same names as those already being used in the `{stats}` and `{base}` R packages. This is fine so long as you don't want to use the original functions. If you wish to use the `{base}` version of the `filter()` function rather than the `{dplyr}` version, use it like this: `base::filter()`. Each time you close and reopen R, you will need to load any packages you want to use using `library()` again.

5.2 The Data

You have daily observations of catch and escapement for every other year between 1917 and 2015. Pink salmon have a simple life history where fish spawned in one odd year return as adults to spawn in the next odd year. There is a commercial fishery in this system located at the river mouth which harvests fish as they enter the river. A counting tower is located upstream of the fishing grounds that counts the number of fish that escaped the fishery and will have a chance to spawn (this number is termed "escapement"). The ultimate goal is to obtain annual totals of spawners and total run (catch + escapement) for the purpose of fitting a spawner recruit analysis. You will perform some other analyses along the way directed quantifying patterns in run timing.

Read in the two data sets for this chapter (see the instructions for details on acquiring data files):

```
catch = read.csv("../Data/daily_catch.csv")
esc = read.csv("../Data/daily_escape.csv")
```

Look at the first 6 rows and columns of the `catch` data:

```
catch[1:6,1:6]
```

```
##   doy y_1917 y_1919 y_1921 y_1923 y_1925
## 1 160    175    229    245    135   2685
## 2 161    221    501   1379   1504   2361
## 3 162    242    355   1149     13    277
## 4 163     90    197     52   2721    548
## 5 164    134    428    674    747   1209
## 6 165     76    224   1211   1231    772
```

The column `doy` is the day of the year that record corresponds to, and each column represents a different year. Notice the format of the `esc` data is the same:

```
esc[1:6,1:6]
```

```
##   doy y_1917 y_1919 y_1921 y_1923 y_1925
```

```
## 1 161     90    189    161     73   1700
## 2 162    113    412    907    819   1495
## 3 163    124    292    756      7    176
## 4 164     46    162     34   1481    347
## 5 165     69    351    443    407    766
## 6 166     39    184    796    670    489
```

But notice the first `doy` is one day later for `esc` than for `catch`. This is because of the time lag for fish to make it from the fishery grounds to the counting tower (a one day swim: fish that were in the fishing grounds on day `d` passed the counting tower on day `d+1` if they were not harvested).

5.3 Change format using `melt`

These data are in what is called **wide** format: different levels of the `year` variable are stored as columns. A **long** format would have three columns: one for `doy`, `year`, and `catch` or `esc`. Turn the `catch` data frame into long format using the `melt` function from `{reshape2}`:

```
long.catch = melt(catch, id.var = "doy",
                  variable.name = "year",
                  value.name = "catch")
head(long.catch)
```

```
##   doy   year catch
## 1 160 y_1917   175
## 2 161 y_1917   221
## 3 162 y_1917   242
## 4 163 y_1917    90
## 5 164 y_1917   134
## 6 165 y_1917    76
```

The first argument is the data frame to reformat, the second argument (`id.var`) is the variable that identifies one observation from another, here it is the `doy` that the count occurred on. In this case, it is actually all we need to run `melt` properly. The optional `variable.name` and `value.name` arguments specify the names of the other two columns. These default to "variable" and "value" if not specified. Do the same thing for escapement:

```
long.esc = melt(esc, id.var = "doy",
                variable.name = "year",
                value.name = "esc")
head(long.esc)
```

```
##   doy   year esc
## 1 161 y_1917  90
## 2 162 y_1917 113
## 3 163 y_1917 124
## 4 164 y_1917  46
## 5 165 y_1917  69
## 6 166 y_1917  39
```

Anytime you do a large data manipulation like this, you should compare the new data set with the original to verify that it worked properly. Ask if `all` of the daily catches for a given year in the original data set match up to the new data set:

```
all(catch$y_2015 == long.catch[long.catch$year == "y_2015", "catch"])
```

```
## [1] TRUE
```

The single `TRUE` indicates that every element matches up for 2015 in the catch data, just like they should.

5.4 Join Data Sets with `merge`

You now have two long format data sets. You can turn them into one data set with the `merge` function (which is actually in the `{base}` package). `merge` takes two data frames and joins them based on certain grouping variables. Here, you want to combine the data frames `long.catch` and `long.esc` into one data frame, and match the rows by the `doy` and `year` for each unique pair:

```
dat = merge(x = long.esc, y = long.catch,
            by = c("doy", "year"), all = T)
head(dat)
```

```
##   doy   year esc catch
## 1 160 y_1917  NA   175
## 2 160 y_1919  NA   229
## 3 160 y_1921  NA   245
## 4 160 y_1923  NA   135
## 5 160 y_1925  NA  2685
## 6 160 y_1927  NA   163
```

The `all = T` argument is important to specify here. It says that you wish to keep **all** of the unique records in the columns passed to `by` from both data frames. The `doy` in the catch data ranges from day 160 to day 209, whereas in escapement it ranges from day 161 to day 210. In this system, the fishery opens on the 160^{th} day of every year, and the counting tower starts the 161^{st}. If you didn't use `all = T`, you would drop all the rows in each data set that do not have a match the other data set (you would lose day 160 and day 210 from every year). Notice that because no escapement observations were ever made one `doy` 160, the `esc` values on this day is `NA`.

Notice how the data set is now ordered by `doy` instead of `year`. This is not a problem, but if you want to reorder the data frame by `year`, you can use the `arrange` function from `{dplyr}`:

```
head(arrange(dat, year))
```

```
##   doy   year esc catch
## 1 160 y_1917  NA   175
## 2 161 y_1917  90   221
## 3 162 y_1917 113   242
## 4 163 y_1917 124    90
## 5 164 y_1917  46   134
## 6 165 y_1917  69    76
```

5.5 Lagged vectors

The present task is to the calculate daily cumulative run proportion[2] in 2015 for comparison to the other years. Given the information you have, this task would be very cumbersome if not for the flexibility allowed by `{dplyr}`.

Remember the counting tower is an average one day swim for the salmon from the fishing grounds. This means that the escapement counts are **lagged** relative to the catch counts. If you want the daily run (defined as the number in fishing grounds each day), you need to account for this lag. An easy way to think about the lag is to show it graphically. First, pull out one year of data using the `filter` function in `{dplyr}`:

[2]This is the fraction of all fish that came back each year that did so on or before a given day

```
y15 = filter(dat, year == "y_2015")
head(y15)
```

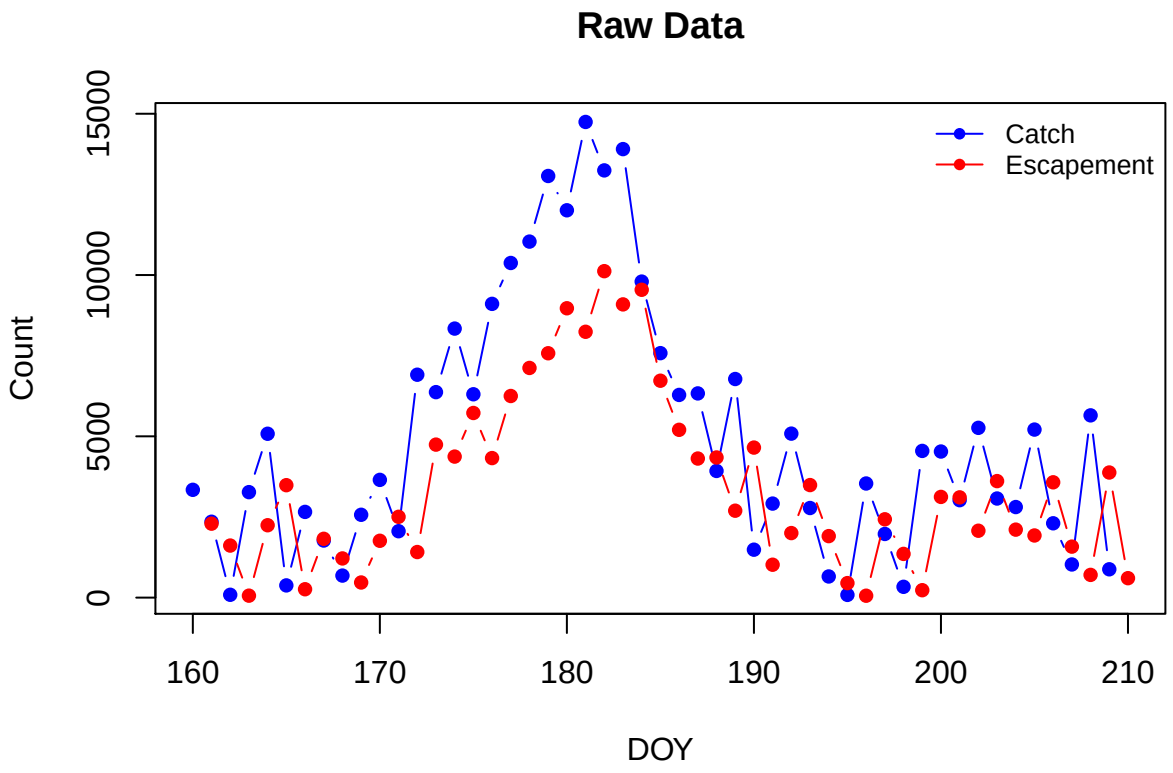
```
##   doy   year  esc catch
## 1 160 y_2015   NA  3342
## 2 161 y_2015 2293  2352
## 3 162 y_2015 1614    88
## 4 163 y_2015   61  3270
## 5 164 y_2015 2244  5080
## 6 165 y_2015 3486   379
```

The `{base}` analog to the `filter` function is:

```
y15 = dat[dat$year == "y_2015", ]
# or
y15 = subset(dat, year == "y_2015")
```

They take about the same amount of code to write, but `filter` has some advantages when used with other `{dplyr}` functions that will be discussed later. Now plot the daily catch and escapement (not cumulative yet) for 2015:

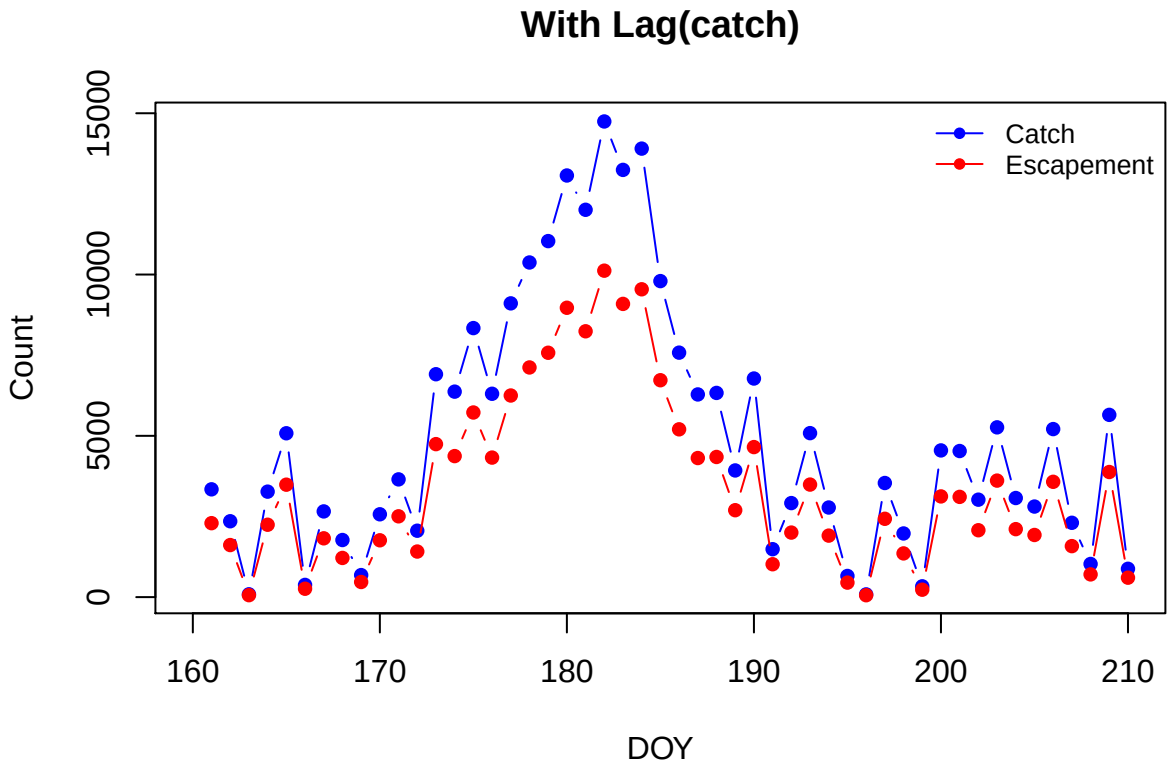
```
plot(catch ~ doy, data = y15, type = "b", pch = 16, col = "blue",
     xlab = "DOY", ylab = "Count", main = "Raw Data")
lines(esc ~ doy, data = y15, type = "b", pch = 16, col = "red")
legend("topright", legend = c("Catch", "Escapement"),
       col = c("blue", "red"), pch = 16, lty = 1, bty = "n", cex = 0.8)
```



Notice how the escapement counts are shifted to the right by one day. This is the lag. To account for it,

you can use the `lag` function from `{dplyr}`. `lag` works by lagging some vector by `n` elements (it shifts every element in the vector to the right by one place by putting `n` `NAs` at the front and removing the last `n` elements from the original vector). One way to fix the lag problem would be to use `lag` on `catch` to make the days match up with `esc` (don't lag `esc` because that would just lag it by `n` additional days):

```
plot(lag(catch, n = 1) ~ doy, data = y15, type = "b", pch = 16, col = "blue",
     xlab = "DOY", ylab = "Count", main = "With Lag(catch)")
lines(esc ~ doy, data = y15, type = "b", pch = 16, col = "red")
legend("topright", legend = c("Catch", "Escapement"),
       col = c("blue", "red"), pch = 16, lty = 1, bty = "n", cex = 0.8)
```



Notice how the two curves line up better now.

5.6 Adding columns with `mutate`

Apply this same idea to the data set to get the total number of fish that were in the fishing grounds on each day (the daily run, what was harvested plus what was not). First, make a function to take two vectors, lag one, then sum them: this will give you the daily run. However, you don't need to move catch forward a day, you need to move escapement back a day. The opposite of `lag` is `lead`. The way to think of this is that `lag` lags a vector, and `lead` unlags a vector. `lead` shifts every element to the left by `n` elements by removing the first `n` elements of the original vector and adding `n` `NAs` to the end. This is what you want for your function:

```
lag_sum = function(x, y, n = 1){
  # x is lagged from y by n days
  rowSums(cbind(lead(x, n), y), na.rm = T)
}
```

Note the use of `na.rm = T` to specify that summing a number with an `NA` should still return a number. If two `NA`s are summed, the result will be a zero.

Try out the `lag_sum` function:

```
lag_sum(y15$esc, y15$catch)
```

```
##  [1]  5635  3966   149  5514  8566   639  4480  2985  1152  4330  6155
## [12]  3472 11653 10742 14064 10630 15356 17494 18612 22040 20248 24865
## [23] 22334 23447 16521 12781 10595 10675  6622 11428  2506  4916  8573
## [34]  4685  1106   144  5966  3329   565  7670  7637  5095  8875  5180
## [45]  4735  8784  3885  1733  9528  1481     0
```

You will want to add this column to your data set. You add columns in `{dplyr}` using `mutate`:

```
y15 = mutate(y15, run = lag_sum(esc, catch))
```

The `{base}` equivalent to `mutate` is:

```
y15$run = lag_sum(y15$esc, y15$catch)
```

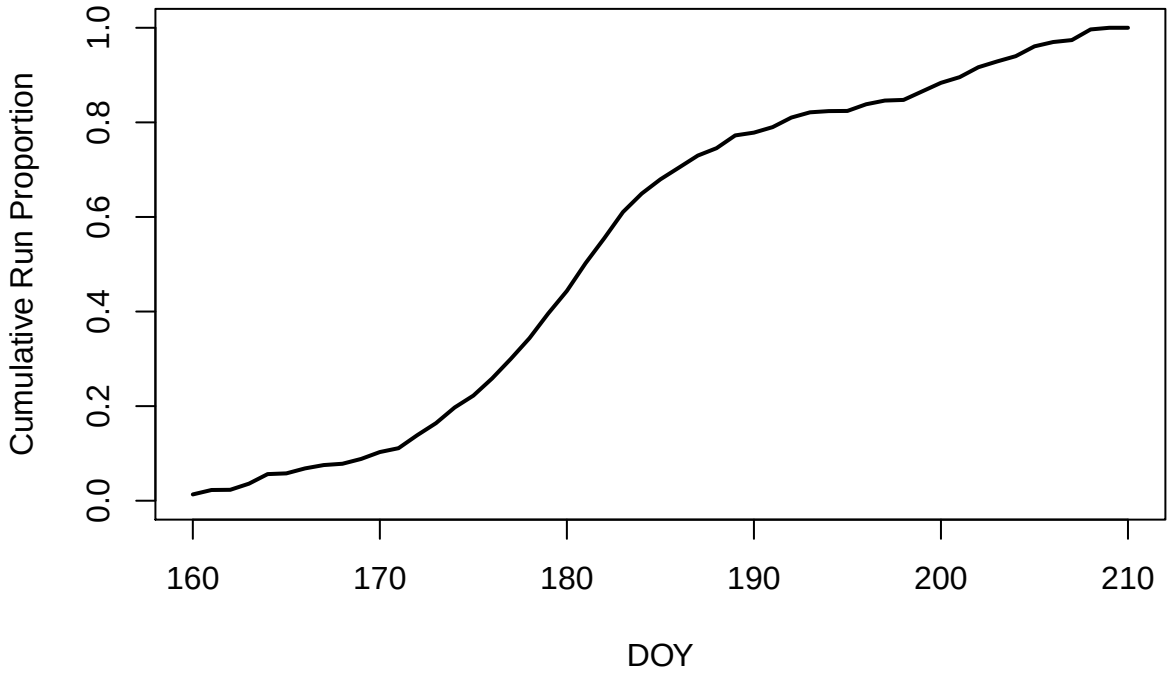
`mutate` has advantages when used with other `{dplyr}` functions that will be shown soon. Use the new variable `run` to calculate two new variables: the cumulative run by day (`crun`) and cumulative run proportion by day (`cprun`). Cumulative means that each new element is the value that occurred on that day plus all of the values that happened on days before it. You can use R's `cumsum` for this:

```
y15 = mutate(y15, crun = cumsum(run), cprun = crun/sum(run, na.rm = T))
```

Here is one advantage of `mutate`: you can refer to a column you just created to make a new column within the same function. You would have to split this into two lines to do it in `{base}`. Indeed, you could have made `run`, `crun`, and `cprun` all in the same `mutate` call. Plot `cprun`[3]:

```
plot(cprun ~ doy, data = y15, type = "l", ylim = c(0, 1),
     xlab = "DOY", ylab = "Cumulative Run Proportion", lwd = 2)
```

[3]It doesn't look like much, but this information is incredibly important for fishery managers. It provides information on how much of the annual run has passed on each day. You can see that the rate of change is the fastest in the middle of the run, this corresponds to the major hump in the plots you made earlier (which were raw numbers by day). Of course during the run, the managers don't know what proportion of the total run has passed because they don't know the total number of fish that will come that season, but by characterizing the mean and variability of the different run completion proportions on each day in the past, they can have some idea of how much of the run to expect yet to come on any given day.



5.7 Apply to all years

You have just obtained the daily and cumulative run for one year (2015), but `{dplyr}` makes it easy to apply these same manipulations to all years. To really see the advantage, you'll need to learn to use piping. A code **pipe** is one that takes the result of one function and inserts it into the input of another function. Here is a basic example of a pipe:

```
rnorm(10) %>% length
```

```
## [1] 10
```

The pipe took the result of `rnorm(10)` and passed it as the first argument to the `length` function. This allows you to string together commands so you aren't continuously making intermediate objects or nesting a ton of functions together.

The reason piping works so well with `{dplyr}` is because its functions are designed to take a data frame as input as the first argument and return another data frame as output, which allows you to string them together with pipes. Do a more complex pipe: take your main data frame (`dat`), group it by `year`, and pass it to a `mutate` call that will add the new three columns to the entire data set:

```
dat = dat %>%
  group_by(year) %>%
  mutate(run = lag_sum(esc, catch),
         crun = cumsum(run),
         cprun = crun/sum(run, na.rm = T))
```

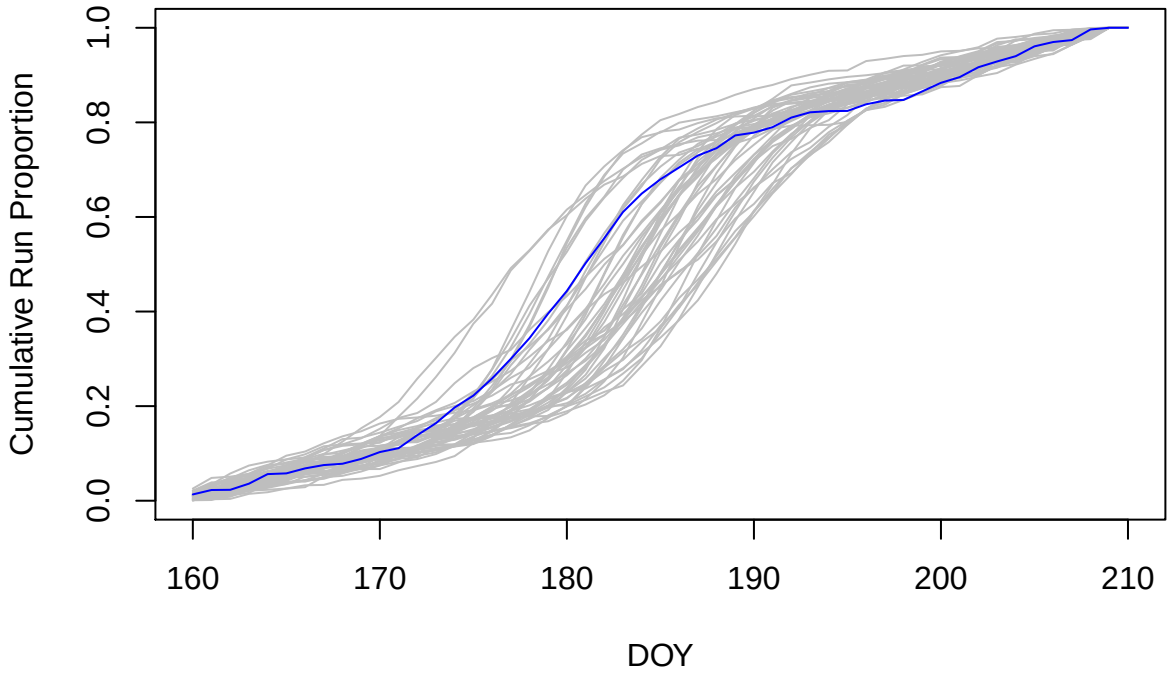
```
arrange(dat, year)
```

```
## # A tibble: 2,550 x 7
## # Groups:   year [50]
##      doy year     esc catch   run  crun  cprun
##    <int> <fct>  <int> <int> <dbl> <dbl>  <dbl>
##  1   160 y_1917    NA   175  265.  265. 0.0127
##  2   161 y_1917    90   221  334.  599. 0.0288
##  3   162 y_1917   113   242  366.  965. 0.0464
##  4   163 y_1917   124    90  136. 1101. 0.0530
##  5   164 y_1917    46   134  203. 1304. 0.0627
##  6   165 y_1917    69    76  115. 1419. 0.0683
##  7   166 y_1917    39    89  134. 1553. 0.0747
##  8   167 y_1917    45   139  210. 1763. 0.0848
##  9   168 y_1917    71   127  192. 1955. 0.0941
## 10   169 y_1917    65    61   92. 2047. 0.0985
## # ... with 2,540 more rows
```

The `group_by` function is spectacular: you grouped the data frame by `year`, which tells the next function to apply its commands to each year independently. With this function, you can group your data in any way you please and calculate statistics on a group-by-group basis. Note that in this example, it doesn't do much for you, because the data are so neat (all years have same number of days, `NAs` in all of the same places, etc.), but for data that are messier, this trick is very handy. Also note that the `dat` object looks a little different. When you print it, it only shows the first 10 rows. This is a `{dplyr}` feature that prevents you from printing a huge data set to the console.

Now plot the cumulative run proportion by day for each year, highlighting 2015:

```
# make an empty plot
plot(x = 0, y = 0, type = "n", xlim = range(dat$doy), ylim = c(0,1),
     xlab = "DOY", ylab = "Cumulative Run Proportion")

years = levels(dat$year)

# use sapply to "loop" through years, drawing lines for each
tmp = sapply(years, function(x) {
  lines(cprun ~ doy, data = filter(dat, year == x),
        col = ifelse(x == "y_2015", "blue", "grey"))
})
```



5.8 Calculate Daily Means with `summarize`

Oftentimes, you want to calculate a statistic for a value across grouping variables, like year or site or day. Remember the `tapply` function does this in `{base}`. Here you want to calculate the mean cumulative run proportion for each day averaged across years. For this, you can make use of the `summarize` function in `{dplyr}`. Begin by ungrouping the data to remove the `year` groups and grouping the data by `doy`. This will allow you to calculate the mean proportion by day. You then pass this grouped data frame to `summarize` which will apply the `mean` function to the cumulative run proportions across all years on a particular day. It will do this for all days:

```
mean_cprun = dat %>%
  ungroup %>%
  group_by(doy) %>%
  summarize(cprun = mean(cprun))
head(mean_cprun)
```

```
## # A tibble: 6 x 2
##     doy   cprun
##   <int>   <dbl>
## 1   160 0.00970
## 2   161 0.0194
## 3   162 0.0291
## 4   163 0.0387
## 5   164 0.0491
## 6   165 0.0579
```

The way to do this in {base} is with tapply:

```
tapply(dat$cprun, dat$doy, mean)[1:6]
```

```
##         160         161         162         163         164         165
## 0.009699779 0.019400196 0.029111609 0.038735562 0.049130566 0.057901058
```

Note that you get the same result, only the output from tapply is a vector, and the output of summarize is a data frame.

One disadvantage of the summarize function, however, is that it can only be used to apply functions that give one number as output (e.g., mean, max, sd, length, etc.). These are called aggregate functions: they take a bunch of numbers and aggregate them into one number. tapply does not have this constraint. Illustrate this by trying to use the range function (which combines the minimum and maximum values of some vector into another vector of length 2).

```
# with summarise
dat %>%
  ungroup %>%
  group_by(doy) %>%
  summarize(range = range(cprun))
```

```
## Error in summarise_impl(.data, dots): Column `range` must be length 1 (a summary value), not 2
```

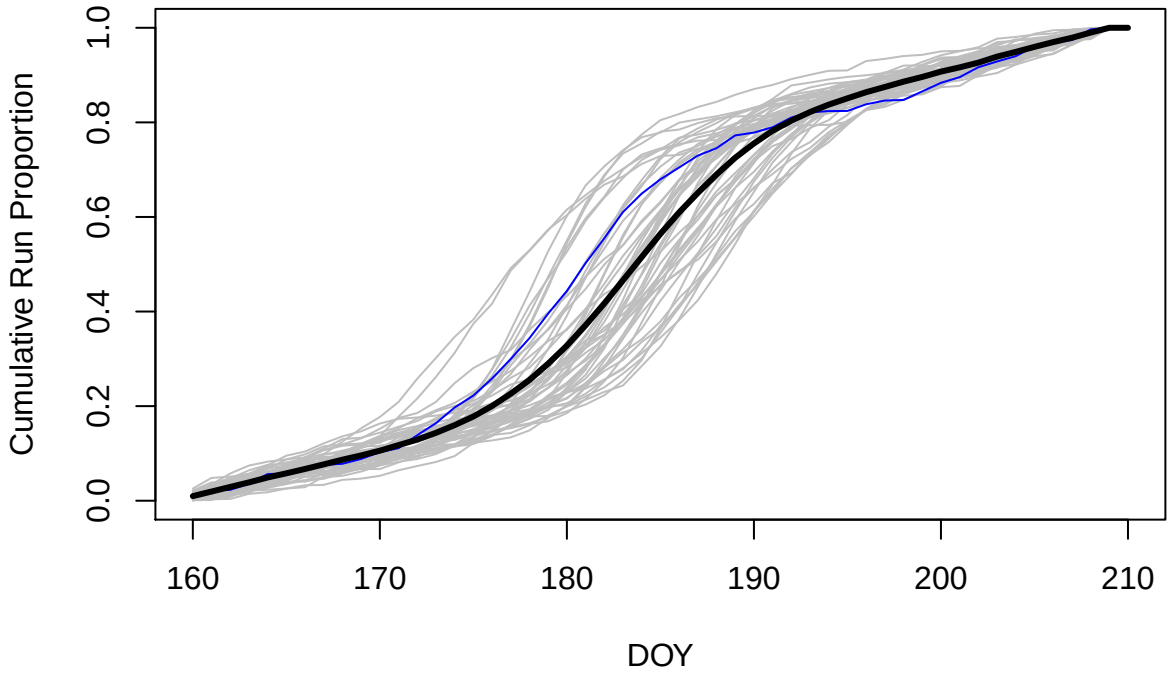
```
# with tapply
tapply(dat$cprun, dat$doy, range)[1:5]
```

```
## $`160`
## [1] 0.0002721184 0.0256423751
##
## $`161`
## [1] 0.001485884 0.048191292
##
## $`162`
## [1] 0.003895501 0.057528072
##
## $`163`
## [1] 0.01450474 0.07435435
##
## $`164`
## [1] 0.01797648 0.08253284
```

You can see that tapply allows you to do this (but gives a list), whereas summarize returns a short and informative error. However, you could very easily fix the problem by making minimum and maximum columns in the same summarize call:

```
dat %>%
  ungroup %>%
  group_by(jday) %>%
  summarize(min = min(cprun), max = max(cprun))
```

Add the mean cumulative run proportion to our plot (use lines just like before, add it after the sapply call).



It looks like 2015 started like an average year but the peak of the run happened a couple days earlier than average.

5.9 More `summarize`

Your colleagues have been talking lately about how the run has been getting earlier and earlier in recent years. To determine if their claims are correct, you decide to develop a method to find the day on which 50% of the run has passed (the median run date) and plot it over time. If there is a downward trend, then the run has been getting earlier. First, you need to define another function to find the day that corresponds to 50% of the run. If there were days where the cumulative run on that day equaled exactly 0.5, we could simply use `which`:

```
ind = which(dat$cprun == 0.5)
dat$jday[ind]
```

The `which` function is useful: it returns the indices (element places) *for which* the condition is `TRUE` However you need to do a workaround here, since the median day is likely not exactly 0.5 (and it has to be for `==` to return a `TRUE`). We will give the function two vectors and it will look for a value that is closest to 0.5 in one vector and pull out the corresponding value in the second vector:

```
find_median_doy = function(p,doy) {
  ind = which.min(abs(p - 0.5))
  doy[ind]
}
```

This function will take every element of `p`, subtract 0.5, take the absolute value of the results (make it

positive, even if it is negative), and find the element number that is smallest. This will be the element with the value closest to 0.5. Note that you could find the first quartile (0.25) or third quartile (0.75) of the run by inserting these in for 0.5 above[4]. Try your function on the 2015 data only:

```
# use function
med_doy15 = find_median_doy(p = y15$cprun, doy = y15$doy)

# pull out the day it called the median to verify it works
filter(y15, doy == med_doy15) %>% select(cprun)
```

```
##       cprun
## 1 0.5023032
```

The `select` function in `{dplyr}` extracts the variables requested from the data frame. If there is a grouping variable set using `group_by`, it will be returned as well. It looks like the function works. Now combine it with `summarize` to calculate the median day for every year:

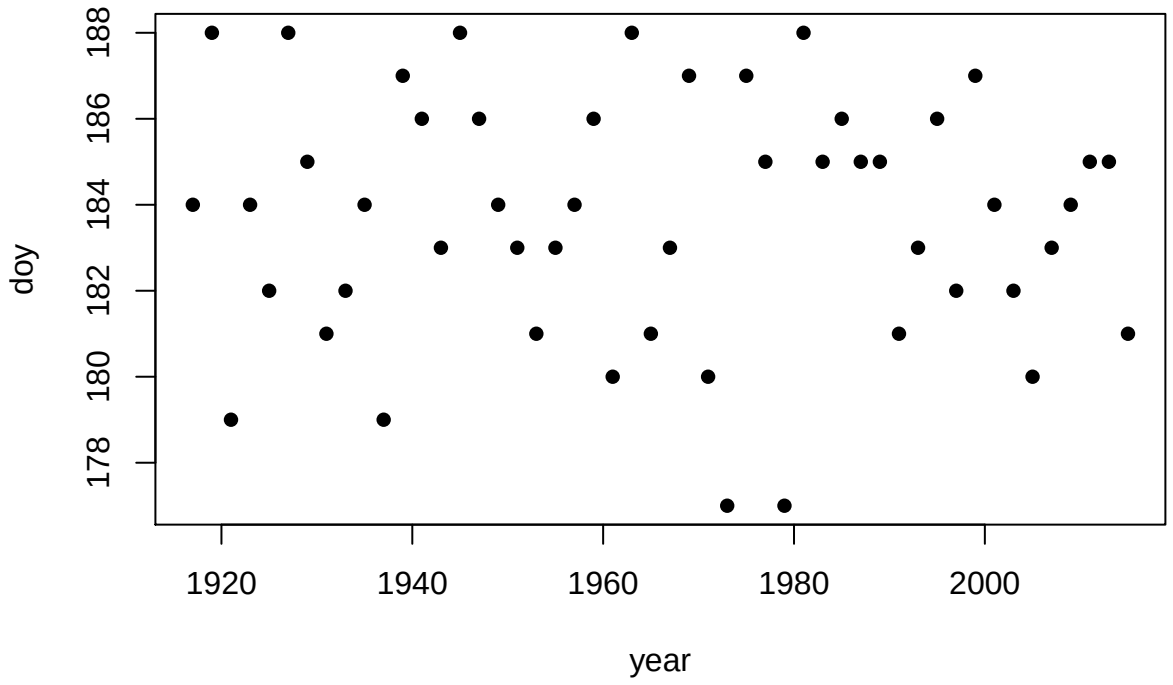
```
med_doy =
  dat %>%
  group_by(year) %>%
  summarize(doy = find_median_doy(cprun, doy))
head(med_doy)
```

```
## # A tibble: 6 x 2
##   year     doy
##   <fct>  <int>
## 1 y_1917   184
## 2 y_1919   188
## 3 y_1921   179
## 4 y_1923   184
## 5 y_1925   182
## 6 y_1927   188
```

Now, use your `{dplyr}` skills to turn the year column into a numeric variable:

```
# get years as a number
med_doy$year =
  ungroup(med_doy) %>%
  # extract only the year column
  select(year) %>%
  # turn it to a vector and extract unique values
  unlist %>% unname %>% unique %>%
  # replace "y_" with nothing
  gsub(pattern = "y_", replacement = "") %>%
  # turn to a integer vector
  as.integer

plot(doy ~ year, data = med_doy, pch = 16)
```

[4]This is the perfect kind of thing to make an argument to your function!

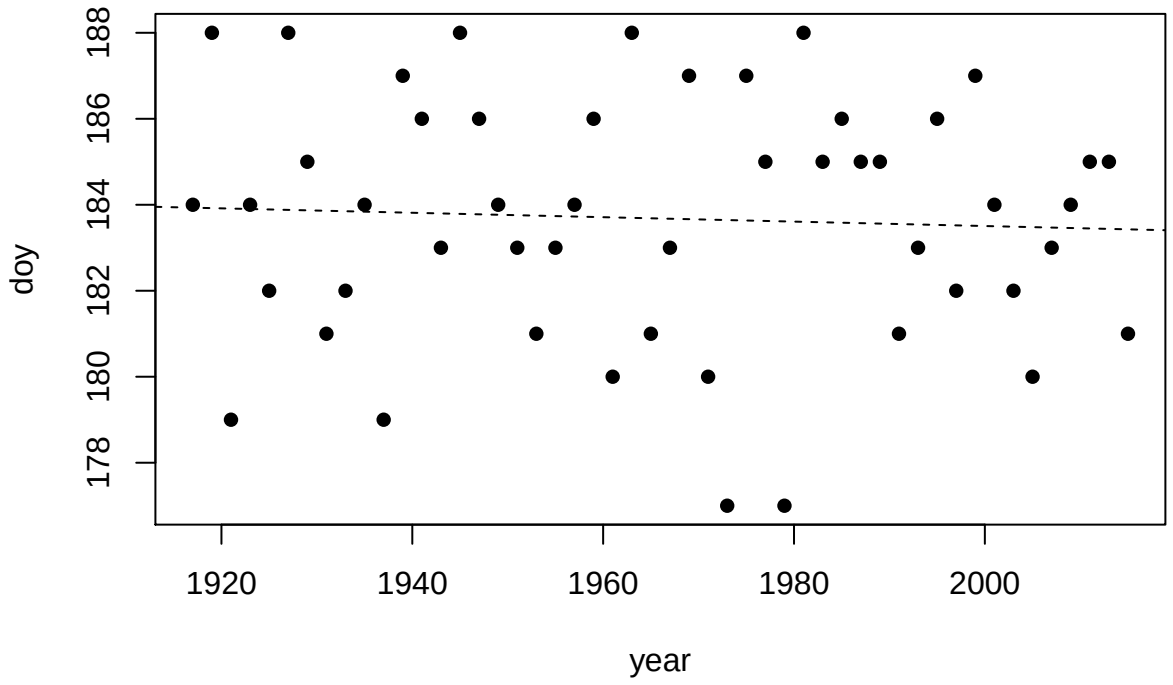


It doesn't appear that there is a trend in either direction, but fit a regression line because you can (Section 3.1.1):

```
fit = lm(doy ~ year, data = med_doy)
summary(fit)$coef
```

```
##                  Estimate  Std. Error    t value     Pr(>|t|)
## (Intercept) 193.781416567 27.80436398  6.9694605 8.190067e-09
## year         -0.005138055  0.01414108 -0.3633424 7.179445e-01
```

```
plot(doy ~ year, data = med_doy, pch = 16)
abline(fit, lty = 2)
```



The results tell you that there is an 0.005 day decrease in median run date for every 1 year that has gone by and that it is not significantly different from a zero day decrease per year.

5.10 Fit the Model

You have done some neat data exploration exercises (and hopefully learned `{dplyr}` and piping along the way!), but you ultimate task with this data set is to run a spawner-recruit analysis on all the data. A **spawner-recruit analysis** is one that links the number of total fish produced (the recruits) by a given number of spawners (the escapement). This is done in an attempt to describe the productivity and carrying capacity of the population and to obtain **biological reference points** off of which harvest policies can be set. You will need to summarize the daily data into annual totals of escapement and total run:

```
sr_dat =
  dat %>%
  summarize(S = sum(esc, na.rm = T),
            R = sum(run, na.rm = T))

head(sr_dat)
```

```
## # A tibble: 6 x 3
##   year       S       R
##   <fct>  <int>   <dbl>
## 1 y_1917  7031  20785.
## 2 y_1919 19647  43553.
## 3 y_1921 45106 113682.
```

```
## 4 y_1923 69160 196212.
## 5 y_1925 66306 171006.
## 6 y_1927 96622 243212.
```

You didn't need to use `group_by` here because the data were still grouped from an earlier task. There is no harm in putting it in, and it would help make your code more readable if you were to include it.

That is the main data manipulation you need to do in order to run the spawner-recruit analysis. You can fit the model using `nls` (Section 3.4). The model you will fit is called a **Ricker spawner-recruit model** and has the form:

$$R_t = \alpha S_{t-1} e^{-\beta S_{t-1} + \varepsilon_t}, \varepsilon_t \sim N(0, \sigma) \tag{5.1}$$

where α is a parameter representing the maximum recruits per spawner (obtained at very low spawner abundances) and β is a measure of the strength of **density-dependent mortality**. Notice that the error term is in the exponent, which makes e^{ε_t} lognormal. To get `nls` to fit this properly, we will need to fit to $log(R_t)$ as the response variable. Write a function that will predict log recruitment from spawners given the two parameters (ignore the error term):

```
ricker = function(S, alpha, beta) {
  log(alpha * S * exp(-beta * S))
}
```

Fit the model to the data using `nls`. Before you fit it, however, you'll need to do another lag. This is because the run that comes back in one year was spawned by the escapement the previous odd year (you only have odd years in the data). This time extract the appropriate years "by-hand" rather than using the `lag` or `lead` functions as before:

```
# the number of years
nyrs = nrow(sr_dat)
# the spawners to fit to:
S_fit = sr_dat$S[1:(nyrs - 1)]
# the recruits to fit to:
R_fit = sr_dat$R[2:nyrs]
```

Fit the model:

```
fit = nls(log(R_fit) ~ ricker(S_fit, alpha, beta),
          start = c(alpha = 6, beta = 0))
coef(fit); summary(fit)$sigma
```

```
##        alpha         beta
## 5.341553e+00 6.896867e-06
```

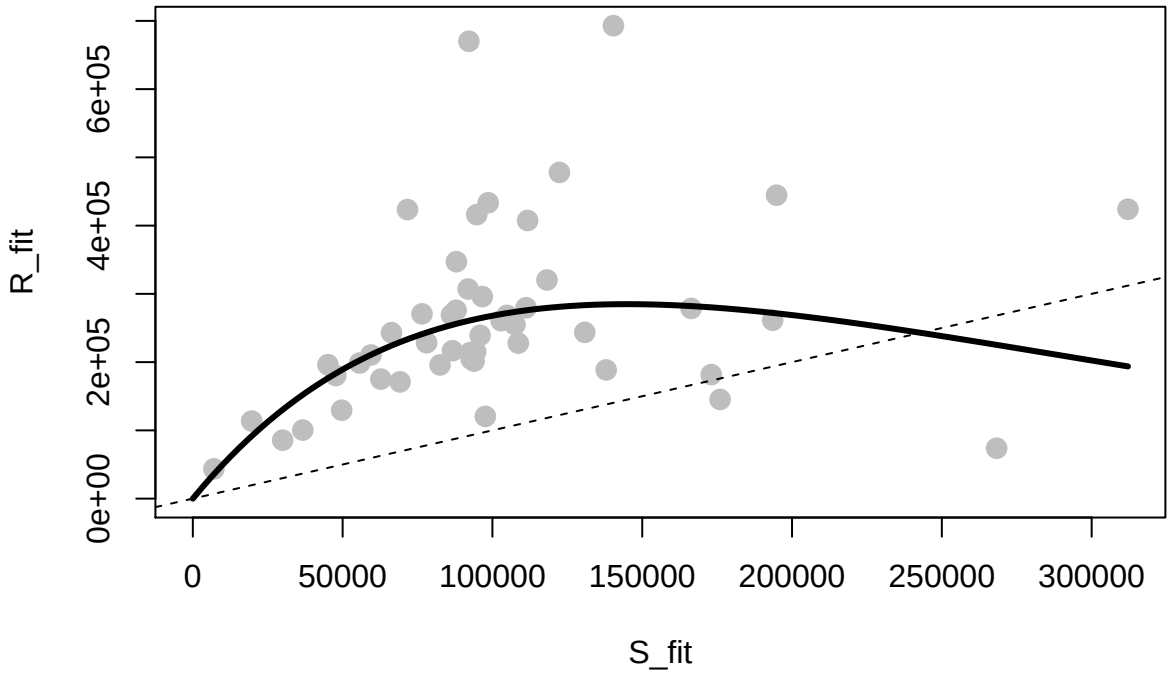
```
## [1] 0.4047541
```

Given that the true values for these data were: $\alpha = 6$, $\beta = 8 \times 10^{-6}$, and $\sigma = 0.4$, these estimates look pretty good.

Plot the recruits versus spawners and the fitted line:

```
# plot the S-R pairs
plot(R_fit ~ S_fit, pch = 16, col = "grey", cex = 1.5,
     xlim = c(0, max(S_fit)), ylim = c(0, max(R_fit)))
# extract the estimates
ests = coef(fit)
# obtain and draw on a fitted line
S_line = seq(0, max(S_fit), length = 100)
R_line = exp(ricker(S = S_line,
```

```
                alpha = ests["alpha"],
                beta = ests["beta"]))
lines(R_line ~ S_line, lwd =3)
# draw the 1:1 line
abline(0, 1, lty = 2)
```



The diagonal line is the **1:1 replacement line**: where 1 spawner would produce 1 recruit. The distance between this line and the curve is the theoretical **harvestable surplus** available at each spawner abundance that would keep the stock at a fixed abundance. The biological reference points that might be used in harvest management are:

$$
\begin{aligned}
S_{MAX} &= \frac{1}{\beta}, \\
S_{eq} &= log(\alpha) S_{MAX}, \\
S_{MSY} &= S_{eq}\,(0.5 - 0.07 * log(\alpha))
\end{aligned}
$$

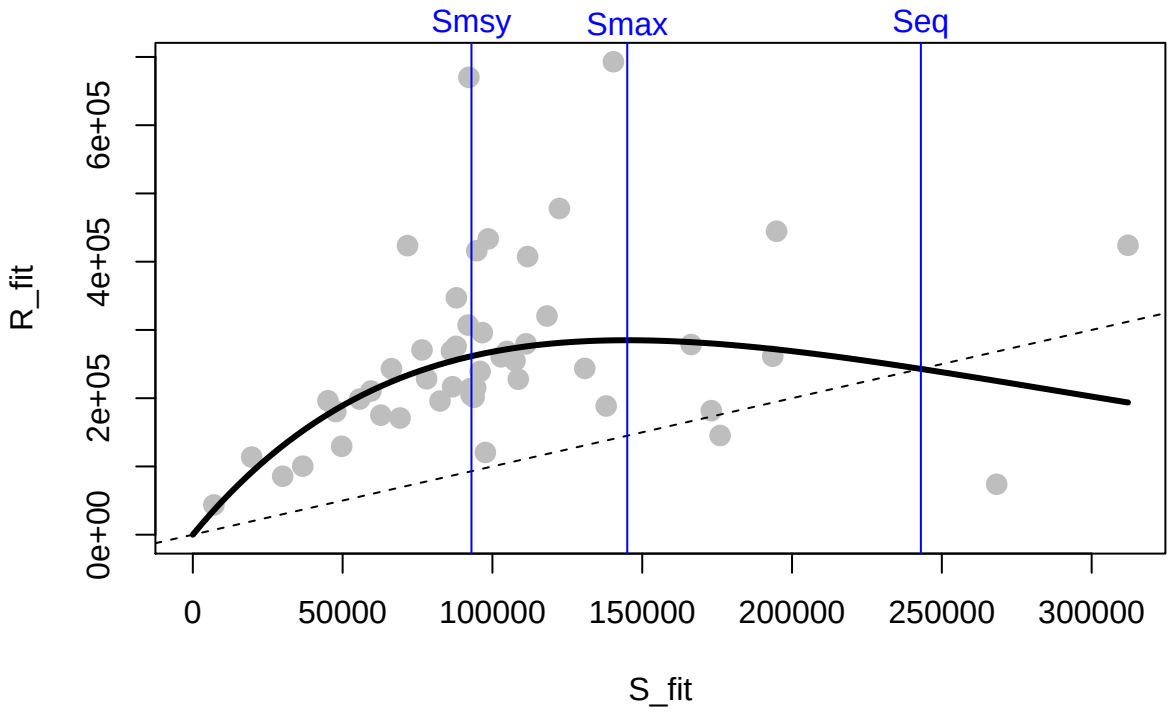
Where S_{MAX} is the spawner abundance expected to produce the maximum recruits, S_{eq} is the spawner abundance that should produce exactly replacement recruits, and S_{MSY} is the spawner abundance that is expected to produce the maximum surplus. You can calculate these from your parameter estimates:

```
Smax = 1/ests["beta"]
Seq = log(ests["alpha"]) * Smax
Smsy = Seq * (0.5 - 0.07 * log(ests["alpha"]))
brps = round(c(Smax = unname(Smax),
```

```
                Seq = unname(Seq),
                Smsy = unname(Smsy)), -3)
```

Draw these reference points onto your plot and label them:

```
abline(v = brps, col = "blue")
text(x = brps, y = max(R_fit) * 1.08,
     labels = names(brps), xpd = T, col = "blue")
```



Exercise 5

In this exercise, you will be working with another simulated salmon data set. This time, you have information on individual fish passing a **weir**. A weir is a blockade in a stream that biologists use to count fish as they pass by. Most of the day, the weir is closed and fish stack up waiting to pass. Then a couple times a day, biologists open a gate in the weir and count fish as they pass. Each year, the biologists sample a subset of fish that pass by to get age, sex, and length. There are concerns that by using large mesh gill nets, the fishery is selectively taking the larger fish and that this is causing a shift in the size-at-age, age composition, and sex composition. If this is the case, it could potentially mean a reduction in productivity since smaller fish have fewer eggs. You are tasked with analyzing these data to determine if these quantities have changed overtime. Note that documenting directional changes in these quantities over time and the co-occurrence of the use of large mesh gill nets does not imply causation.

*The solutions to this exercise are found at the end of this book (here). You are **strongly recommended** to make a good attempt at completing this exercise on your own and only look at the solutions when you are*

truly stumped.

Download the data file `asl.csv` (**a**ge, **s**ex, **l**ength) from the GitHub repo (or go here for details) and save it in your working directory for this chapter. Create a new R script `Ex5.R` there. Open this script and read the data file into R. If this is a new session, load the `{dplyr}` package.

1. Count the number of females that were sampled each year using the `{dplyr}` function `n` within a `summarize` call (*Hint: `n` works just like length - it counts the number of records*).
2. Calculate the proportion of females by year.
3. Plot percent females over time. Does it look like the sex composition has changed over time?
4. Calculate mean length by age, sex, and year.
5. Come up with a way to plot a time series of mean length-at-age for both sexes. Does it look like mean length-at-age has changed over time for either sex?

Exercise 5 Bonus

1. Calculate the age composition by sex and year (what proportion of all the males in a year were age 4, age 5, age 6, age 7, and same for females).
2. Plot the time series of age composition by sex and year. Does it look like age composition as changed over time?

Chapter 6

Mapping and Spatial Analysis

Henry's chapter will go here.

Exercise Solutions

Exercise 1A Solutions

1. Create a new file in your working directory called `Ex1A.R`.

Go to *File* > *New File* > *R Script*. This will create an untitled R script. Go the *File* > *Save*, give it the appropriate name and click *Save*. If your working directory is already set to `C:/Users/YOU/Documents/R-Book/Chapter1`, then the file will be saved there by default.

You can create a new script using **CTRL + SHIFT + N** as well.

2. Enter these data (found in Table 1.1) into vectors. Call the vectors whatever you would like. Should you enter the data as vectors by rows, or by columns? (Hint: remember the properties of vectors).

Because you have both numeric and character data classes for a single row, you should enter them by columns:

```
Lake = c("Big", "Small", "Square", "Circle")
Area = c(100, 25, 45, 30)
Time = c(1000, 1200, 1400, 1600)
Fish = c(643, 203, 109, 15)
```

3. Combine your vectors into a data frame. Why should you use a data frame instead of a matrix?

You should use a data frame because, unlike matrices, they can store multiple data classes in the different columns. Refer back the sections on matrices (Section 1.4.3) and data frames (Section 1.4.4) for more details.

```
df = data.frame(Lake, Area, Time, Fish)
```

4. Subset all of the data from Small Lake.

Refer back to Section 1.7 for details on subsetting using indices and by column names, see Section 1.11 for details on logical subsetting.

```
df[df$Lake == "Small",]
# or
df[3,]
```

5. Subset the area for all of the lakes.

Refer to the suggestions for question 4 for more details.

```
df$Area
# or
df[,2]
```

6. Subset the number of fish for Big and Square Lakes only.

Refer to the suggestions for question 4 for more details.

```
df[df$Lake == "Big" | df$Lake == "Square","Fish"]
# or
df$Fish[c(1,3)]
```

7. You realize that you sampled 209 fish at Square Lake, not 109. Fix the mistake. There are two ways to do this, can you think of them both? Which do you think is better?

The two methods are:

- Fix the mistake in the first place it appears: when you made the `Fish` vector. If you change it there, all other instances in your code where you use the `Fish` object will be fixed after you re-run everything.

```
Fish = c(643, 203, 209, 15)
# re-run the rest of your code and see the error was fixed
```

- Fix the cell in the data frame only:

```
df[df$Lake == "Square","Fish"] = 209
```

The second method would only fix the data frame, so if you wanted to use the vector `Fish` outside of the data frame, the error would still be present. For this reason, the first method is likely better.

8. Save your script. Close RStudio and re-open your script to see that it was saved.

File > *Save* or **CTRL + S**

Exercise 1B Solutions

First, did you find the error? It is the `#VALUE!` entry in the `chao` column. You should have R treat this as an `NA`. The two easiest ways to do this are to either enter `NA` in that cell or delete its contents. You can do this easily by opening `ponds.csv` in Microsoft Excel or some other spreadsheet editor.

1. Read in the data to R and assign it to an object.

After placing `ponds.csv` (and all of the other) to your working directory and creating `Ex1B.R`,

```
dat = read.csv("..Data/ponds.csv")
```

2. Calculate some basic summary statistics of your data using the `summary()` function.

```
summary(dat)
```

3. Calculate the mean chlorophyll *a* for each pond (*Hint: pond is a grouping variable*).

Remember the `tapply()` function. The first argument is the variable you wish to calculate a statistic for (chlorophyll), the second argument is the grouping variable (pond), and the third argument is the function you wish to apply.

```
tapply(dat$chl.a, dat$pond, mean)
```

4. Calculate the mean number of *Chaoborus* for each treatment in each pond using `tapply()`. (*Hint: You can group by two variables with:* `tapply(dat$var, list(dat$grp1, dat$grp2), fun)`.

The hint pretty much gives this one away:

```
tapply(dat$chao, list(dat$pond, dat$treatment), mean)
```

5. Use the more general `apply()` function to calculate the variance for each zooplankton taxa found only in pond S-28.

First, subset only the correct pond and the zooplankton counts. Then, specify you want the `var()` function applied to the second dimension (columns). Finally, because `chao` has an `NA`, you'll need to include the `na.rm = T` argument.

```
apply(dat[dat$pond == "S.28",c("daph", "bosm", "cope", "chao")], 2, var, na.rm = T)
```

6. Create a new variable called `prod` in the data frame that represents the quantity of chlorophyll *a* in each replicate. If the chlorophyll *a* in the replicate is greater than 30 give it a "high", otherwise give it a "low". (*Hint: are you asking R to respond to one question or multiple questions? How should this change the strategy you use?*)

Remember, you can add a new column to a data set using the `df$new_column = something()`. If the column `new_column` doesn't exist, it will be added. If it exists already, it will be written over. You can use `ifelse()` (not `if()`!) to ask if each chlorophyll measurement was greater or less than 30, and to do something differently based on the result:

```
dat$prod = ifelse(dat$chl.a > 30, "high", "low")
```

Bonus 1. Use `?table` to figure out how you can use `table()` to count how many observations of high and low there were in each treatment (*Hint: `table()` will have only two arguments.*).

After looking through the help file, you should have seen that `table()` has a `...` as its first argument. After reading about what it takes there, you would see it is expecting:

> one or more objects which can be interpretted as factors (including character strings)...

So if you ran:

```
table(dat$prod, dat$treatment)
```

```
##
##        Add Control
##   high   8       0
##   low    2      10
```

You would get a table showing how many high and low chlorophyll observations were made for each treatment.

Bonus 2. Create a new function called `product()` that multiplies any two numbers you specify.

See Section 1.14 for more details on user-defined functions. Your function might look like this:

```
product = function(a,b) {
  a * b
}
product(4,5)
```

```
## [1] 20
```

Bonus 3. Modify your function to print a message to the console and return the value `if()` it meets a condition and to print another message and not return the value if it doesn't.

```
product = function(a,b,z) {
  result = a * b

  if (result <= z) {
    cat("The result of a * b is less than", z, "so you don't care what it is")
  } else {
    cat("The result of a * b is", result, "\n")
    result
  }
}
```

```
product(4, 5, 19)
```

```
## The result of a * b is 20
```

```
## [1] 20
```

```
product(4, 5, 30)
```

```
## The result of a * b is less than 30 so you don't care what it is
```

The use of `cat()` here is similar to `print()`, but it is better for printing messages to the console.

Exercise 2 Solutions

1. Create a new R script called `Ex2.R` and save it in the `Chapter2` directory. Read in the data set `sockeye.csv`. Produce a basic summary of the data and take note of the data classes, missing values (`NA`), and the relative ranges for each variable.

File > New File > R Script, then *File > Save > call it Ex2.R > Save*. Then:

```
dat = read.csv("../Data/sockeye.csv")
summary(dat)
```

2. Make a histogram of fish weights for only hatchery-origin fish. Set `breaks = 10` so you can see the distribution more clearly.

```
hist(dat[dat$type == "hatch","weight"], breaks = 10)
```

3. Make a scatter plot of the fecundity of females as a function of their body weight for wild fish only. Use whichever plotting character (`pch`) and color (`col`) you wish. Change the main title and axes labels to reflect what they mean. Change the x-axis limits to be 600 to 3000 and the y-axis limits to be 0 to 3500. (*Hint: The `NA`s will not cause a problem. R will only use points where there are paired records for both x and y and ignore otherwise*).

```
plot(fecund ~ weight, data = dat[dat$type == "wild",],
     main = "Fecundity vs. Weight",
     pch = 17, col = "red", cex = 1.5,
     xlab = "Weight (g)", xlim = c(600, 3000),
     ylab = "Fecundity (#eggs)", ylim = c(0, 3500))
```

All of these arguments are found in Table 2.1.

4. Add points that do the same thing but for hatchery fish. Use a different plotting character and a different color.

```
points(fecund ~ weight, data = dat[dat$type == "wild",],
       pch = 15, col = "blue", cex = 1.5)
```

5. Add a legend to the plot to differentiate between the two types of fish.

```
legend("bottomright",
       legend = c("Wild", "Hatchery"),
       col = c("blue", "red"),
       pch = c(15, 17),
       bty = "n",
       pt.cex = 1.5
)
```

Make sure the correct elements of the `legend`, `col`, and `pch` arguments match the way they were specified in the `plot()` and `lines()` calls!

6. Make a multi-panel plot in a new window with box-and-whisker plots that compare (1) spawner weight, (2) fecundity, and (3) egg size between hatchery and wild fish. (*Hint: each comparison will be on its own panel*). Change the titles of each plot to reflect what you are comparing.

```
vars = c("weight", "fecund", "egg_size")
par(mfrow = c(1,3))
sapply(vars, function(v) {
  plot(dat[,v] ~ dat[,"type"], xlab = "", ylab = v)
})
```

7. Save the plot as a .png file in your working directory with a file name of your choosing.

One way to do this:

```
ppi = 600
png("SockeyeComparisons.png", h = 5 * ppi, w = 7 * ppi, res = ppi)
par(mfrow = c(1,3))
sapply(vars, function(v) {
  plot(dat[,v] ~ dat[,"type"], xlab = "", ylab = v)
})
dev.off()
```

EXERCISE 2 BONUS

Bonus 1. Make a bar plot comparing the mean survival to eyed-egg stage for each type of fish (hatchery and wild). Add error bars that represent 95% confidence intervals.

First, adapt the `calc_se()` function to be able to cope with `NAs`:

```
calc_se = function(x, na.rm = F) {
  # include a option to remove NAs before calculating SE
  if (na.rm) x = x[!is.na(x)]

  sqrt(sum((x - mean(x))^2)/(length(x)-1))/sqrt(length(x))
}
```

Then, calculate the mean and standard error for the % survival to the eyed-egg stage:

```
mean_surv = tapply(dat$survival, dat$type, mean, na.rm = T)
se_surv = tapply(dat$survival, dat$type, calc_se, na.rm = T)
```

Then, get the 95% confidence interval:

```
lwr_ci_surv = mean_surv - 1.96 * se_surv
upr_ci_surv = mean_surv + 1.96 * se_surv
```

Finally, plot the means and intervals:

```
mp = barplot(mean_surv, ylim = c(0, max(upr_ci_surv)))
arrows(mp, lwr_ci_surv, mp, upr_ci_surv, length = 0.1, code = 3, angle = 90)
```

Bonus 2. Change the names of each bar, the main plot title, and the y-axis title.

```
mp = barplot(mean_surv, ylim = c(0, max(upr_ci_surv)),
             main = "% Survival to Eyed-Egg Stage by Origin",
```

```
             ylab = "% Survival to Eyed-Egg Stage",
             names.arg = c("Hatchery", "Wild"))
arrows(mp, lwr_ci_surv, mp, upr_ci_surv, length = 0.1, code = 3, angle = 90)
```

Bonus 3. Adjust the margins so there are 2 lines on the bottom, 5 on the left, 2 on the top, and 1 on the right.

Place this line above your `barplot(...)` code:

```
par(mar = c(2,5,2,1))
```

Exercise 3 Solutions

Exercise 4A Solutions

Exercise 4B Solutions

Exercise 4C Solutions

Exercise 4D Solutions

Exercise 4E Solutions

Exercise 4F Solutions

Exercise 5 Solutions

Exercise 6 Solutions

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., and Chang, W. (2018). *rmarkdown: Dynamic Documents for R*. R package version 1.10.

Anderson, D. R., Burnham, K. P., and Thompson, W. L. (2000). Null hypothesis testing: Problems, prevalence, and an alternative. *The Journal of Wildlife Management*, 64(4):912–923.

Kline, P. A. and Flagg, T. A. (2014). Putting the red back in redfish lake, 20 years of progress toward saving the pacific northwest's most endangered salmon population. *Fisheries*, 39(11):488–500.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown*. Chapman and Hall/CRC, Boca Raton, Florida. ISBN 978-1138700109.