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, 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. 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 fitting statistical models using built-in functionality for generalized linear models as well as non-linear models.

- **Chapter 4** covers the basics simulation modeling in R.
- **Chapter 5** covers the basics 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.

Data sets

Many of the data sets used in this book were simulated (using R!) by the author. In cases where the data set used is not simulated, this is noted and a citation to the data source is provided.

The data sets used in this book are hosted on a GitHub repository: **INSERT GITHUB INFO HERE**. **INSERT INSTRUCTIONS HERE**

0.1 Development of this book

This book was created using R and the packages `bookdown` and `knitr`. It is hosted on GitHub Pages

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.

---

[1]While it is possible to run R on it's 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

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":

- 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

---

[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.

- 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.
- 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

You will want to save your hard work. A key part of doing saving your work is thinking about **where** you save it. In R, a key concept is the **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, and is where all of your data should be stored for your analysis in R. 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.

Save your script somewhere like `C:/Users/YOU/Documents/R-Workshop/Chapter1` on your computer. To set the working directory to this location, 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-Workshop/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.

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-Workshop/Chapter1/file.txt"` each time.

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 lots of built-in information 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 termed an **element**, and the number of elements is termed 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`:

```r
length(year)
```

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

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

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

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

```r
# 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:

```r
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 we attempted to calculate the sum of our vector:

```r
sum(v)
```

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

we 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. One easy way to make a matrix is by combining vectors you have already made:

```r
# 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
```

```r
# 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"
```

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. Simply run `class(object.name)`:

```
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 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
```

That looks weird, huh? 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 3.3 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
```

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 (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 a data object (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. To subset by index, you specify the object, then what rows, then what columns. 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 3
m1; m1[1,3]
```

```
##       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
```

```
## year
## 2018
```

```
# 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.

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[,"year"]
```

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

The by name (`$`) 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, and is described in Section 1.11.

EXERCISE 1A

Take a break to apply what you've learned so far to enter this data frame into R by hand and do some basic data subsets.

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

  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("FileName.csv")
```

The data files for this workshop are found in the **GitHub repository, reminder**. Navigate to this repository, find the file called `streams.csv`, download that file[8], and place it in your working directory for this session. This document assumes your working directory is `C:/Users/YOU/Documents/R-Workshop/Chapter1`, though the one you are actually using may be slightly different (the `YOU` should at least be changed for your computer). We can read the contents of `streams.csv` into R using this code (ensure your working directory is set first):

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

If `streams.csv` is in your working directory, all you need to provide is the file name. 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 has 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 well as do the quotes and `.csv` at the end. Ensure the file name is typed correctly.

2. Check what files are in your working directory: run `dir()`. This will return a vector with the names of the files located in 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?

3. If the file is not in your working directory, and you wish to keep it that way, you must point R to where that file is. Pretend for a moment that you put `streams.csv` in `/R-Workshop/DataHere`. Here are two examples that would both read in the file in this case:

```
dat = read.csv("C:/Users/YOU/Documents/R-Workshop/DataHere/streams.csv")
dat = read.csv("../DataHere/streams")
```

The first line is the full file path to the file in question. The second line uses a **relative path**: the `../` says "look one folder up from the working directory for a folder called `DataHere` then find a file called `streams.csv` in it."

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:

---

[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.

[8]To download a single file from GitHub, click on the file, then click "Raw" in the toolbar on the top right, then right-click anywhere in the document and click "Save As…". You can also download a zip folder with all of the `.csv` files, go to "Clone or Download"

- 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, `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 `NA`s, 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:

```r
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!).

```r
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`:

```r
colnames(dat)
```

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

Calculate the mean of all the `stream_width` records:

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

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

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

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

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

It returned an `NA` because there is an `NA` in the data for this variable. There is a way to tell R to ignore this `NA`: include 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).

```r
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):

```r
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:

```r
apply(dat[,2:3], 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 we 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 equals-sign to denote equality as opposed to the single `=` as used in assignment when you ran `x = 5` a minute ago.

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.

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:

```r
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:

```r
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:

```r
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:

```r
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:

```r
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 ask one question at a time. If you supply it with a vector of length greater than 1, it will give a warning:

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

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

```
## Warning in if (xs < 0) print("negative"): 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 ask multiple questions at once, we must use `ifelse`. This function is similar, but it combines the `if` and `else` syntax into one useful function function:

```r
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:

```r
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 the stream width is greater or less than 50**:

```r
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[9]. **What if we wanted three categories: 'small', 'medium', and 'large'?**

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

---

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

```
##      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:

```r
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.

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` to your working directory (in its current state) then load it in to a future R session. You can save the new data frame using:

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

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

```r
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. We will be going into more detail 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 (I know). 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 4 different zooplankton taxa. In this experiment, there were two ponds, two treatments per pond, and five replicates of each treatment. There is one error in the data set "ponds.csv" (**found on GitHub, reminder**). 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.

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 a question of a single number or multiple numbers? 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:

- the different types of plotting functions
- the basics of how to make:
    - 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 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 document.

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 a directory like `C:/Users/YOU/Documents/R-Workshop/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:

Main Plot Title Goes Here
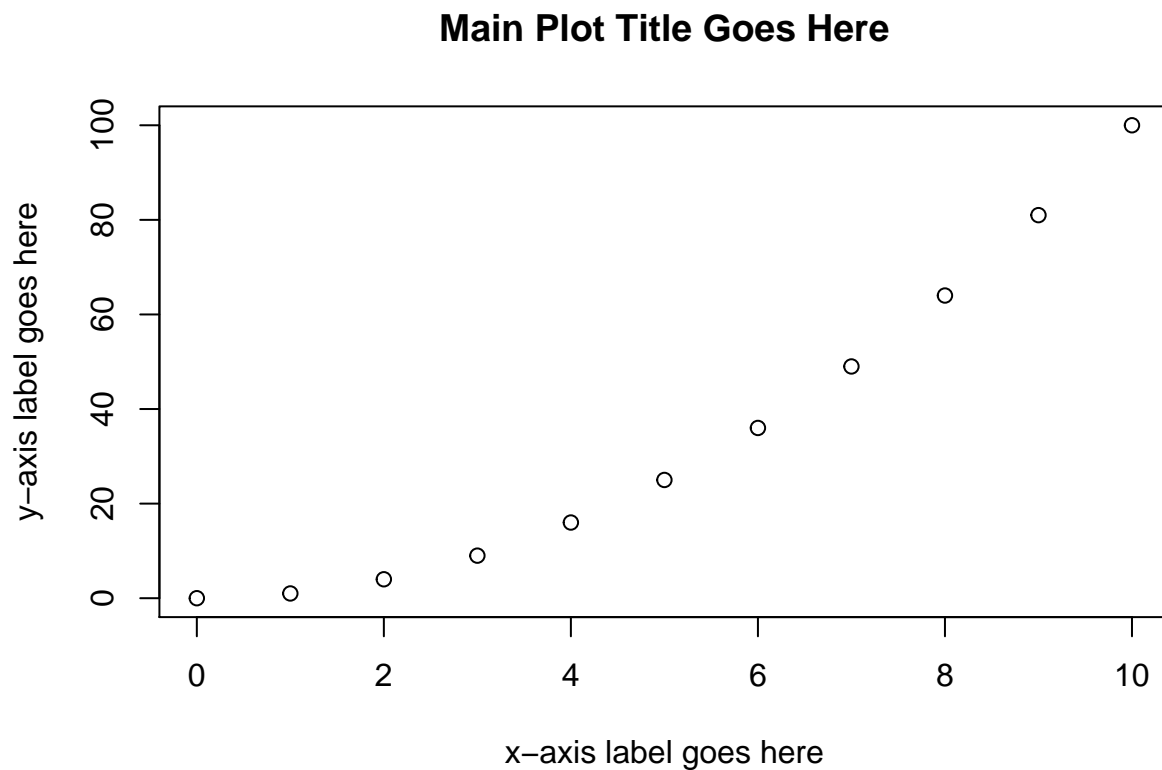
There are a few components:

- 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

Arg.	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^['cex' is a multiplier: 'cex = 1.5' says mal
'pch'	'pch = 17'	changes the symbol type^[There are approximately 20 different 'pch' settings: 'pch = 1' is e
'xlim'	'xlim = range(x)'	changes the endpoints (limits) of the x-axis^['xlim' and 'ylim' both require a numeric vector
'ylim'	'ylim = c(0,1)'	same as 'xlim', but for the y-axis
'type'	'type = "l"'	changes the way points are connected by lines^[The default is points only, 'type = "l"' is for
'lty'	'lty = 2'	changes the line type^['lty = 1' is solid, 'lty = 2' is dashed, 'lty = 3' is dotted, etc. You can
'lwd'	'lwd = 2'	changes the line width^[works just like 'cex': 'lwd = 3' codes for a line that is 3 times as th
'col'	'col = "blue"'	changes the color of plotted objects^[there is a whole host of colors you can pass R by name

```
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.

Here are just a handful of them to get you started:

You are advised to try at least some of these arguments 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)
```

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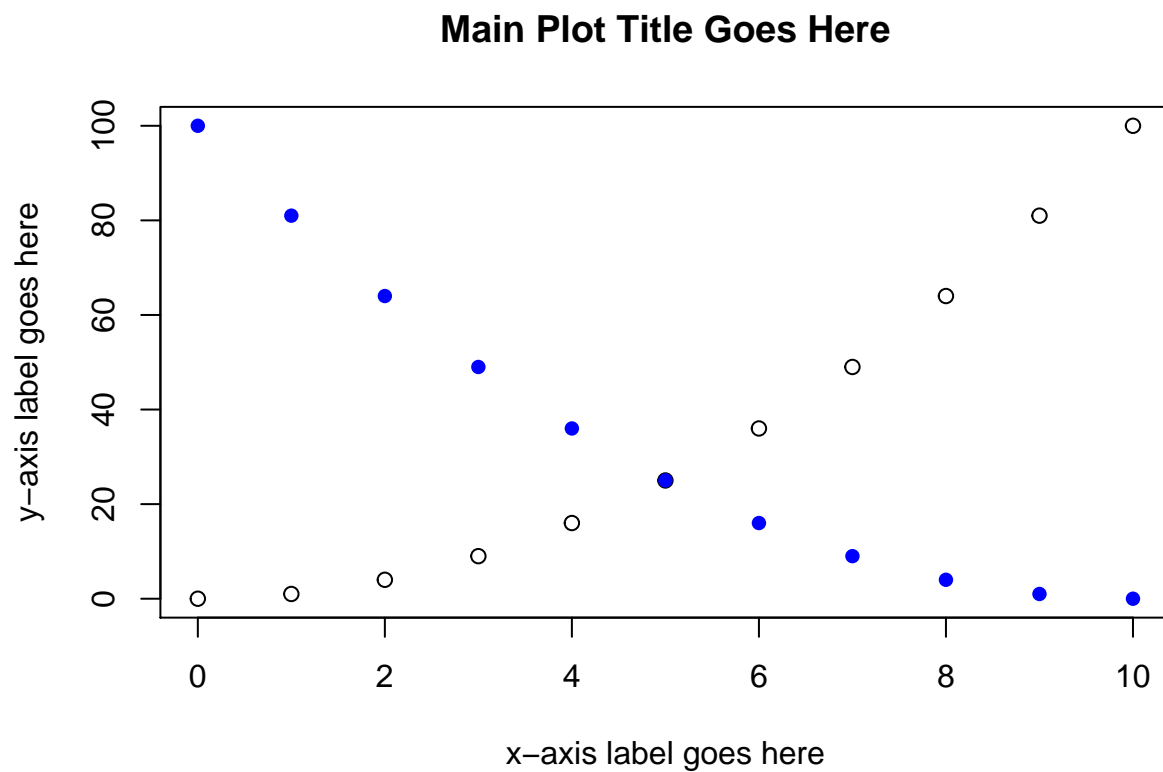
Here, `points` acted like a low-level plotting function because it added points to a plot you already made. Many of the arguments shown in Table 2.1 can be used in both high-level and low-level plotting functions (notice how `col` and `pch` were used in `points`). Just like `points`, there is also `lines`:

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

Main Plot Title Goes Here
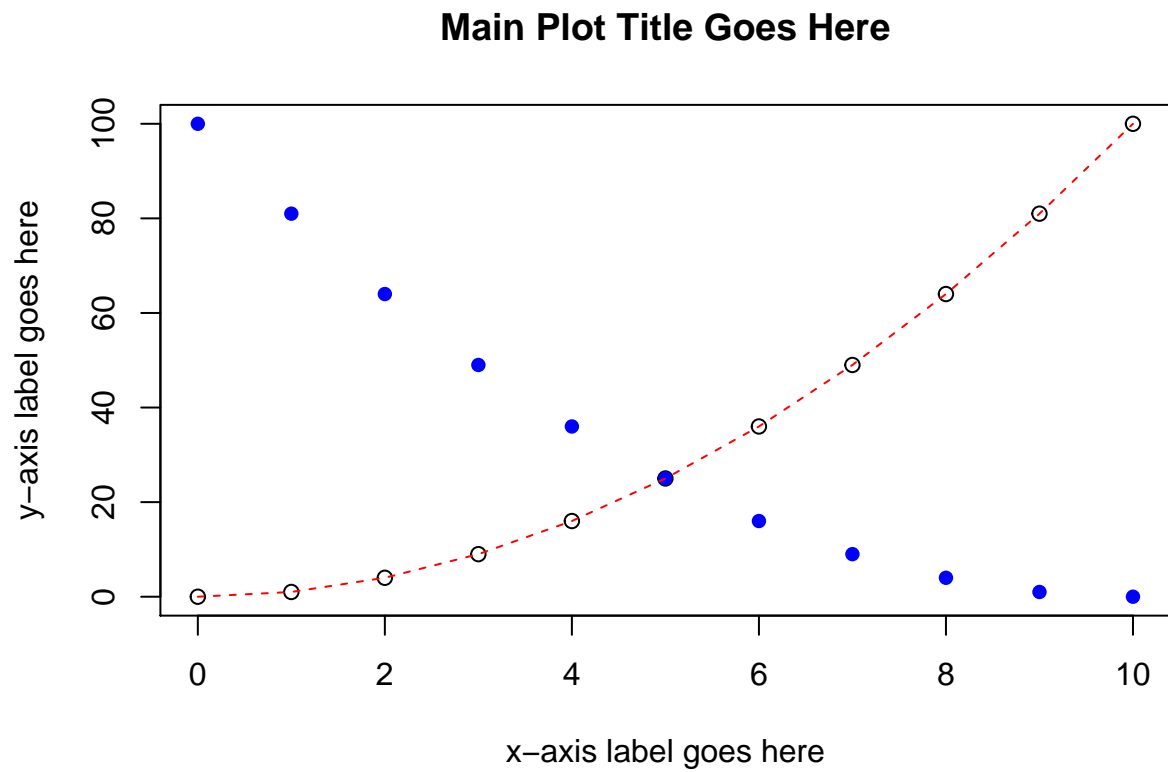
You can also add text to the plotting region:

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

Main Plot Title Goes Here

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")
```

Main Plot Title Goes Here
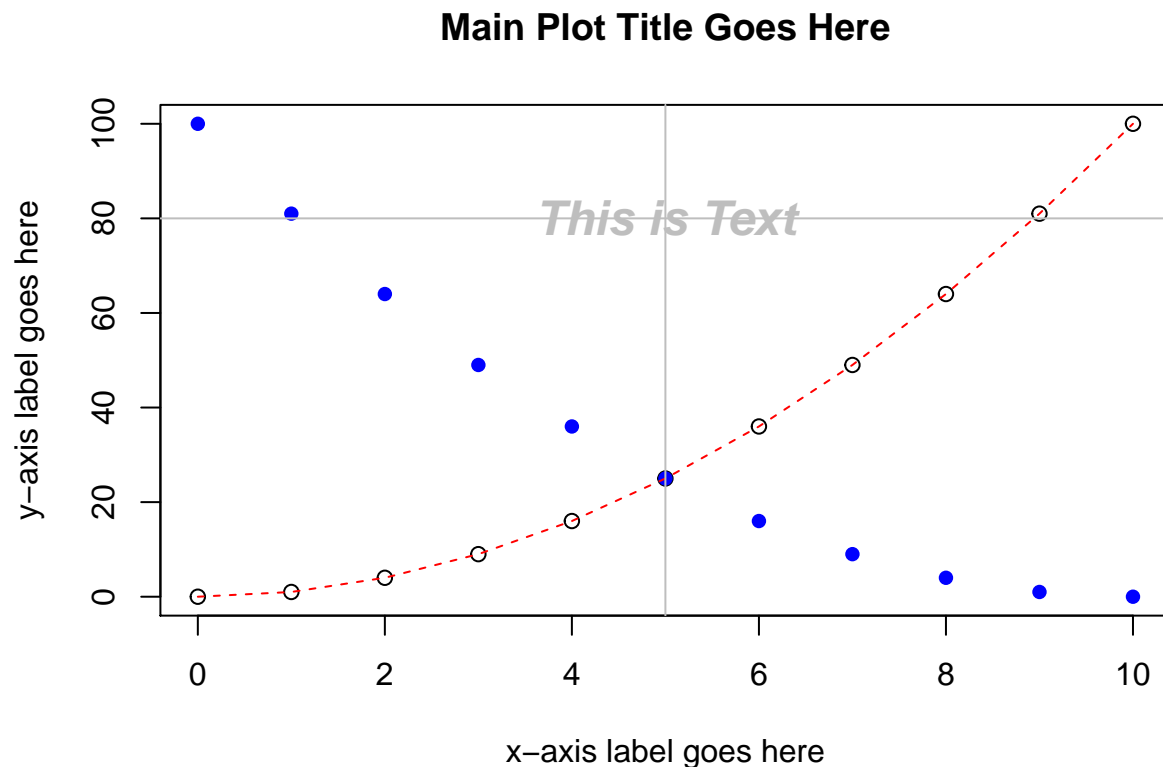
If you accidentally add a plot element that you don't want using a low-level plotting function, the only way to remove it is by re-running the high-level plotting function to start a new plot and adding only the objects you want. Try removing the "This is Text" text and the straight lines you drew with `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:
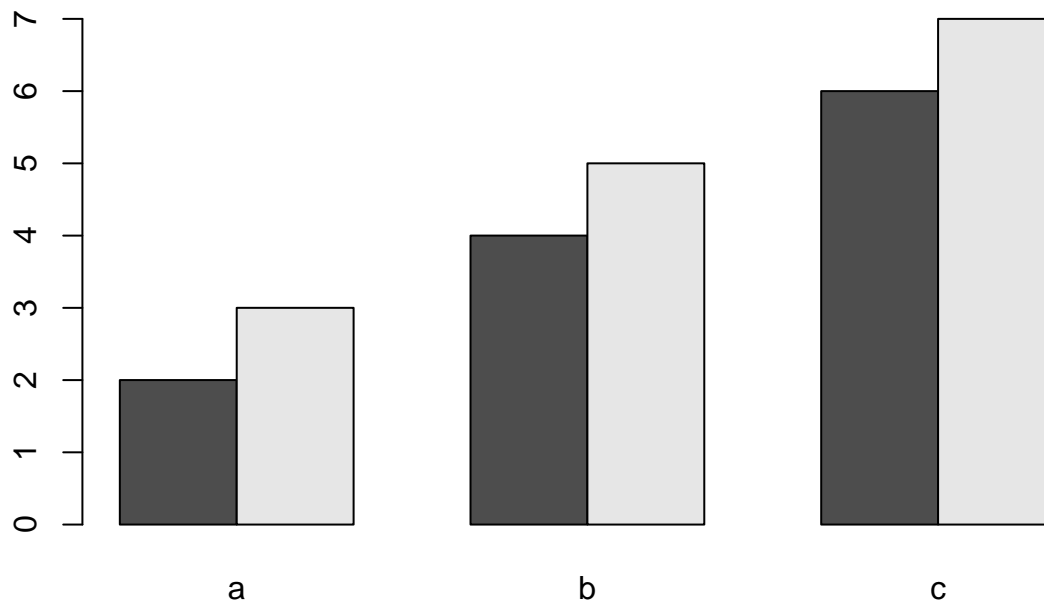
```
x1 = c(2,4,6)
barplot(x1)
```
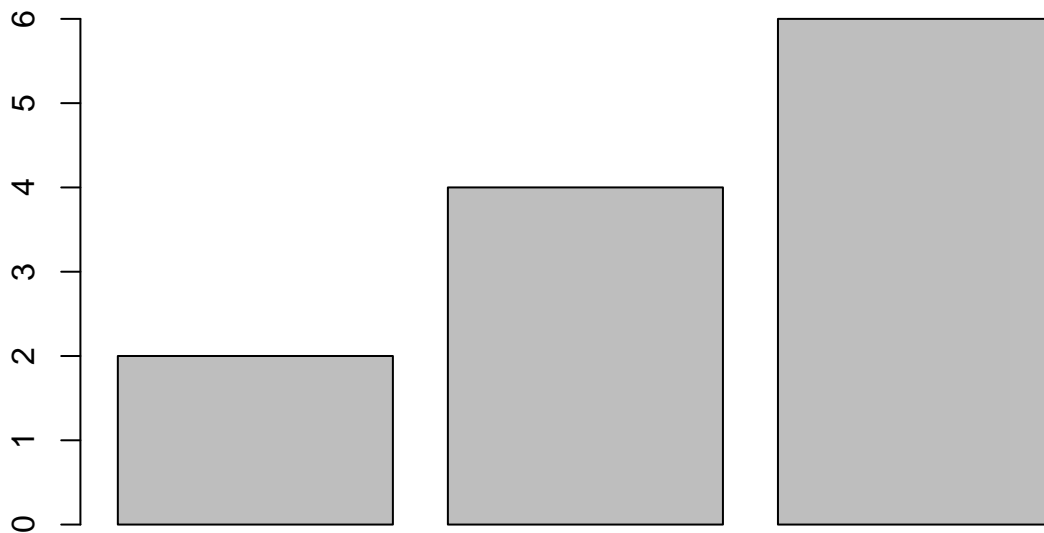
Notice that there are no group names on the bars (if **x1** had names, there would be). You can add names by using the argument **names.arg**:

```
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.

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. Download the `creel.csv` data set from the **GitHub repository, reminder**, place it in the same directory as your R script `Ch2.R` (which is the one you should be working on right now).

Read the data in and print a summary:

```
dat = read.csv("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:

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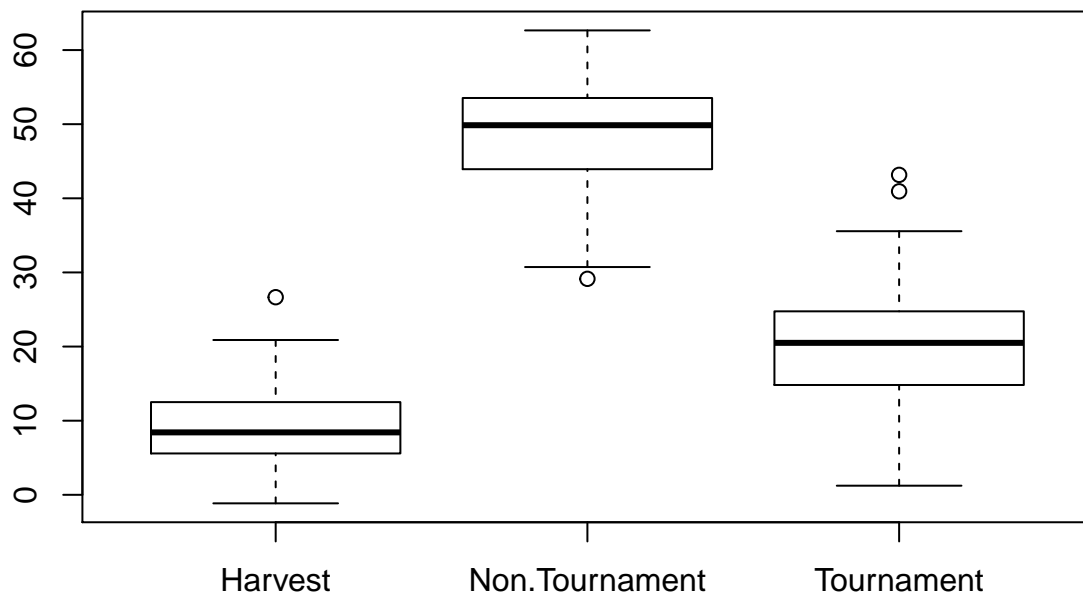

In the box-and-whisker plot above, the heavy line is the median, the ends of the boxes are the 25th and 75th percentiles and the "whiskers" are the 2.5^{th} and 97.5^{th} percentiles. Any points that are outliers (i.e., fall outside of the whiskers) will be shown as points[3].

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

```r
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:

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


Histogram of dat$hours[dat$fishery == "Tournament"]
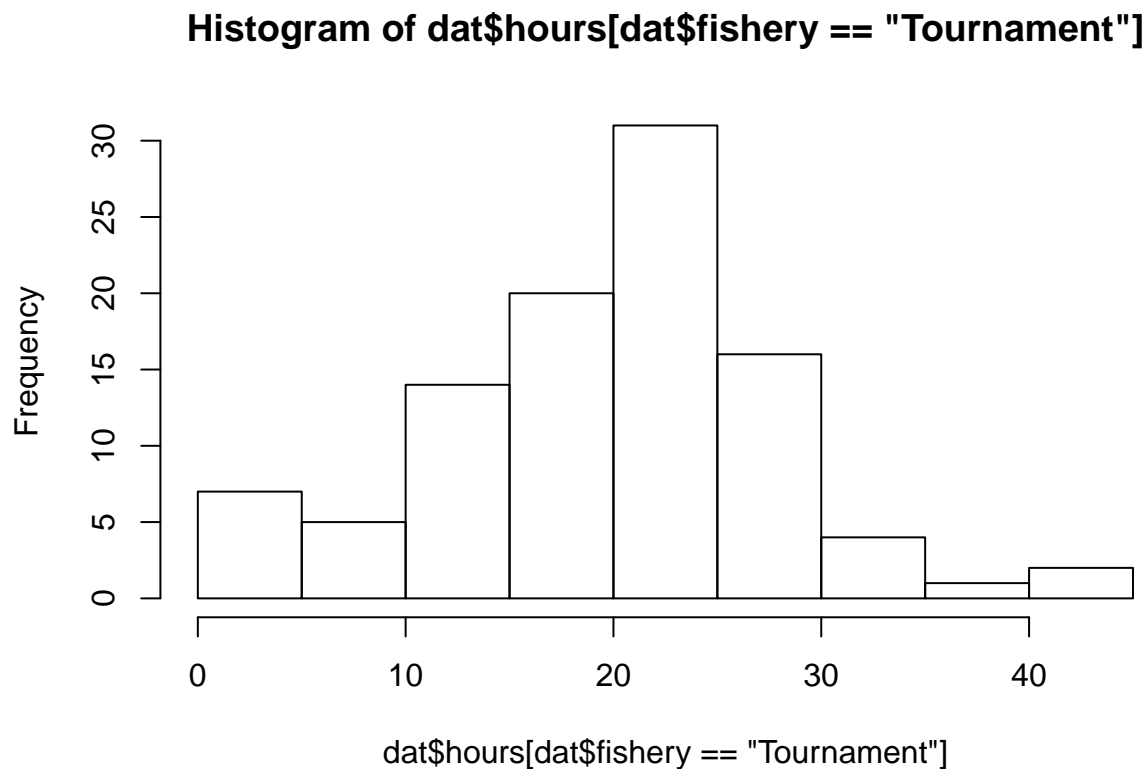
Notice the subset that extracts hours fished for tournament anglers only before plotting.

`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:

```r
# 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")
```
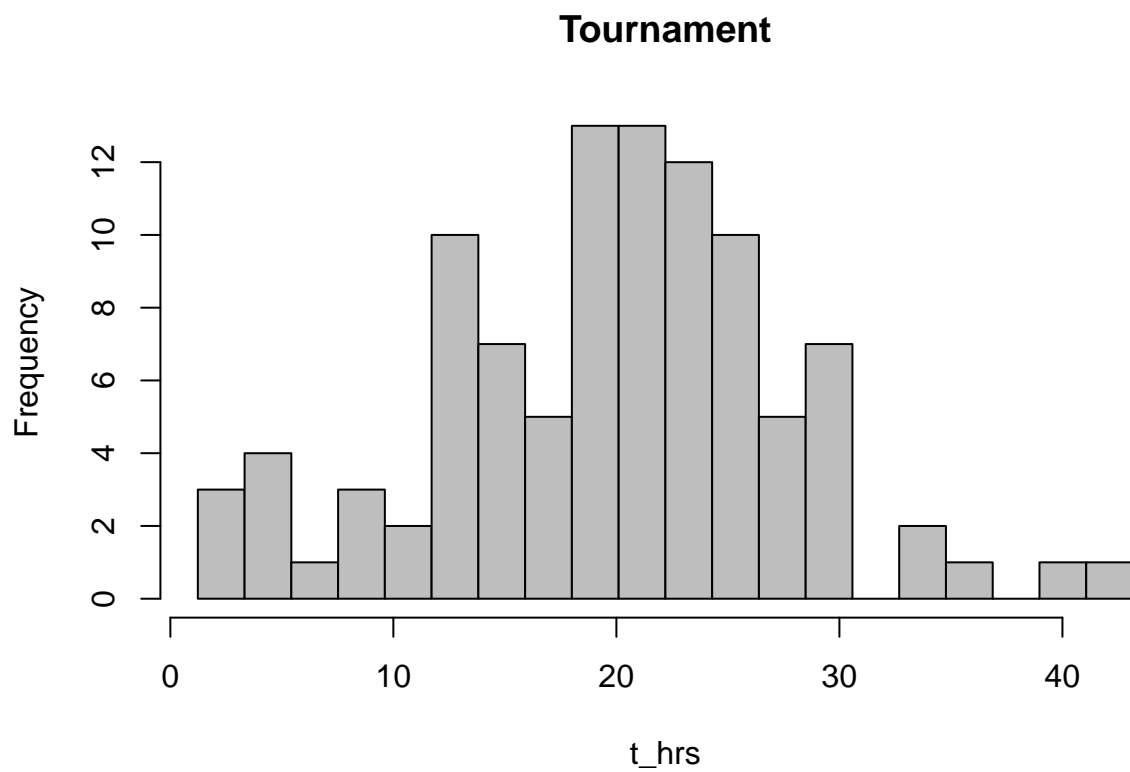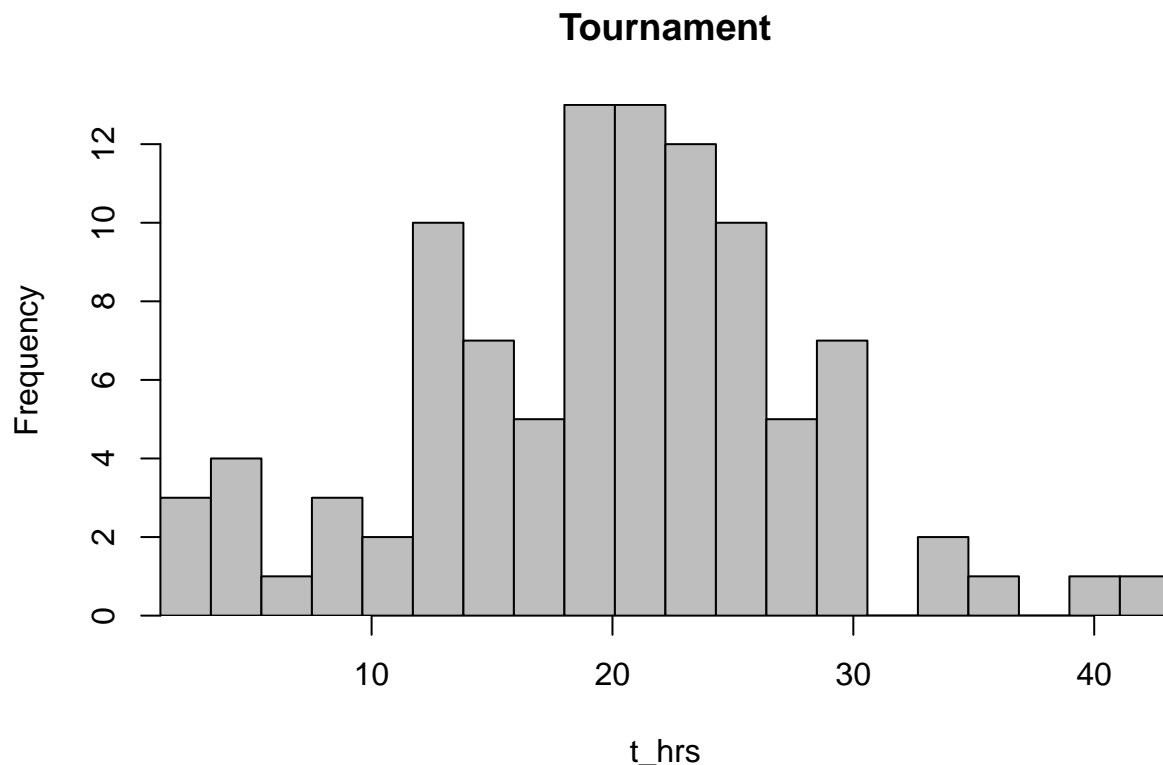
Tournament

2.6   The `par` Function

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

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

Tournament

Here, you changed the graphical parameters of the graphics device by using the `par` argument. 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:

```r
par(mfrow = c(1,3))
hist(dat$hours[dat$fishery == "Tournament"], main = "Tourn")
hist(dat$hours[dat$fishery == "Non.Tournament"], main = "Non.Tourn")
hist(dat$hours[dat$fishery == "Harvest"], main = "Harvest")
```
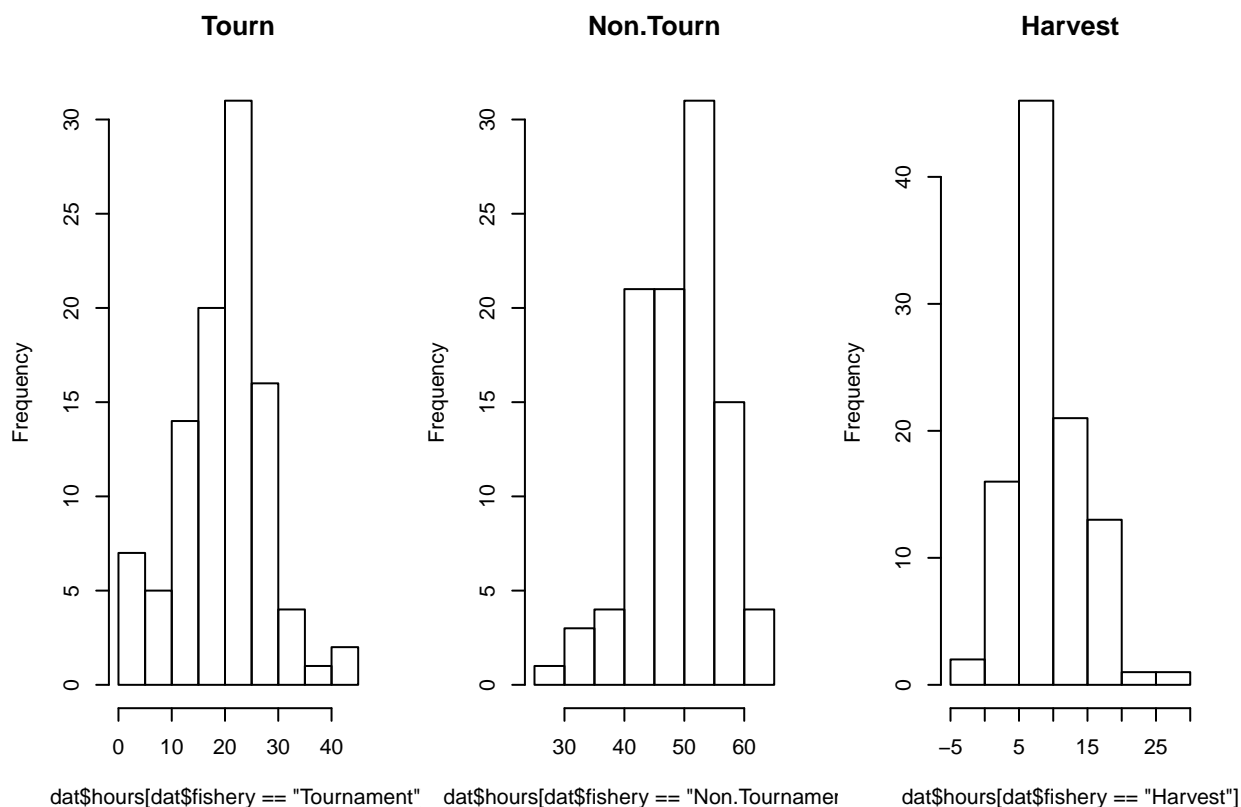

There are other ways to make multi-panel plots, however, they are beyond the scope of this beginner's workshop. See `?layout` for details. With this function you can change the size of certain plots and make them have different shapes (i.e., some squares, some rectangles, etc.), but it takes some pretty 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:

```r
barplot(x3, beside = T,names.arg = c("a", "b", "c"), col = c("skyblue2", "tomato2"))
legend("topleft", legend = c("Group 1", "Group 2"), fill = c("skyblue2", "tomato2"))
```
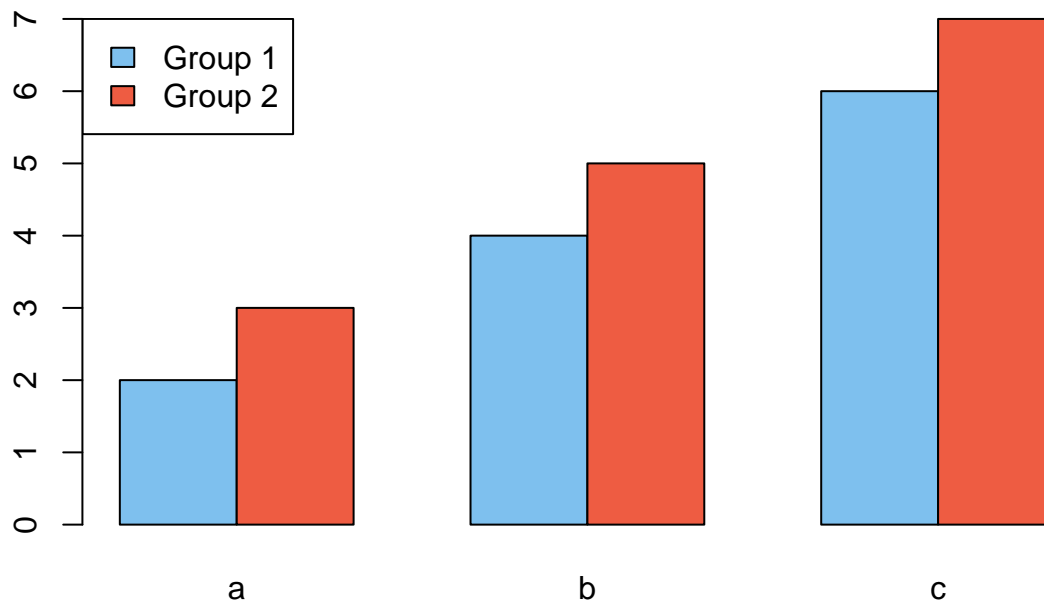
The box can be removed using the `bty = "n"` argument and the size can be changed using `cex`. The position can be specified either with words (like above) or by using x-y coordinates.

Here is a more complex example:

```r
# 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")
```
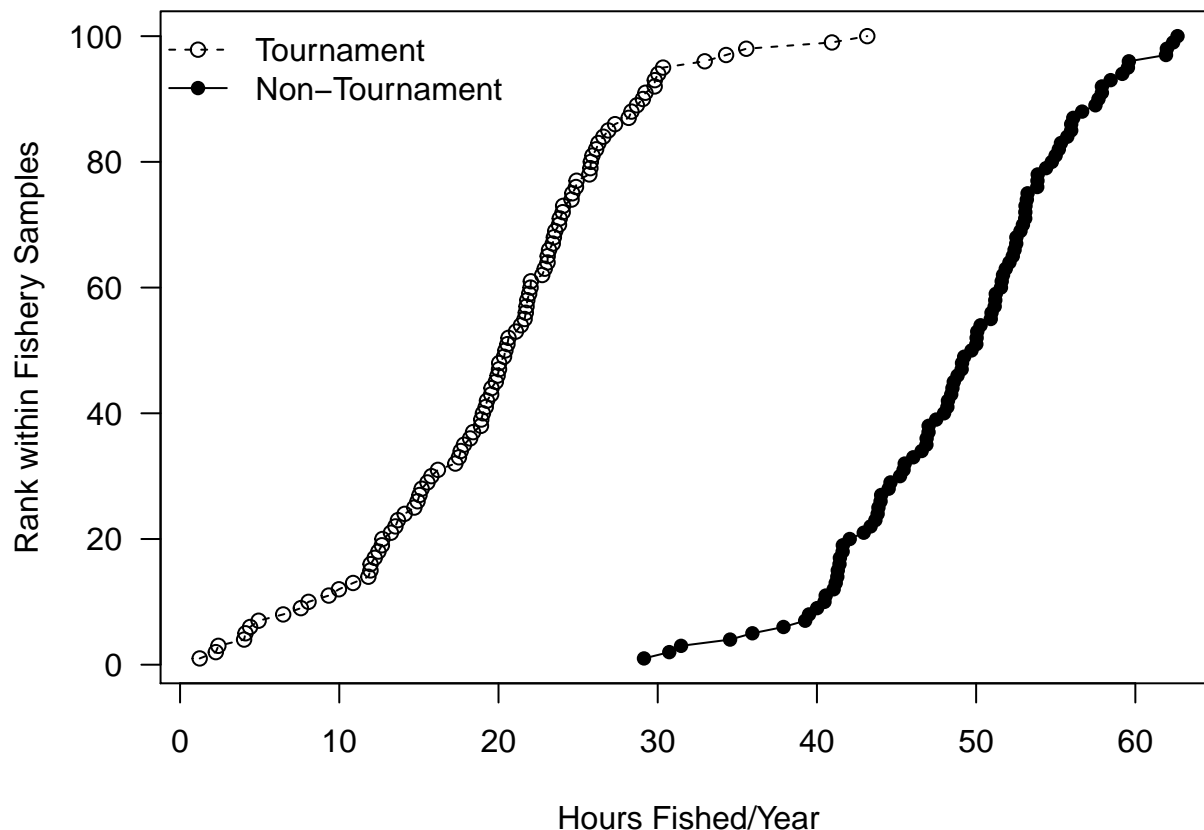

Notice that you need to be careful about the order of how you specify which `lty` and `pch` settings match up with the elements of the `legend` argument. In the `plot code`, you specified that the `lty = 2` but didn't specify what `pch` should be (it defaults to `1`). So when you put the "Tournament" 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 run.

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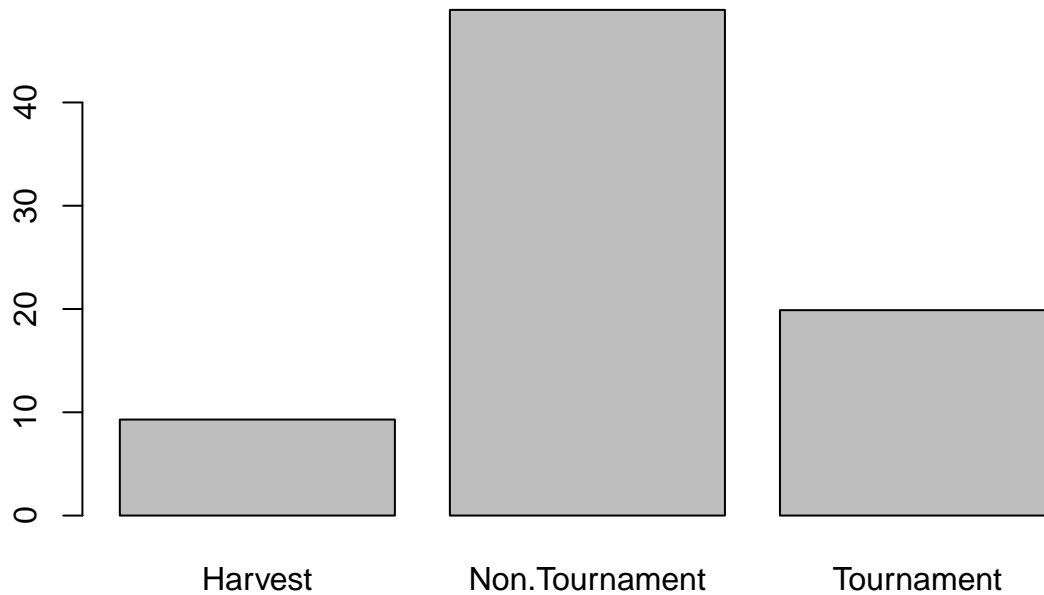
You wish to add error bars that represent 95% confidence intervals on the mean. You can create a 95% confidence interval using this basic formula:

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

where

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

and

$$SE(\bar{x}) = \sqrt{\frac{\sum_{i}^{n} (x_i - \bar{x})^2}{n - 1}}$$

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))
}
```

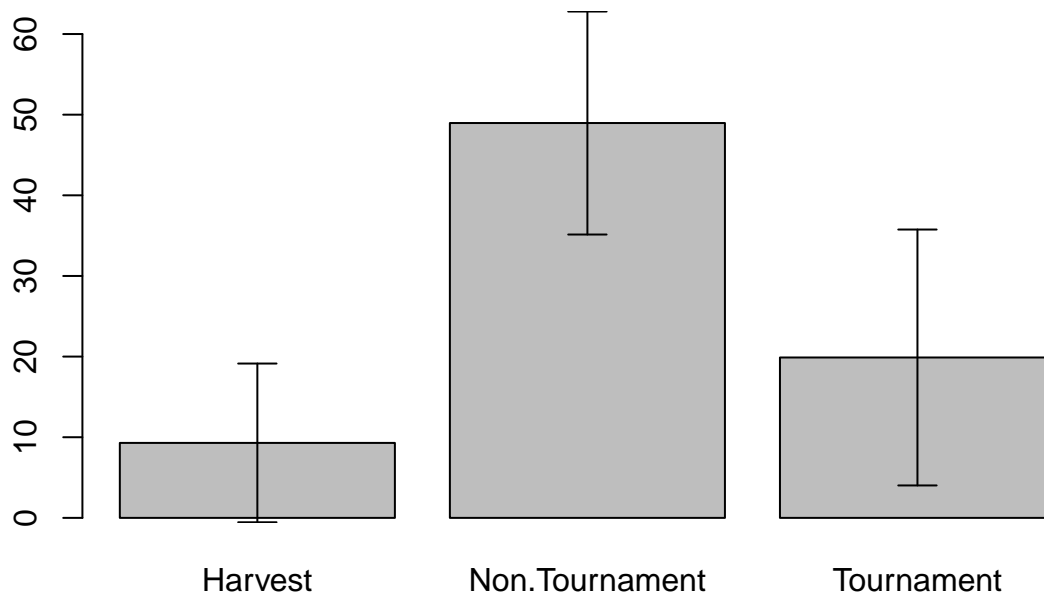Then calculate the standard errors for each fishery type:

```
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(lwr, 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` coordinate. 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(lwr, 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 this time from a sockeye salmon (*Oncorhynchus nerka*) population from the Columbia/Snake River system. 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 preserve 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.

1. Create a new R script called `Ex2.R` and save it in the `Chapter2` directory. Download the `sockeye.csv` data set from **GitHub, reminder** and read it into R. 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 `NAs` 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 +/- 2SE of each mean.

2. Change the names of each bar, the main plot title, and the y-axis title.

Reference for data Kline, P.A. and T.A. Flagg. 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.

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:

- how to fit and interpret the output from various general linear models:
  - simple linear regression models
  - multiple regression models
  - higher order polynomial regression models
  - T-tests (also ANOVA)
  - ANCOVA models
  - Interactions
- very basic model selection
- basic GLMs: the logistic regression model
- Bonus topic: fitting non-linear regression models using `nls`
- Bonus topic: fitting custom maximum-likelihood models using `optim`

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 document.

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-Workshop/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   Review: The General Linear Model

This 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:

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

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 they are "significantly" different from the value given 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`, and store the results in a new object called `fit`".

3.2   Simple Linear Regression

Download the data set `sockeye.csv` from GitHub and place it in your working directory. This is the same data set you used in Exercise 2, see that section for more details on the different variables.

Read these data into R:

```
dat = read.csv("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`, but `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 of an individual to the average weight of an individual by treating years as replicate data points. 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
```

---

[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

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

If x_{i1} is `weight`, then 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 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)
```
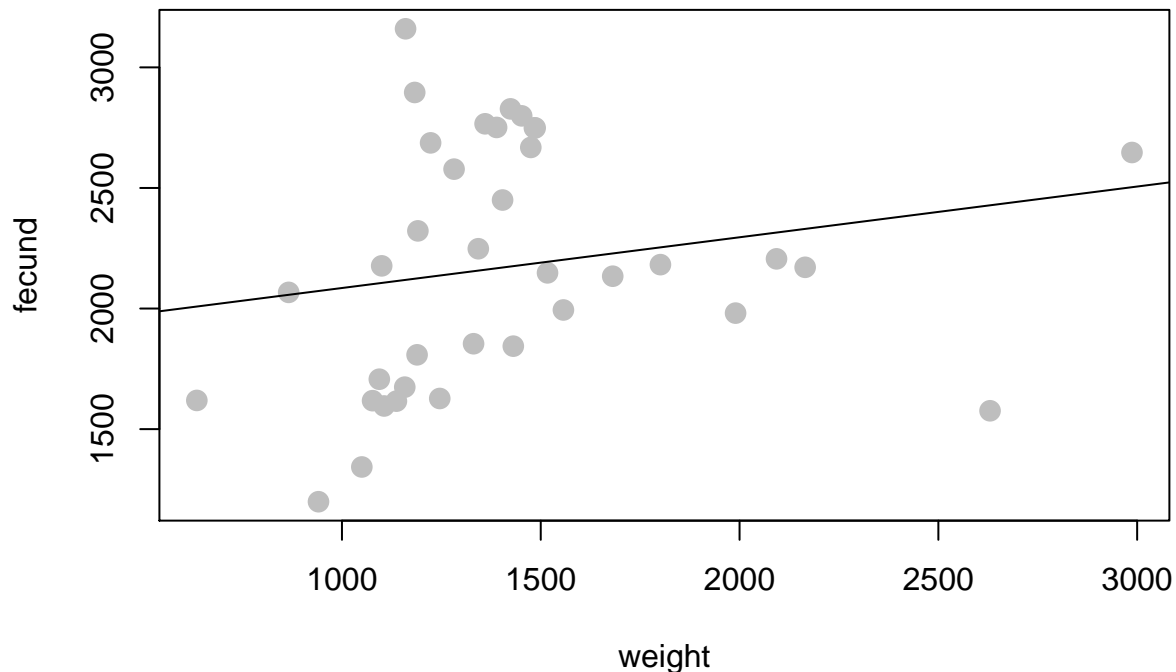
It fits, but not very well. It seems there are two groups: one with data points mostly above the line and one with data points mostly below the line. You'll now run a new model to get at this.

3.3   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, the 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:

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

and assume that $x_{i1} = 0$ if observation i is from `"hatch"` fish and $x_{i1} = 1$ if observation i is from `"wild"` fish. In that case:

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

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.

3.4 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 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:

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

If x_{i1} is `type` coded with 0's and 1's as in Section 3.3 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 both lines (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)
```
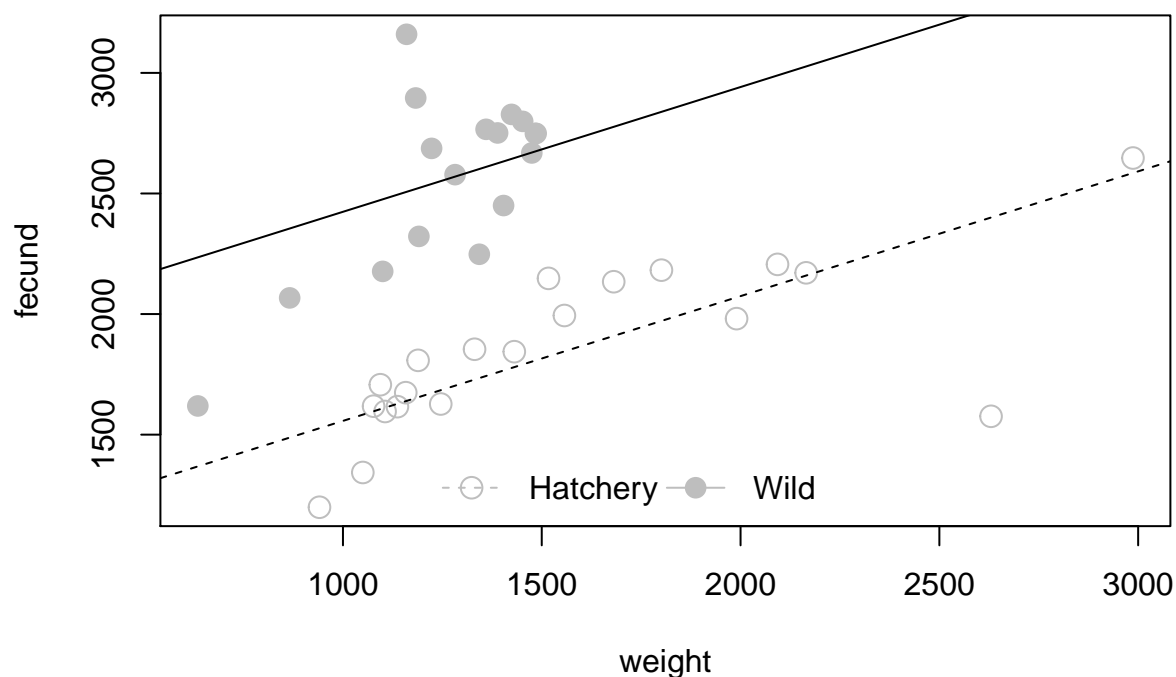
3.5 Interactions

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:

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

If x_{i1} is `type` coded with 0's and 1's as in Section 3.3 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.

You can 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)
```
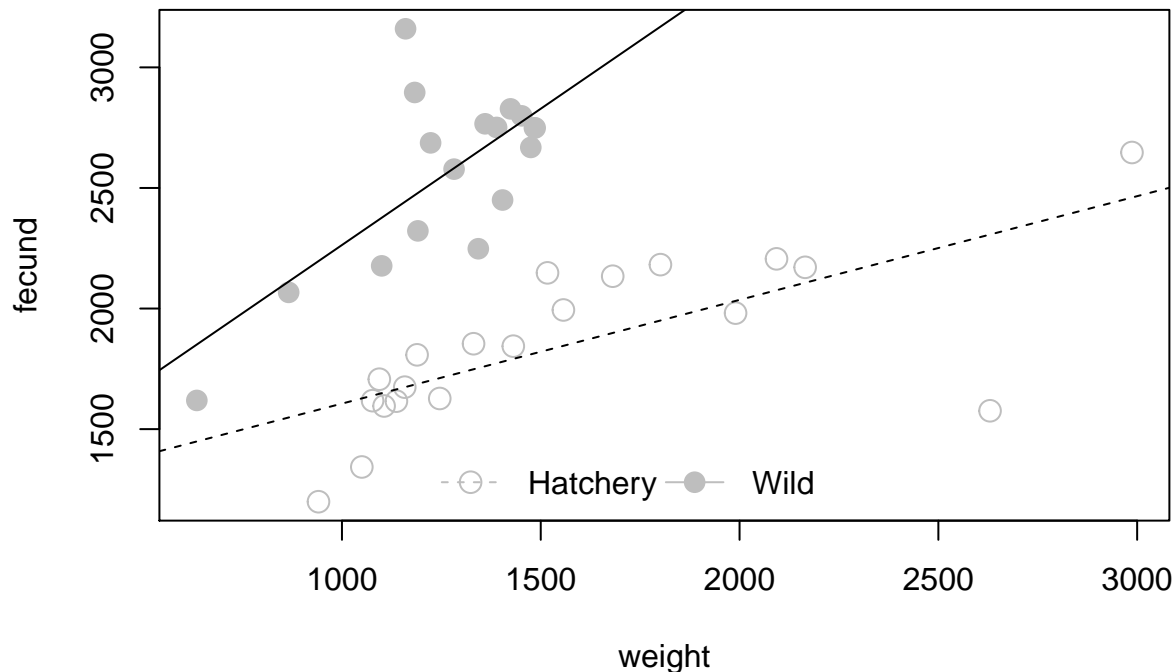
Based on the coefficients table:

```
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.

3.6   Model Selection: AIC

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. Compare the four models you fitted with AIC:

```
AIC(fit1, fit2, fit3, fit4)
```

```
##      df      AIC
```

---

[3] Akaike's Information Criterion. **GIVE A CITATION HERE**

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

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.

3.7 An Example GLM: Logistic Regression

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

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:

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

Where p_i is the probability that observation y_i was a success ($y_i = 1$). 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:

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

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:

```
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) and need to get `p[i]`, you can apply the inverse logit function:

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

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 previous models you've fitted in this chapter: 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

---
[4]General linear models are a member of this family

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.

A basic model would have just `weight` as the predictor variable:

```
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:

- β_0: the log odds of success for a fish with zero weight (which is not all that important). 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. Remember "success" is defined as having at least 70% egg survival to the stage of interest.

- β_1: the additive effect of fish weight on the log odds of success. e^{β_1} is interpretted as the ratio of the odds at two consective weights (e.g., 1500 and 1501). 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 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)
```
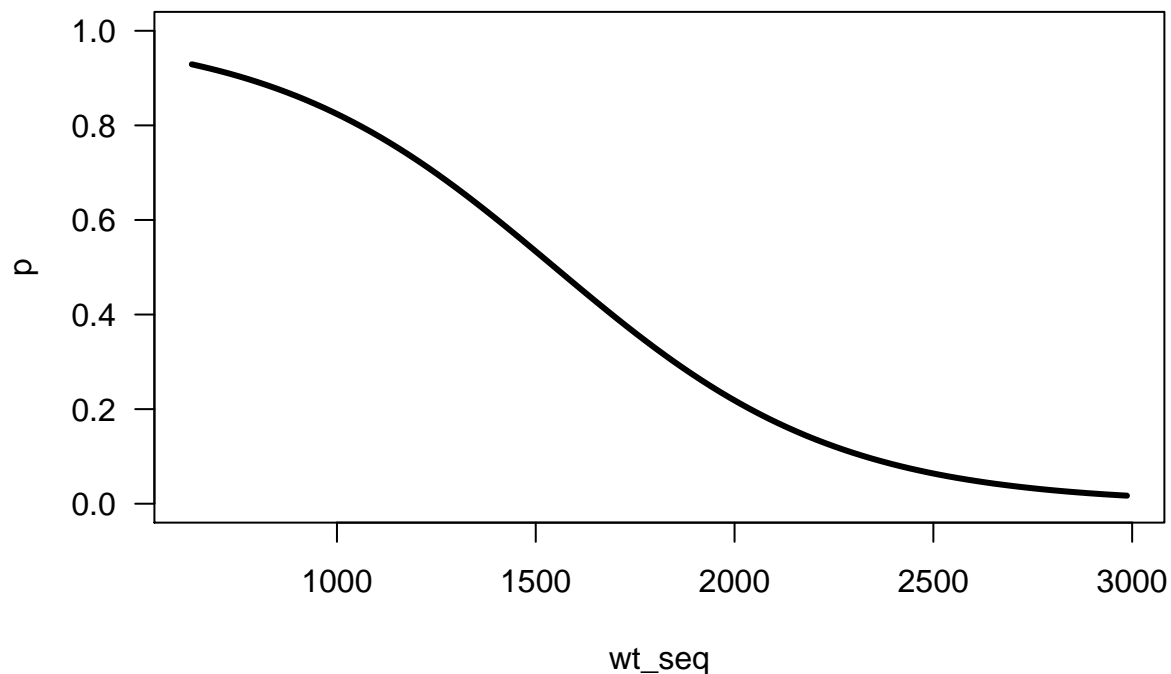
Fit another model comparing the success rates between hatchery and wild fish:

```
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")
```
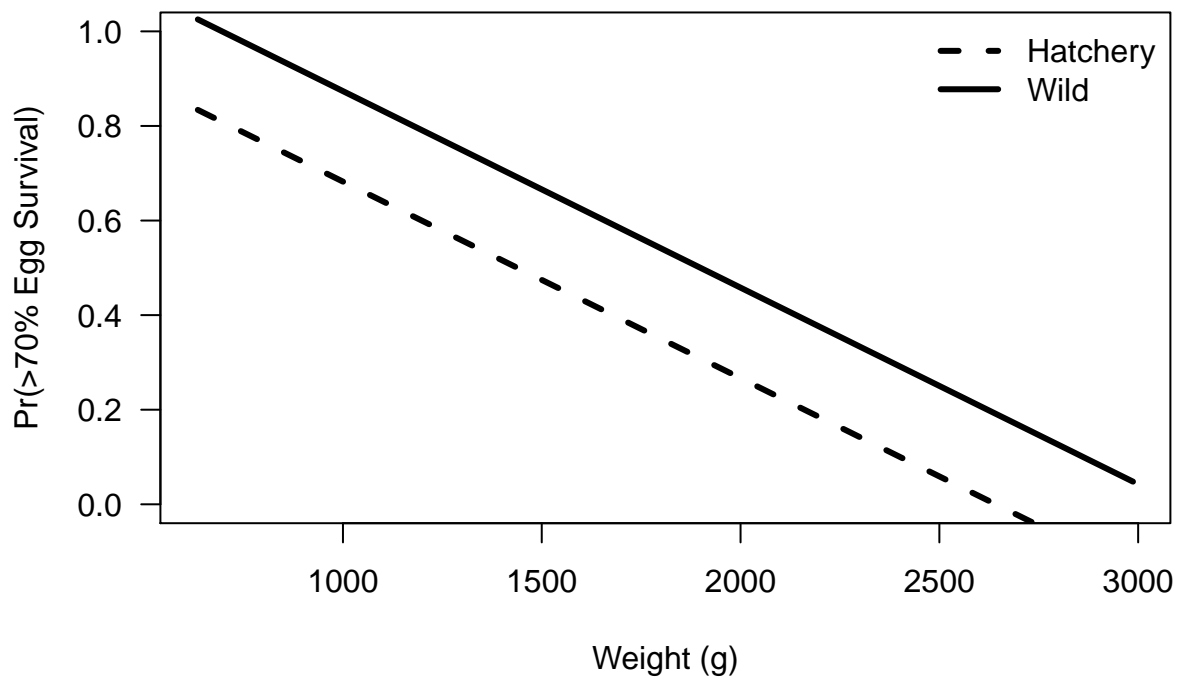

Look for an interaction (all the code is the same except use `glm(binary ~ type * weight)` instead of `glm(binary ~ type + weight)` and change everything to `fit4` instead of `fit3`).
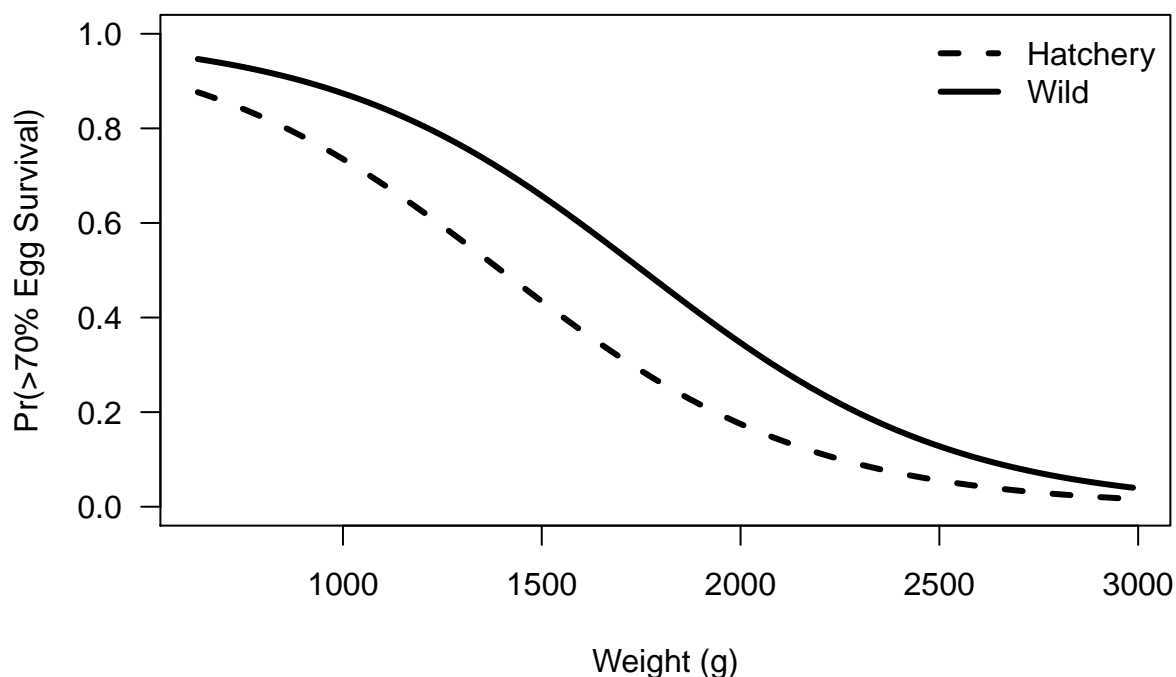
You may have noticed that you just did the same analysis with `binary` as the response instead of `fecund`. Perform an AIC analysis to determine which model is likely to be best for prediction:

```
AIC(fit1, fit2, fit3, fit4)
```

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

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.

3.8 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 below, along with the suffixes of the functions in R that correspond to each distribution[5].

In R, there are four different ways to use each of this distribution functions (each as 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.

---

[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.

Type	Distribution	Common Uses
Continuous	Normal	Models the relative frequency of outcomes that are symmetric around a mean, can be negativ
	Lognormal	Models the relative frequency of outcomes that are normally-distributed on the log-scale
	Uniform	Models values that are between two endpoints and that all occur with the same frequency
	Beta	Models values that are between 0 and 1
Discrete	Binomial	Models the number of successes from a given number of trials when there are only two possib
	Multinomial	The same as the binomial distribution, but when there are more than two possible outcomes
	Poisson	Used for count data in cases where the variance and mean are roughly equal

- **The cumulative density function** (`p-`): the sum of the probability densities for all random variables below some value.

- **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 $x \sim N(\mu = 500, \sigma = 50)$ (x is a normal random variable with mean equal to 500 and standard deviation equal to 50). Here is the usage of each of the distribution functions and a plot illustrating them:

```r
# 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

# 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 = 100, 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)
```

3.9   Bonus Topic: Non-linear Regression

You fitted linear and logistic regression models in Sections 3.2 and 3.7, however, R allows you to fit non-linear regression models as well.

First, aquire the `feeding.csv` data set from the GitHub repo and place it in your working directory. Read the data into R:

```
##       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
```

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

These are hypothetical data from an experiment in which you were interested in quantifying the functional feeding response[6] of a fish predator on zooplankton in an aquarium. You experimentally manipulated the prey density (`dat$prey`) and counted how many prey items were consumed (`dat$cons`).

Plot the data:

```
plot(cons ~ prey, data = dat)
```

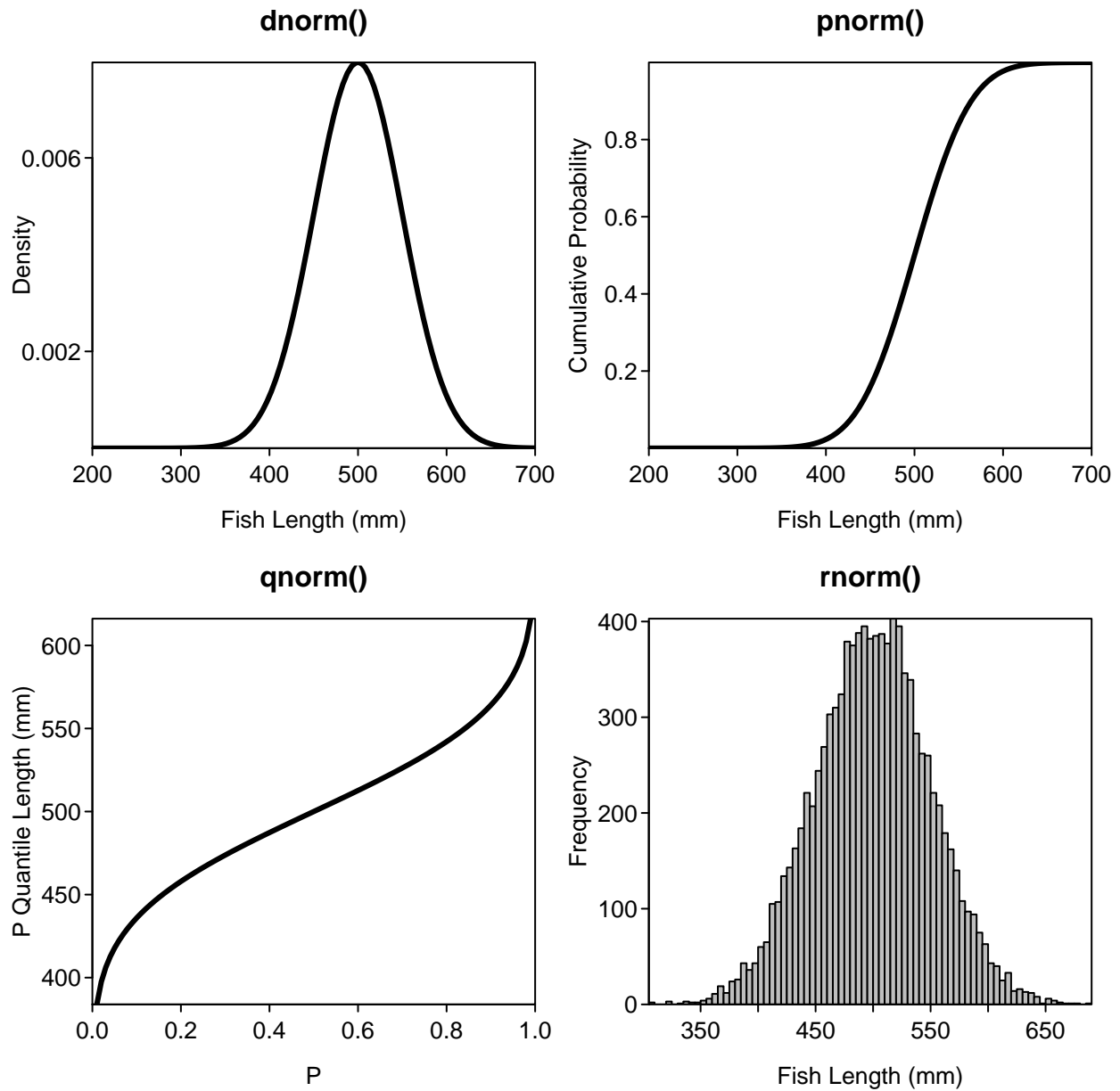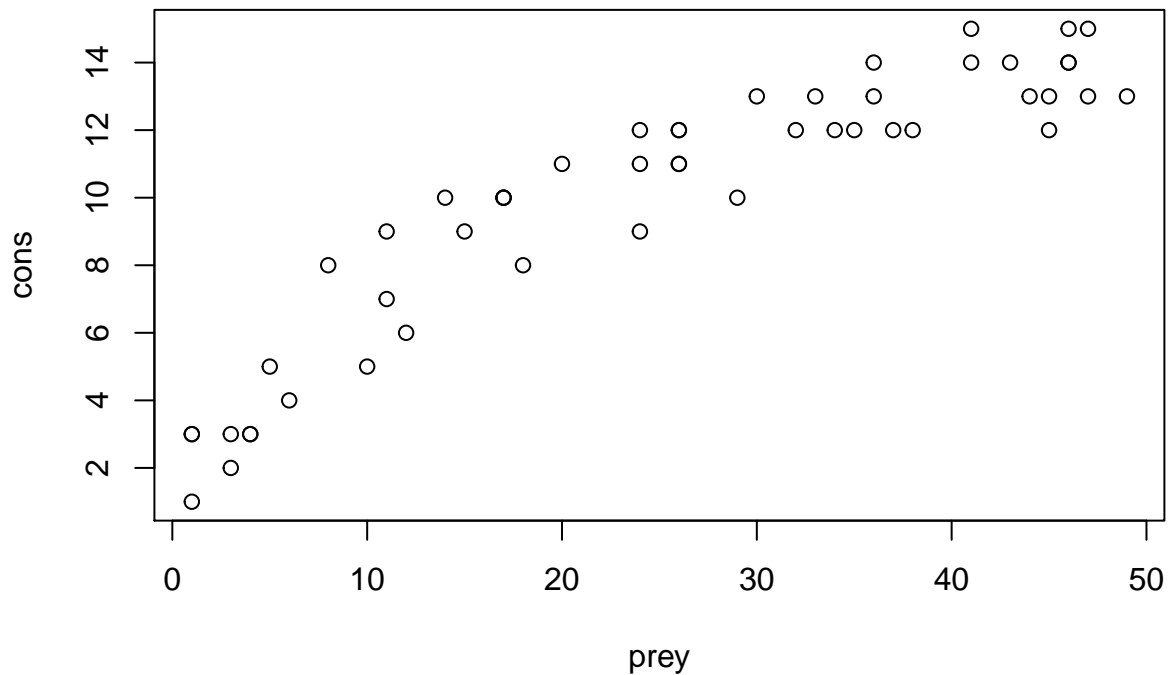[6]A functional response is the number of prey consumed by a predator at various prey densities

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

You can see a distinct non-linearity to the relationship. The Holling Type II functional response[7] has this functional form:

$$y_i = \frac{ax_i}{1 + ahx_i}$$

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

You can fit this model in R using the `nls` function, behaves very similarly to the `lm` function.

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

You can obtain similar output as from `lm` using the `summary`, `coef`, and `predict`. 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)
```

---

[7]This function rises quickly at low prey densities, but saturates at high densities
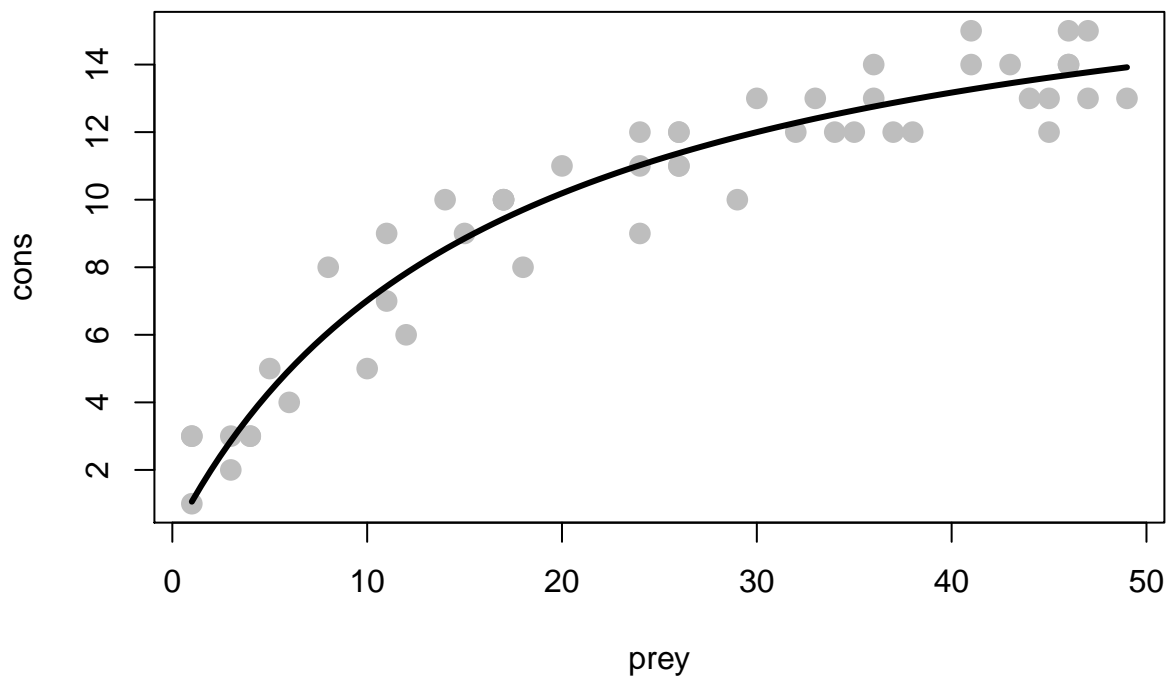
EXERCISE 3

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[8].

---

[8]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

Simulation and Randomization

Chapter 5

Large Data Manipulation

Chapter 6

Mapping and Spatial Analysis

Datasets