An introduction to basic fisheries analysis with R

Contents

Chapter 1

Objective

> The purpose of this guidebook is to provide an introduction to using the powerful programming language R to conduct analyses commonly used for fisheries management. The guidebook is designed to help you get quickly started in R with some basic analyses and visualizations, but it is only and introduction and is not exhaustive. We do however point to some helpful resources for learning more.

R is a free programming language/software environment that allows users to analyze, model, and vizualize large data sets in much more powerful and complex ways than traditional spreadsheet programs like Excel or Google Sheets. Best of all, R is open source, meaning that it is freely available from the Comprehensive R Archive Network (CRAN) and anyone can contribute to making R better. In fact, numerous R packages (more on these later) are specifically designed for conducting analyses related to fisheries management. RStudio is the powerful graphical interface that allows users to manage their code, data, and files all in one convenient program. If you are interested in learning more about data analysis with R, the free online book called R for Data Science is an excellent resource.

Chapter 2

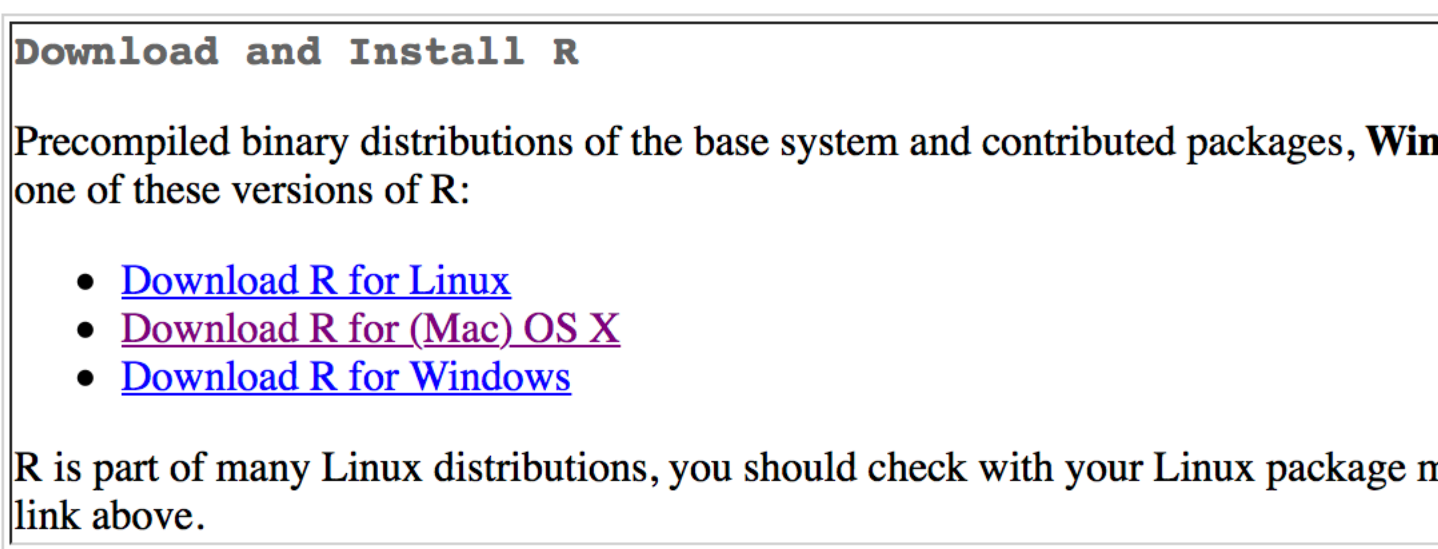
Installation

Though R/RStudio may seem intimidating, it is actually quite straight forward to set up and, after learning a few basics, you can start running analyses and writing your own in no time. **The objective of this guide is to provide an introduction to R/RStudio basics so that interested resource managers without programming experience can start leveraging R for their management decisions.**

R and RStudio are **separate** programs and that need to be installed and updated individually. If you do not keep both relatively up-to-date you will likely run into problems.

2.1 R

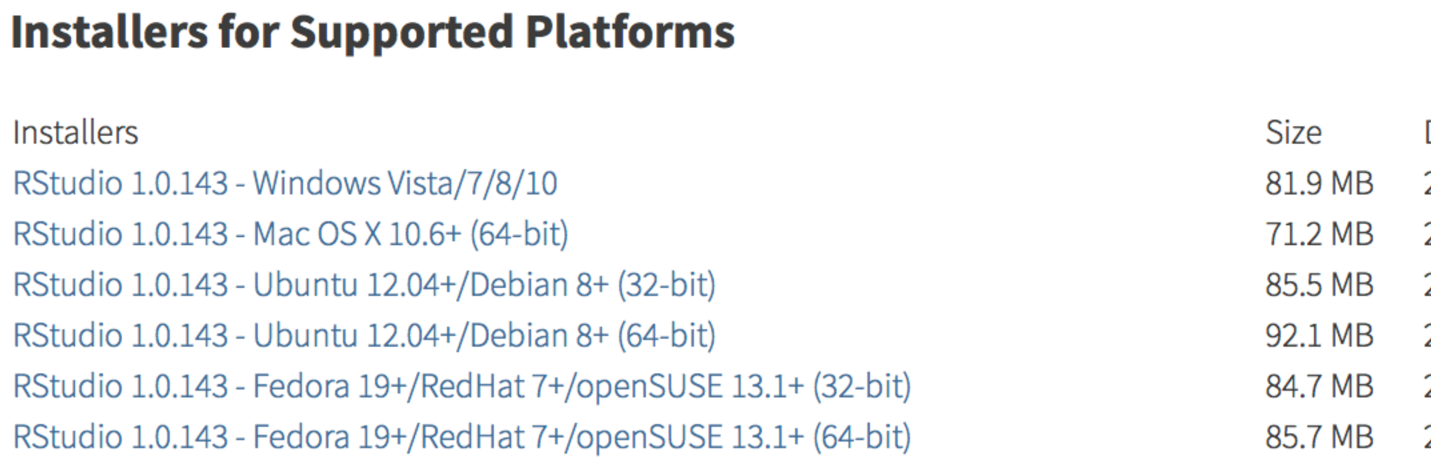
To install R, go to the R webpage and follow the link to your operating system of choice (Linux, Max OS X, Windows). Then, click on the most recent `.pkg` file to download it. Follow the instructions to complete the installation process.



2.2 RStudio

After installing R, visit the RStudio Products site and click the **DOWNLOAD RSTUDIO DESKTOP** button located partway down the page.

Next, scroll to the bottom and click on the link under **Installers** that again corresponds to your operating system of choice.



Installers for Supported Platforms

Installers	Size
RStudio 1.0.143 - Windows Vista/7/8/10	81.9 MB
RStudio 1.0.143 - Mac OS X 10.6+ (64-bit)	71.2 MB
RStudio 1.0.143 - Ubuntu 12.04+/Debian 8+ (32-bit)	85.5 MB
RStudio 1.0.143 - Ubuntu 12.04+/Debian 8+ (64-bit)	92.1 MB
RStudio 1.0.143 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	84.7 MB
RStudio 1.0.143 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	85.7 MB

Save the file on your desktop. Once it finishes downloading, open the file and follow the instructions to complete the installation process. You may then delete the file.

Congratulations! You successfully completed the installation process and are one step closer to using R and RStudio for analysis!

2.3 Helpful Resources

- Installing R and RStudio by Jenny Bryan

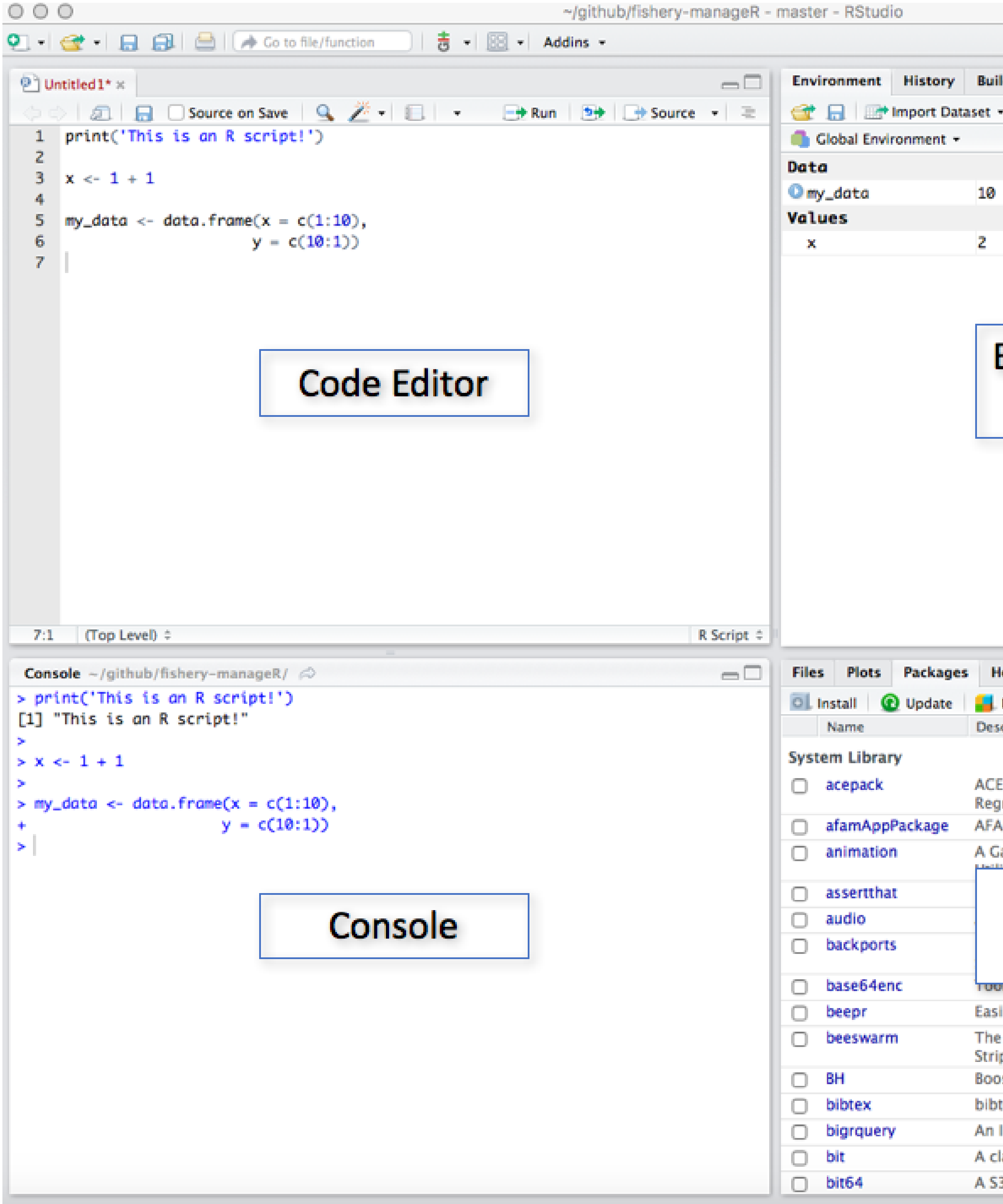
Chapter 3

Getting Started with RStudio

Rstudio is what's referred to as an Integrated Development Environment (IDE) for the R programming language. It provides a single interface for an R user to manage all aspects of an analysis (write code, manage and plot data. see outputs, get help, etc.).

3.1 RStudio Interface

There are four main panels in RStudio (Figure 1).


~/github/fishery-manageR - master - RStudio
Go to file/function
Addins
Untitled1*
Source on Save
Run
Source
print('This is an R script!')
x <- 1 + 1
my_data <- data.frame(x = c(1:10),
y = c(10:1))
Code Editor
7:1
(Top Level)
R Script
Environment
History
Import Dataset
Global Environment
Data
my_data
Values
x
2
Console ~/github/fishery-manageR/
> print('This is an R script!')
[1] "This is an R script!"
> x <- 1 + 1
> my_data <- data.frame(x = c(1:10),
+ y = c(10:1))
Console
Files
Plots
Packages
Install
Update
Name
System Library
acepack
afamAppPackage
animation
assertthat
audio
backports
base64enc
beepr
beeswarm
BH
bibtex
bigrquery
bit
bit64

- **Code Editor** - This is where you write the code for your analysis. Each tab represents a different R *script* file (e.g. `snapper_analysis.R`)
- **Console** - This is where R prints the output of your code when it's run. You can also write code directly in the console after the `>` symbol
- **Environment/History** - This panel generally has the following two tabs:
  - **Environment** - Displays all your data, variables, and user-defined functions. These are created by the user either in the code editor or directly in the console.
  - **History** - A list of your command history
- **Files/Packages/Help/Viewer** - This panel contains numerous helpful panels:
  - **Files** - The list of all files contained in your current *working directory*. You can also navigate to different folders on your computer. This is where you can click on different R scripts to open them in the code editor.
  - **Plots** - When you produce plots with your code they will be displayed here
  - **Packages** - The list of packages (groups of functions) currently installed on your computer. You can install new packages or update existing packages from this tab by clicking `Install` or `Update`.
  - **Help** - Where you can search the R documentation to get help using the different R functions. This is a very useful feature of RStudio! You can also get help for a function by typing `?` followed by the function name in the console (e.g. `?data.frame()`).

3.2 Working Directory

The **working directory** is an important concept in R (and programming in general) and refers to the current directory (folder) that you are working in. Basically, R requires that you tell it where in your computer's file system it should start looking for files from. This is important because the code used to load data and save results and plots will differ depending on your current working directory.

As an example, let's imagine you are working on an analysis of coral reef fisheries and you have a folder on your Desktop called `reef_fish`. Inside this `reef_fish` directory is the file `reef_fish_data.csv` that you want to analyze. Open RStudio and click `Set Working Directory...` under the `Session` menu in the toolbar. This asks you to specify the folder on your computer that R should consider to be the working directory.

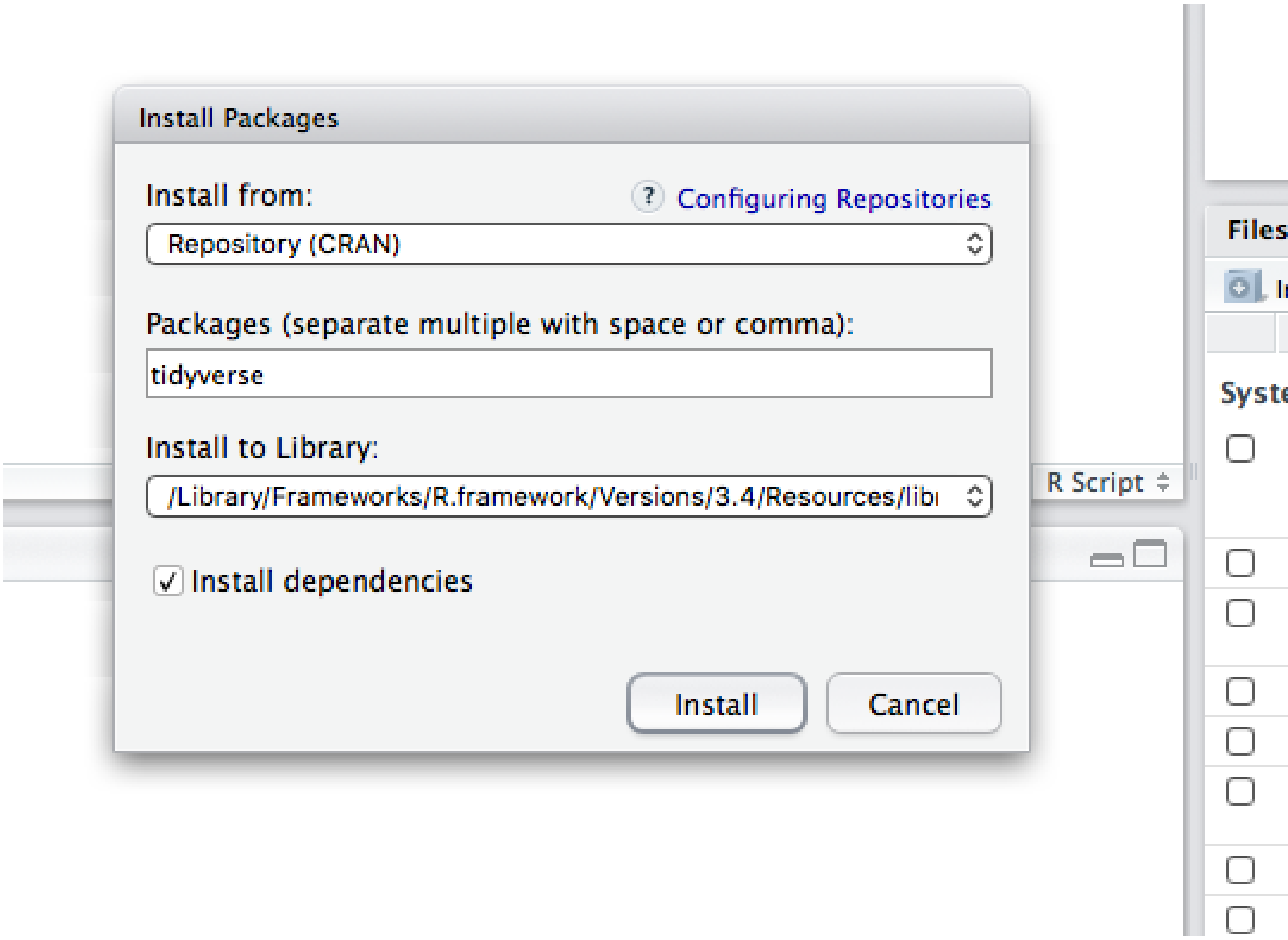
3.2.1 Pathnames (Path)

Now that the working directory is set, you can load your reef fish data into RStudio by specifying the appropriate **pathname** to the file. In this case, the pathname is simply `reef_fish_data.csv` since the file is in the working directory and the data could be loaded with `read.csv(file = "reef_fish_data.csv")`. If, however, `reef_fish_data.csv` was actually stored in a subfolder called `data` the previous `read.csv()` command will not work because `reef_fish_data.csv` is not in the working directory. In this case, you can either tell R where the file is using the **absolute**, complete path (e.g. `/Users/You/Desktop/reef_fish/data/reef_fish_data.csv`), or with the path **relative** to the working directory (e.g. `data/reef_fish_data.csv`).

3.3 Packages

Packages are groups of functions that are designed to excel at certain tasks (making plots, standardizing dates, reading/writing large data files, etc.). Many useful packages come standard with R when you download it, however, many more are available online.

To install a new package, click on the "Install" button located under the "Packages" tab in RStudio. This will open a pop-up where you can search for and install R packages hosted on CRAN. Alternatively, if you know the name of the package you want to install, you can run `install.packages('package_name')`.



Once installed, the packages you need for an analysis are loaded by the `library('package_name')` function.

The following packages are commonly used by UCSB for fisheries analyses:

- **tidyverse** - Contains numerous seperate packages for loading and writing data files (**readr**), data processing (**dplyr** & **tidyr**), plotting (**ggplot2**), and functional programming (**purrr**)
- **readxl** - Reads and writes data from/to Excel workbooks, including workbooks with multiple worksheets
- **lubridate** - Helps R handle dates in an efficient and easy-to-understand manner
- **sf**, **rgeos**, **rgdal** - Key packages for spatial analyses similar to those done with ArcGIS
- **rmarkdown** - Uses R code to author reproducible reports, presentations, and websites
- **shiny** - Creates web applications using R

3.4 Helpful Resources

- Jenny Bryan's Guide to Installing R and RStudio

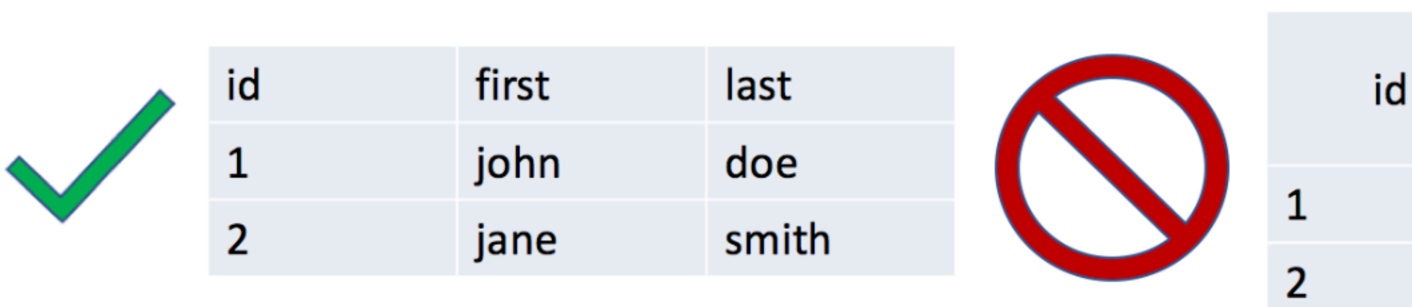
Chapter 4

Data Entry & Management

The careful entry, documentation, and management of data is essential to any data-related project. Being strategic about this process will keep the project organized, protect against data loss, and facilitate analysis and data sharing.

4.1 Best Practices

1. **Using "flat" files and an open data format**
   Raw data should be entered as a "flat" table and saved using an open data format, such as .csv (comma separated values). "Flat" data files are tables where the first "header" row contains the variables in the data set and there is no internal hierarchy to the data. Nested columns and rows make analysis of the raw data outside of the original file very difficult.



2. **Organize data in a tidy format with unique records in rows, not columns**
   Data records should be stored in rows (long-format) instead of columns (wide format). This allows analysis within rows rather than across columns. This "tidy data" format means that each **variable** is saved in its own **column** and each **observation** is saved in its own **row**.

first	last	month	year	catch
john	doe	11	2016	150
john	doe	12	2016	100
john	doe	1	2017	125

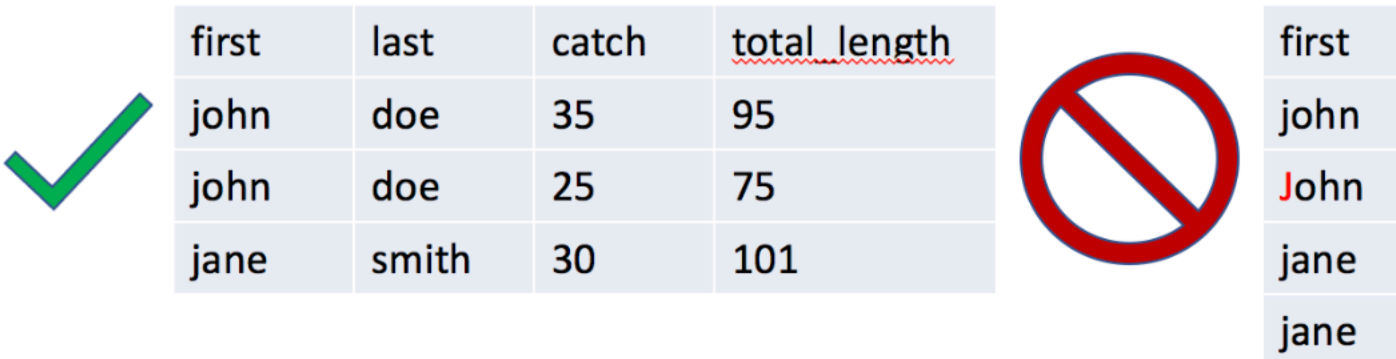
first
john

3. **Describe data in a metadata file**
   Your raw **flat data files should only include data, no comments**. Rather than using complicated spreadsheets, create a metadata document, often called "README", that includes (at a minimum) what data you are collecting, how and when the data were collected, where the data is stored, and who owns the data. This file should also include a "data dictionary" that describes each variable and associated unit in the data file (see example below). Be as specific as possible; for example, the description for fish length variables should include whether it is fork length, total length, or standard length.

Column	Description	Unit
id	Unique ID for fisher	number
first	Fisher's first name	text
last	Fisher's last name	text
catch	The weight of the catch (kg)	kg
biomass	The biomass of fish (kg/ha)	kg/ha
total_length	The total length of the fish (cm)	cm

4. **Use clear and concise descriptive names for data files and variable names**
   File names are the easiest way to explain the contents of a data file. Capturing the place, time, and content of the data, even in an abbreviated fashion, can be extremely useful. For example, consider naming a fish catch monitoring file "muni_fishcatch_month_year.csv", replacing "muni", "month", and "year" with the appropriate values. Similarly, each column in your data should contain a unique variable and be given a clear but concise name that uses letters, numbers, dashes, dots, or underscores. Lastly, always use plain ASCII text, as certain marks (e.g., accents) or characters (e.g., Chinese or Japanese) are not widely supported. File and variable names should NOT be overly long or contain spaces or special characters (e.g. *&$%@/)
   - **YES:** filename: *muni_fishcatch_month_year.csv* | variables: *year; first_name; last.name; Total-Length*
   - **NO:** filename: *November 2007.csv* | variables: *Start Year; First Name; Total weight (kg); $ Value*
5. **Always use consistent formats for data values and (if necessary) put units in a separate column**
   When entering data, **do not mix text and numeric responses**, or include both text and numbers

in the same response. Periods are okay to include for numeric responses but **avoid commas** (commas indicate a new value in a .csv file). For text values, such as a person's name or location, take care not to change capitalization, spelling, spacing, etc. (e.g. John, john, jon) as this will generate confusion. **Consider using identification codes for variables with many possible categories (e.g., local species name, gear type)**. Units should always be in their own column or absent entirely but explained in the metadata file. Also, **do not use color coding**, it cannot be interpreted by other software (data in red below are just to demonstrate improper data entries).



first	last	catch	total_length
john	doe	35	95
john	doe	25	75
jane	smith	30	101

first
john
John
jane
jane

6. **Use standardized formats for dates** When reporting full dates, use standardized formats since date representations vary between the United States and the rest of the world. For example, 01-09-17 will likely be interpreted as January 9th, 2017 in the U.S. but September 1st, 2017 or September 17th, 2001 in other countries. Therefore, always record dates using the international standard of YYYY-MM-DD as prescribed by the International Organization of Standards (ISO) standard ISO 8601 (2004). It is also generally good to have separate columns for **month** and **year** to facilitate analyses that are only interested in certain months or years (e.g. what are the average landings in March?).

7. **Always store an uncorrected original version of the data file and BACK UP YOUR DATA!!!**
   When you make changes or corrections to the original data file you could easily make a mistake. To avoid compromising your original raw data, always store an unadjusted copy of the data file and **do not make any changes or adjustments to this copy** (make it "read-only" if possible). Make a duplicate file if corrections or adjustments are required and be sure to describe what changes were made in the metadata file. Lastly, **ALWAYS BACK UP YOUR DATA** by keeping at least **three** copies of the file in different locations (e.g., desktop, external hard drive, the cloud).

8. **Be consistent with blank cells by using NA** It is common for cells to be blank in a spreadsheet. However, there are many different ways to signify a blank cell, some of which are more difficult to work with. We recommend always using `NA` in any blank cell, rather than simply leaving the cell blank, putting in spaces, or other options like `NaN` or `-999`. This will allow R to quickly identify and deal with empty cells.

9. **For storing data, always use CSV files instead of XLS or XLSX files** There are also several different file formats for storing spreadsheet data. XLS and XLSX are two very commonly used formats, and are the standard Microsoft Excel spreadsheet format (XLSX is the newest version). However, many programming languages have difficulty ingesting Microsoft Excel files. These files also require people using them to have a copy of Microsoft Excel, which not everyone has and can be expensive. We therefore recommend using the CSV file format for storing data. CSV stands for "Comma-Separated Values" and describes how data are stored in the file. If you have a Microsoft Excel file, you can save it as a CSV by clicking `File -> Save As` and selecting the CSV option. CSVs can be read directly into R, and used and manipulated by anyone with a computer.

4.2 Helpful Resources

- https://www.nceas.ucsb.edu/files/news/ESAdatamng09.pdf
- http://ucsd.libguides.com/c.php?g=90957&p=585435

Chapter 5

R as a Programming Language

You may or may not have used other programming languages before coming to R. Either way, R has several distinctive features which are worth noting.

5.1 Data frames

One of R's greatest strengths is in manipulating data. One of the primary structures for storing data in R is called a Data Frame. Much of your work in R will be working with and manipulating data frames. Data frames are made up of rows and columns. The top row is a header and describes the contents of each column. Each row represents an individual data row or observation. Rows can also have names. Each row contains multiple cells which contain the content of the data.

Let's look at a data frame that is included in R as an example. This data frame is called `mtcars` and contains some data about common car models. We can look at this data frame using the `head` function, which previews the first few rows. You can also use the functions `colnames` or `rownames` to get the column or row names of a data frame, respecitvely.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

The columns in data frames can contain different types of information. In this particular data frame, all columns contain numbers, as denoted by the indication `<dbl>`, a type of number that allows numbers after the decimal point. Columns could also be integers (which don't allow numbers after the decimal point), booleans or logicals (`TRUE` or `FALSE`), characters (these contain text), or other types of data.

These days, you may also see the word `tibble`, which is a modern version of the R data frame and is being used more and more widely. Tibbles generally function much like data frames, but do away with some frustrating features.

5.2 Vectors

Another very common data type is the vector, which stores 1-dimensional information, such as a list of numbers or characters. Vectors are built using the `c` function. Below are two examples:

```
myNumericVector <- c(1, 2, 3, 4)

myCharacterVector <- c("A", "B", "C", "D")
```

5.3 Functions

R is a "functional" programming language, which means it gets much of its power by relying on the concept of functions. Functions are small chunks of code that can do a certain task. They require a certain number of inputs, and provide a certain number of outputs. They allow for common tasks to be performed easily and reproducibly.

R contains many built-in functions, including several for helpful statistics, as shown below. In these examples, there are also some helpful comments to tell you what each line is doing in the code blocks below. Comments start with the `#` operation, and are not evaluated by R - they are simply there to document the code.

```
# The sum function takes a vector of numbers as an input, calculates the sum of those numbers, and prod
sum(c(1, 2, 3))
```

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

```
# The mean function takes a vector of numbers as an input, calculates the mean of those numbers, and pr
mean(c(1 ,2, 3))
```

```
## [1] 2
```

```
# The median function takes a vector of numbers as an input, calculates the median of those numbers, an
median(c(1 ,2, 3))
```

```
## [1] 2
```

You can even define your own functions, which can be an incredibly powerful way to save time when doing a task you anticipate needing to do many times. The example below shows how you would write a function that takes in two numbers as an input, manipulates the numbers, and then provides a single number as an output. Once you have defined a function, you can use it again later on.

```
# Define the function
myFunction <- function(x,y){
  z <- (x + 2 * y) / x
  return(z)
}
# Test the function
myFunction(3,4)
```

```
## [1] 3.666667
```

5.4 Tips on coding and style

It is helpful to code in a consistent manner. This will not only make your code readble by others, but will even be helpful for you as you revisit code you have previously written. Using consistent code stying also makes it much easier to collaborate with others. We highly recommend following Google's style guidelines for R.

5.5 Helpful resources

- More information on data frames
- More information on tibbles, the modern version of data frames
- Functions
- Google's style guidelines

Chapter 6

Packages

The first step in any analysis is to load the packages you will need for your analysis. Loading packages allows you to use powerful functions not included in "base" R.

For this analysis, you will use the tidyverse package, which actually loads a group of useful packages including `tidyr`, `dplyr`, `readr`, and `ggplot2`. `tidyr` and `dplyr` are very handy packages for manipulating data, `readr` is great for loading in data such as csv files, while `ggplot2` is one of the best packages from plotting data. `lubridate` is another very handy package for dealing with dates and times.

If you haven't done so yet, follow the instructions in the Installation section to load the `tidyverse` and `lubridate` packages which will be used in the example. You can then type the following lines into the console to load them into your current R session.

```r
library(tidyverse)
library(lubridate)
```

The next few sections of this quick introduction to R will walk you through how to calculate some basic fisheries statistics and plot the results.

Chapter 7

Loading Fisheries Data

Next, you'll load the data you'll be using for the analysis using the `read_csv` function. First, create a new folder in your working directory called "_data". Next, download the following two files onto your computer. Right click on each link, and save it in your"_data" folder.

Right click and save-link-as to download landings data

Right click and save-link-as to download life history data

Next, you'll load these two csv files into R. First load the landings data which includes catch, effort, and length measurements and store these data to a data frame called `landings_data`. Next, load the life history parameter data file and store these data to a data frame called `life_history_parameters`.

```
landings_data <- read_csv("_data/sample_landings_data.csv")

life_history_parameters <- read_csv("_data/life_history_parameters.csv")
```

Let's take a quick look at what's in each of these data frames. We can use the `head` function to get a quick summary.

```
head(landings_data)
```

```
## # A tibble: 6 × 8
##    Year    Date Trip_ID Effort_Hours  Gear       Species Length_cm
##   <int>   <chr>   <int>        <int> <chr>         <chr>     <dbl>
## 1  2003 4/30/03       1           10  Trap Caesio cuning        36
## 2  2003 4/30/03       1           10  Trap Caesio cuning        29
## 3  2003 4/30/03       1           10  Trap Caesio cuning        34
## 4  2003 4/30/03       1           10  Trap Caesio cuning        36
## 5  2003 4/30/03       1           10  Trap Caesio cuning        34
## 6  2003 4/30/03       1           10  Trap Caesio cuning        28
## # ... with 1 more variables: Weight_g <dbl>
```

The `landings_data` data frame is from a fishery-dependent landing site survey. The species included in this data set is *Caesio cuning*, a yellowtail fusilier. You can see the data frame has eight columns, with each row being an individual fish catch measurement. The columns include the year and date when the measurement was collected, the fishing trip ID, how many hours were fished for each trip, what gear was used, the species, the length of the fish, and the weight of the fish. We'll be able to use these data to create length-frequency histograms that describe the size structure of the population, as well as trends in catch and CPUE.

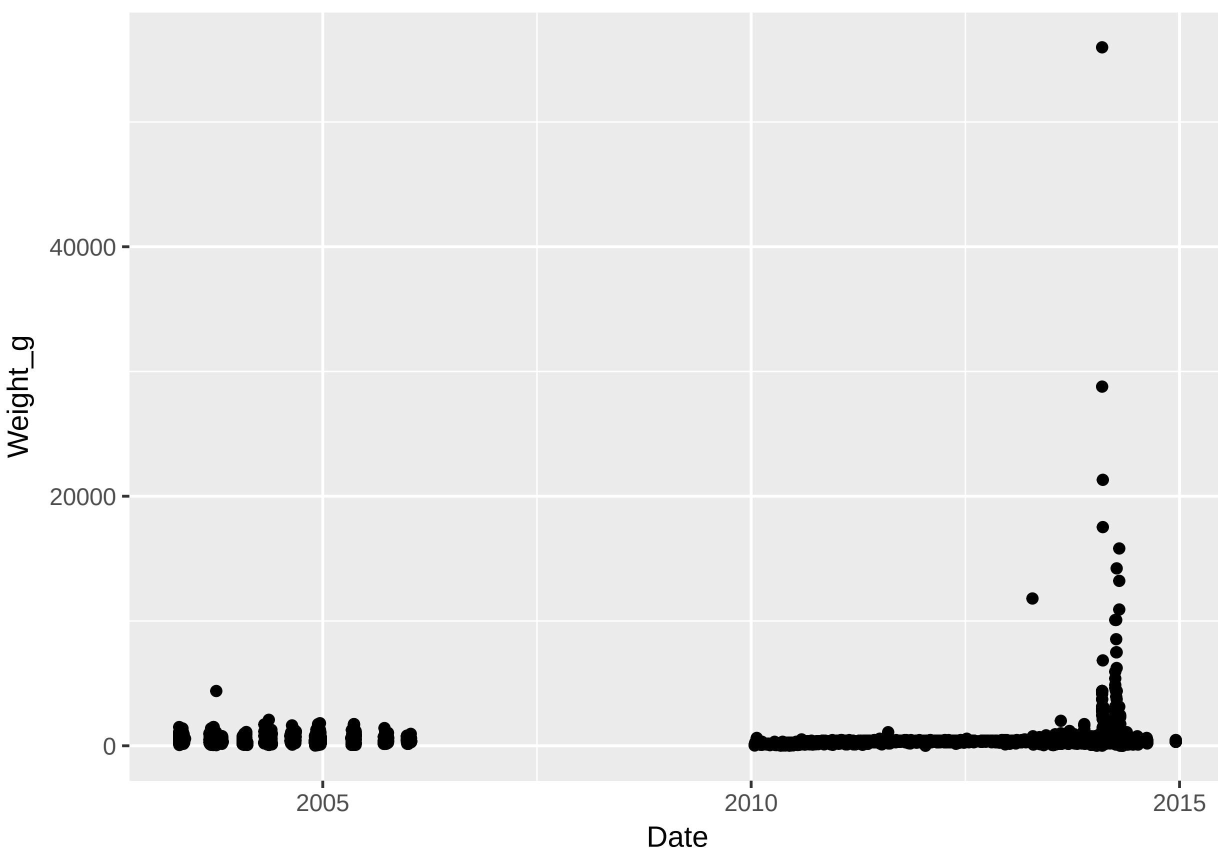
One thing we should do is format the year and date columns to ensure that R knows these represent years and dates as we humans think of them. We can use the `mdy` and `year` functions from the lubridate package to do this, combined with the `mutate` function from the dplyr package. We start by taking the landings data frame we loaded into R, and working through a series of "pipes", designated by the `%>%` operation, which progressively analyzes the data from one step to the next. Essentially, the output of one line is fed into the input of the next line.

```
# Start with the landings data frame
landings_data <- landings_data %>%
  # Turn the date column into a date format that R recognizes
  mutate(Date = mdy(Date)) %>%
  # Create a year column using a year format R recognizes
  mutate(Year = year(Date))
```

One of the first things you should always do is quickly plot your raw data using `ggplot`. This can help you familiarize yourself with what's contained in the data set, as well as look for any potential outliers or errors. We will come back to a more detailed description of plots and visualizations later, but below is some quick code to look at things. When using `ggplot`, first start with your data frame and initialize the ggplot by specifying the plot's *aesthetics* (variables) using `aes()`. Then use the `+` operation to add at least one *geometry* (type of plot, such as a scatter plot) and any additional features to the plot. To learn more about `ggplot`, the Data Visualization with ggplot2 Cheat Sheet is a very helpful resource, as is this ggplot cookbook.

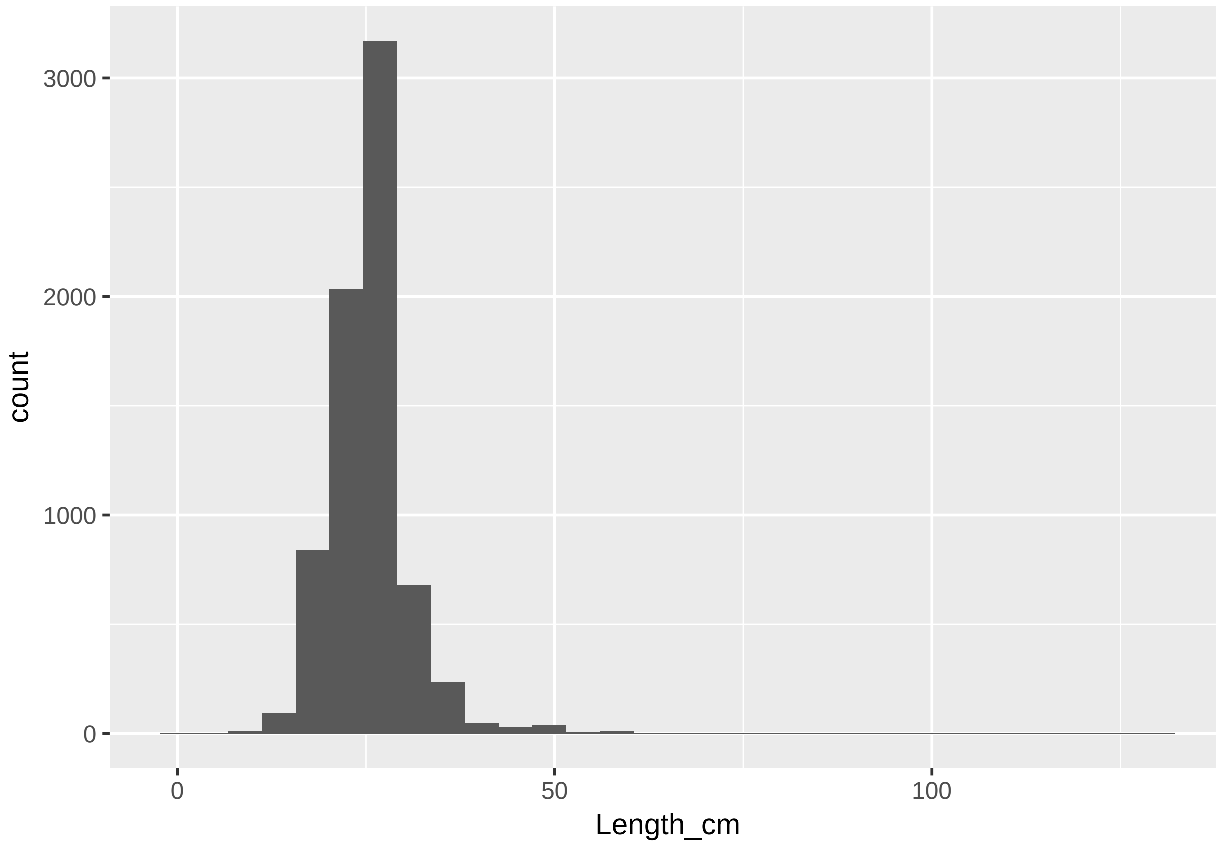
The first figure just shows a point for every weight measurement over time.

```
# Start with the landings data frame
landings_data %>%
  # Initialize a ggplot of weight measurements versus year
  ggplot(aes(x=Date,y=Weight_g)) +
  # Tell ggplot that the plot type should be a scatter plot
  geom_point()
```



The second figure shows a histogram of the length data.

```
# Start with the landings data frame
landings_data %>%
  # Initialize ggplot of data using the length column
  ggplot(aes(Length_cm)) +
  # Tell ggplot that the plot type should be a histogram
  geom_histogram()
```



Let's next look at the `life_history_parameters` data frame.

```
head(life_history_parameters)
```

```
## # A tibble: 1 × 11
##         Species              Common L_inf     k    t0         M     Wa
##           <chr>               <chr> <dbl> <dbl> <dbl>     <dbl>  <dbl>
## 1 Caesio cuning Yellowtail fusilier  62.2  0.32 -0.42 0.8915981 0.0208
## # ... with 4 more variables: Wb <dbl>, m50 <dbl>, m95 <dbl>, maxAge <int>
```

The `life_history_parameters` data frame includes several important life history parameters that were found doing a literature review of this species. *L_infinity*, *k*, and *t0* describe the age-to-length relationship using the von Bertalanffy growth equation. *M* represents natural mortality. *Wa* and *Wb* describe the length-to-weight relationship. *m50* and *m95* describe the lengths at which 50% and 95% of the fish are mature, respectively. *maxAge* is the maximum age of the fish.

Chapter 8

Basic Fisheries Statistics

8.1 Calculating Landings

One of the first analyses you may be interested in is calculating annual landings in the fishery. To calculate annual landings, take your `landings_data` data frame, add a column for weight of individual fish in kilograms by using the `mutate` function, group the data by year by using the `group_by` function, and then summarize the data for each year by summing the total weight of all fish caught in each year using the `summarize` and `sum` functions.

```
# Start with the landings data frame
annual_landings <- landings_data %>%
  # Add colomn for kilograms by dividing gram column by 1000
  mutate(Weight_kg = Weight_g / 1000) %>%
  # Group the data by year
  group_by(Year) %>%
  # Next, summarize the total annual landings per year
  summarize(Annual_Landings_kg = sum(Weight_kg,na.rm=TRUE))

## Display a table of the annual landings data
annual_landings
```

```
## # A tibble: 9 × 2
##    Year Annual_Landings_kg
##   <dbl>              <dbl>
## 1  2003          311.04137
## 2  2004          565.49423
## 3  2005          163.24191
## 4  2006           37.11914
## 5  2010          131.84178
## 6  2011          156.77825
## 7  2012          101.53198
## 8  2013          579.52008
## 9  2014         1278.52662
```

Note the use of `na.rm = TRUE` in the code above. This is an important **argument** of many R functions (`sum()` in this case) and it tells R what to do with `NA` values in your data. Here, we are telling R to first remove `NA` values before calculating the sum of the `Weight_kg` variable. By default, many functions will return `NA` if **any** value is `NA`, which is often not desirable.

You may be interested in looking at landings across different gear types. Here, we now group the data frame by both the year and the gear type in order to summarize the total landings by year and by gear.

```
# Start with the landings data frame
annual_gear_landings <- landings_data %>%
  # Add colomn for kilograms by dividing gram column by 1000
  mutate(Weight_kg = Weight_g / 1000) %>%
  # Group the data by year and gear type
  group_by(Year,Gear) %>%
  # Next, summarize the total annual landings per year and gear type
  summarize(Annual_Landings_kg = sum(Weight_kg,na.rm=TRUE))

## Display a table of the annual landings data by gear type
annual_gear_landings
```

```
## Source: local data frame [39 x 3]
## Groups: Year [?]
##
##     Year     Gear Annual_Landings_kg
##    <dbl>    <chr>              <dbl>
## 1   2003  Gillnet          13.413401
## 2   2003 Handline           3.188572
## 3   2003  Muroami         247.879049
## 4   2003     Trap          46.560347
## 5   2004  Gillnet           4.189301
## 6   2004 Handline          57.705893
## 7   2004  Muroami         370.866460
## 8   2004 Speargun           9.476406
## 9   2004     Trap         118.869619
## 10  2004 Trolling           4.386547
## # ... with 29 more rows
```

8.2 Calculating Catch-per-Unit-Effort (CPUE)

You may also be interested in calculating catch-per-unit-effort (CPUE). CPUE is calculated by dividing the catch of each fishing trip by the number of hours fished during that trip. This gives CPUE in units of kilograms per hour. The median for every year is then calculated in order to remove outliers - some fishers are much more efficient than others.

```
# Start with the landings data frame
cpue_data <- landings_data %>%
  # Add colomn for kilograms by dividing gram column by 1000
  mutate(Weight_kg = Weight_g / 1000) %>%
  # Group by year and Trip ID so that you can calculate CPUE for every trip in every year
  group_by(Year,Trip_ID) %>%
  # For each year and trip ID, calculate the CPUE for each trip by dividing the sum of the catch, conve
  summarize(Trip_CPUE = sum(Weight_kg) / mean(Effort_Hours)) %>%
  # Next, just group by year so we can calculate median CPUE for each year across all trips in the year
  group_by(Year) %>%
  # Calculate median CPUE for each year
  summarize(Median_CPUE_kg_hour = median(Trip_CPUE))

# Display a table of the CPUE data
cpue_data
```

```
## # A tibble: 9 × 2
##    Year Median_CPUE_kg_hour
##   <dbl>               <dbl>
## 1  2003          0.31834277
## 2  2004          0.26070028
## 3  2005          0.40145105
## 4  2006          0.44029501
## 5  2010          0.01742840
## 6  2011          0.03123217
## 7  2012          0.03123217
## 8  2013          0.19638408
## 9  2014          0.88216281
```

8.3 Calculating Percent Mature

You may also wish to analyze your length data. One analysis would be to determine the percentage of mature fish in the catch in every year of the data frame. To do this, we add a column to the data frame using the `mutate` function that represents whether each fish is mature or not (represented by a `TRUE` or `FALSE`), group the data frame by year, and then summarize for each year the percentage of mature fish out of the total number of sampled fish. To determine if a fish is mature or not, it is compared against *m95*, the length at which 95% of fish are mature. The value for `m95` is taken from the `life_history_parameters` data frame and is referenced using a `$` and the code `life_history_parameters$m95`.

```
# Start with the landings data frame
landings_data %>%
  # Add a column to the data that indicates whether each length measurement is from a mature or immatur
  mutate(Mature = Length_cm > life_history_parameters$m95) %>%
  # Group by year so we can see the percent mature for every year
  group_by(Year) %>%
  # The percentage mature is equal to the  number of mature fish divided by the total number of fish an
  summarize(Percent_Mature = sum(Mature) / n() * 100)
```

```
## # A tibble: 9 × 2
##    Year Percent_Mature
##   <dbl>          <dbl>
## 1  2003       98.57595
## 2  2004       98.62306
## 3  2005       97.73371
## 4  2006      100.00000
## 5  2010       91.80556
## 6  2011       99.77629
## 7  2012       99.65398
## 8  2013       99.46164
## 9  2014       99.55709
```

Over 90% of the fish are mature throughout the time series, which is a great sign!

8.4 Helpful Resources

- Data Wrangling with dplyr and tidyr Cheat Sheet is a very helpful resource for learning how to "wrangle" (manipulate) data in R

Chapter 9

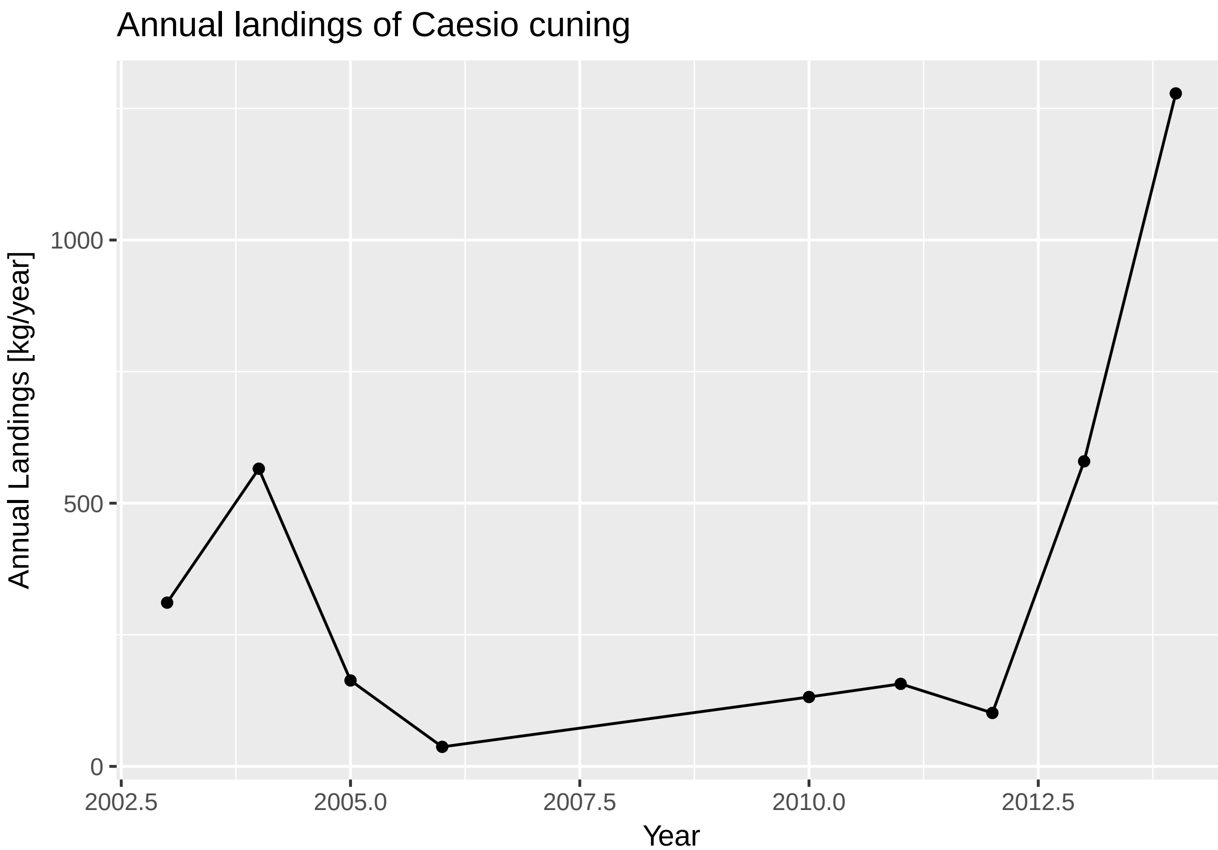
Plotting Fisheries Data

This document will now walk you through how to make make some basic fisheries plots using the data frames you created in the previous analysis section and the `ggplot` plotting function. When using `ggplot`, first start with your data frame and initialize the ggplot by specifying the plot's *aesthetics* (variables) using `aes()`. Then use the `+` operation to add at least one *geometry* (type of plot, such as a scatter plot) and any additional features to the plot. To learn more about `ggplot`, the Data Visualization with ggplot2 Cheat Sheet is a very helpful resource, as is this ggplot cookbook.

9.1 Plotting Landings

Let's plot a time series of annual landings data. We start with the annual landings data we made in the previous step, and then feed this into a ggplot.

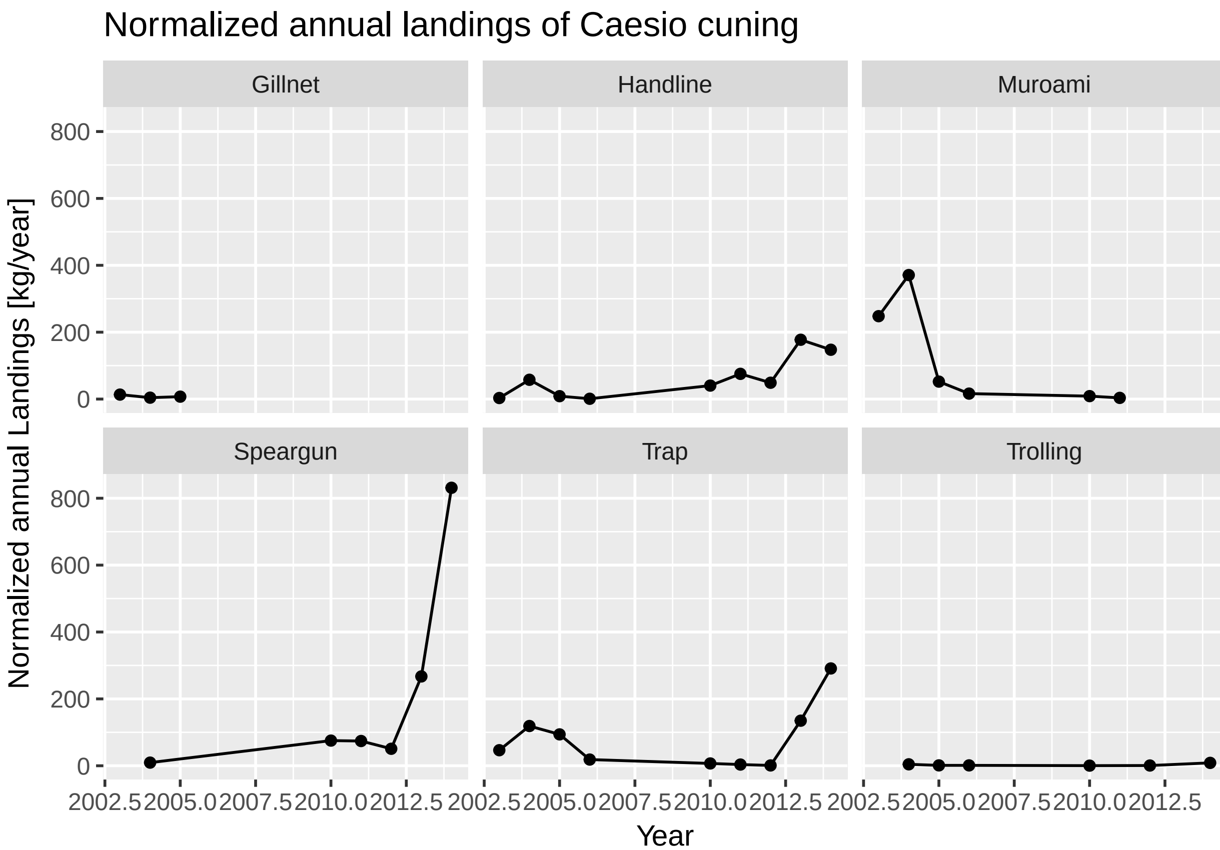
```
# Start with the annual_landings data frame you created in the last step
annual_landings %>%
  # Initialize a ggplot of annual landings versus year
  ggplot(aes(x=Year,y=Annual_Landings_kg)) +
  # Tell ggplot that the plot type should be a scatter plot
  geom_point() +
  # Also add a line connecting the points
  geom_line() +
  # Change the y-axis title
  ylab("Annual Landings [kg/year]") +
  # Add figure title
  ggtitle("Annual landings of Caesio cuning")
```



In this example, we are using `aes(x=Year, y=Annual_Landings_kg)` to specify that the we want to plot years on the x-axis and annual landings on the y-axis. We then want to visualize these variables with both a scatter plot (`geom_point()`) *and* a line plot (`geom_line()`) geometry.

It appears landings were going down between 2004 and 2011, but have been increasing since then. Again, you may be interested in looking across different gear types. To plot, we use ggplot's faceting functionality, which tells `ggplot` to divide up the data by a certain variable, `Gear` in this case, and make multiple similar plots. You can use the `facet_wrap()` function to accomplish this.

```r
# Start with the landings data frame
annual_gear_landings %>%
  # First, group the data by year
  group_by(Year,Gear) %>%
  # Initialize a ggplot of annual landings versus year
  ggplot(aes(x=Year,y=Annual_Landings_kg)) +
  # Tell ggplot that the plot type should be a scatter plot
  geom_point() +
  # Also add a line connecting the points
  geom_line() +
  # Change the y-axis title
  ylab("Normalized annual Landings [kg/year]") +
  # Add figure title
  ggtitle("Normalized annual landings of Caesio cuning") +
  # This tells the figure to plot by all different gear types
  facet_wrap(~Gear)
```

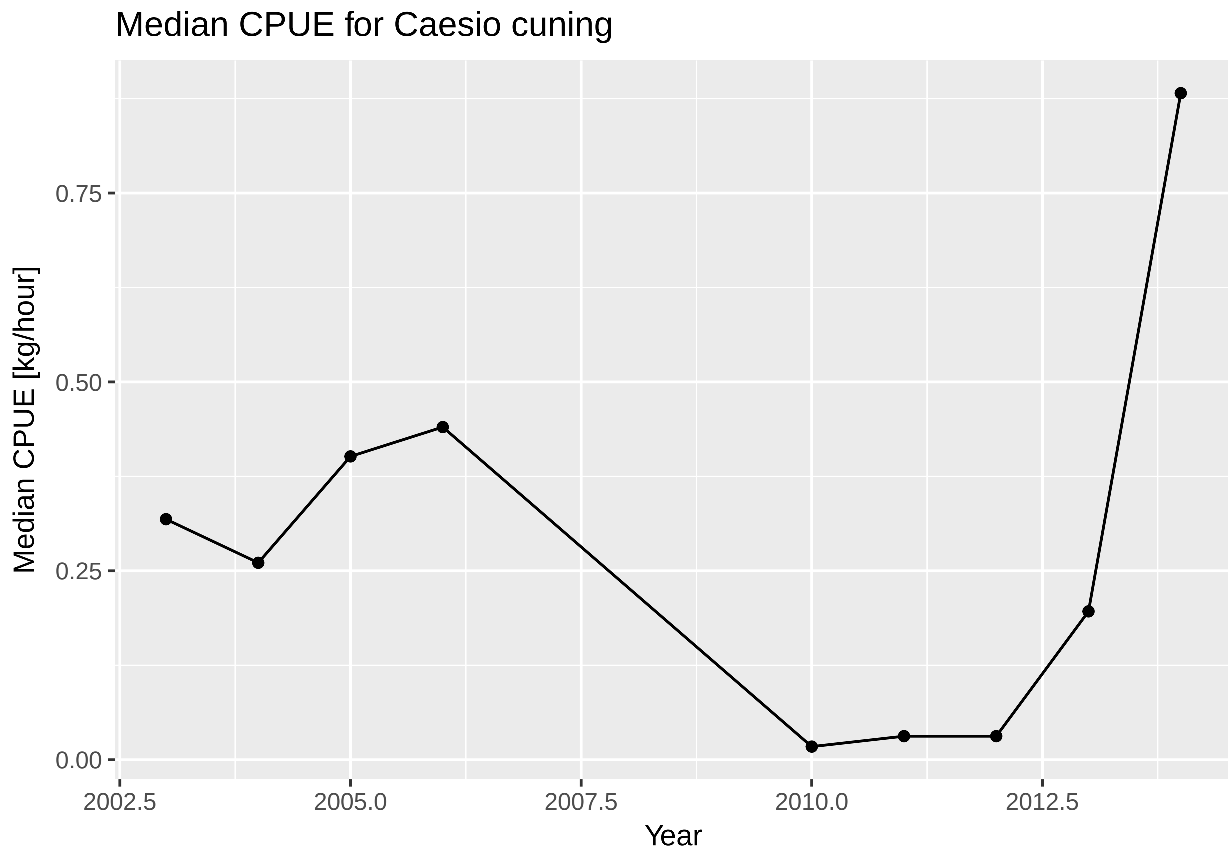


It now becomes clear that the recent increase in catch seems to be concentrated in speargun and trap fishing. Meanwhile, catch from muroami, a very destructive type of gear where nets are driven into the reef, has dropped to 0 since a ban of that gear in 2012 - a good sign that management regulation is working.

9.2 Plotting CPUE

You may also be interested in plotting median catch-per-unit-effort (CPUE). You take your CPUE data frame made in the last step and feed it into ggplot.

```
# Start with the CPUE data frame
cpue_data %>%
  # Initialize a ggplot of median CPUE versus year
  ggplot(aes(x=Year,y=Median_CPUE_kg_hour)) +
  # Tell ggplot that the plot type should be a scatter plot
  geom_point() +
  # Also add a line connecting the points
  geom_line() +
  # Change the y-axis title
  ylab("Median CPUE [kg/hour]") +
  # Add figure title
  ggtitle("Median CPUE for Caesio cuning")
```



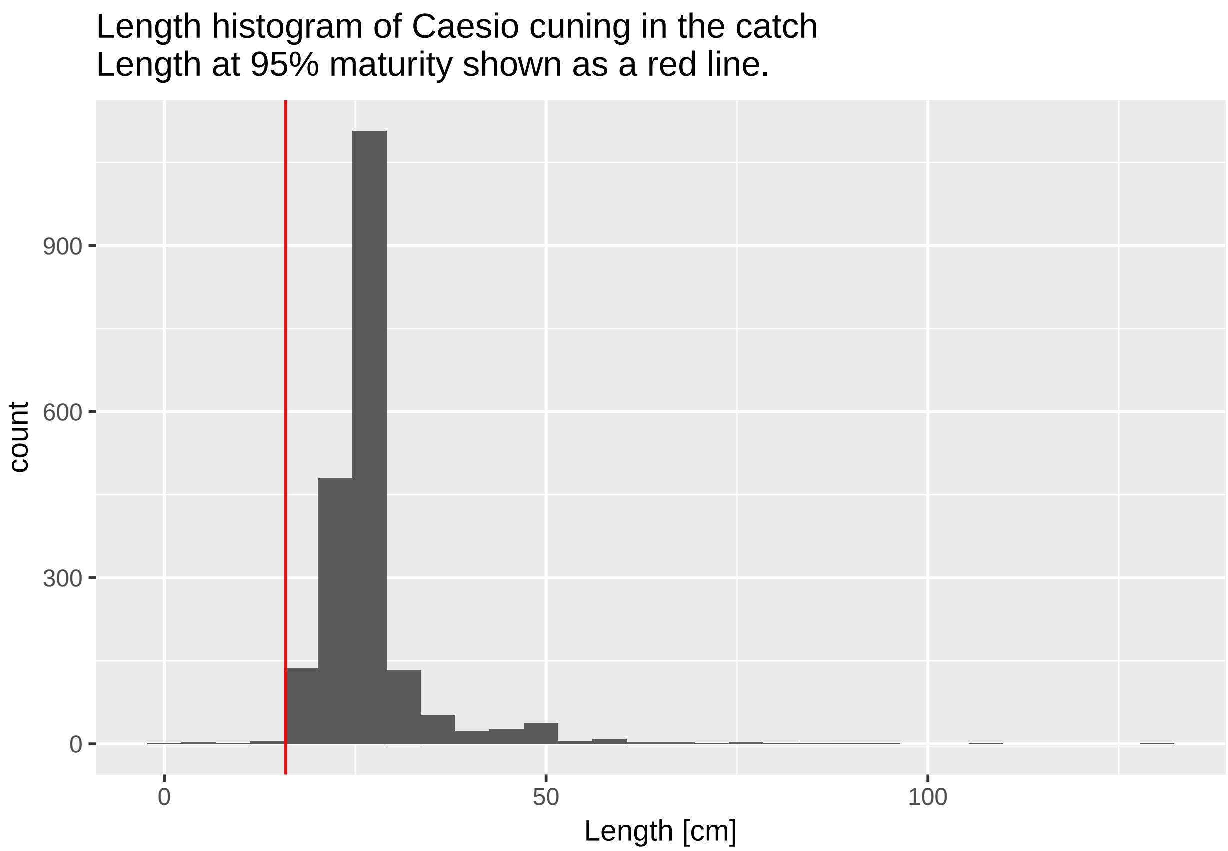
CPUE appears to have increased significantly during the last years. This may be due to increasing abundance in the water, which would be a good thing, but may also be indicative of increased gear efficiency coinciding with the transition to traps and spearguns, which may be concerning.

9.3 Plotting Length Frequency

Finally, let's first look at the length data from the catch, which gives an indication of the size structure and health of the population. Let's look at the length data for 2014, which is the most recent year of data available. We first filter the data to be only from 2014 using the `filter()` function. We then create a histogram of the length data, which shows how many individuals of each size class were measured in the catch. On the histogram, we'll also add a vertical line to show the length at which fish mature to get a sense of how sustainable the catch is - the catch should be composed mostly of mature fish. This information comes from the life history parameter data input file.

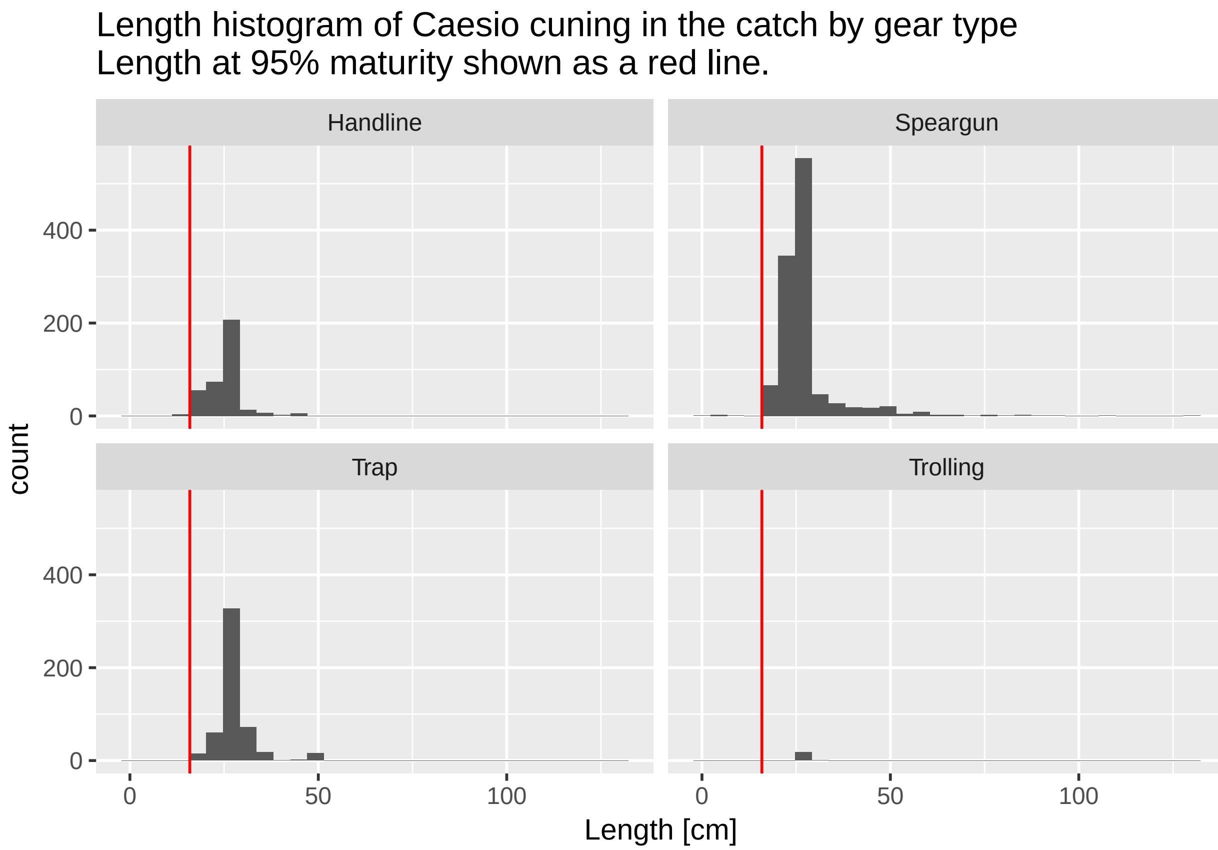
```
# Start with the landings data frame
landings_data %>%
  # Filter data to only look at length measurements from 2014
  filter(Year == 2014) %>%
  # Initialize ggplot of data using the length column
  ggplot(aes(Length_cm)) +
  # Tell ggplot that the plot type should be a histogram
  geom_histogram() +
  # Change x-axis label
  xlab("Length [cm]") +
  # Add figure title
  ggtitle("Length histogram of Caesio cuning in the catch\nLength at 95% maturity shown as a red line.")
```

```
  # Add a red vertical line for m95, the length at which 95% of fish are mature. Any fish below this le
  geom_vline(aes(xintercept=life_history_parameters$m95),color="red")
```



You might also be interested in seeing how the size composition varies by gear type. You can recreate the figure above, but separate the histograms out by gear type using ggplot's "facet" function.

```
# Start with the landings data frame
landings_data %>%
  # Filter data to only look at length measurements from 2014
  filter(Year == 2014) %>%
  # Initialize ggplot of data using the length column
  ggplot(aes(Length_cm)) +
  # Tell ggplot that the plot type should be a histogram
  geom_histogram() +
  # Change x-axis label
  xlab("Length [cm]") +
  # Add figure title
  ggtitle("Length histogram of Caesio cuning in the catch by gear type\nLength at 95% maturity shown as
  # Add a red line form95, the length at which 95% of fish are mature. Any fish below this length may b
  geom_vline(aes(xintercept=life_history_parameters$m95),color="red") +
  # This tells the figure to plot by all different gear types, known as facetting
  facet_wrap(~Gear)
```



It appears as if the size structure is about the same from each gear, although by far the most amount of fish are caught using speargun. Very few fish are caught using trolling.

These plots indicate a generally increasing catch, CPUE, and a healthy size structure. Our results demonstrate that the population is likely doing fairly well, and may be recovering since the 2012 ban of muroami fishing gear.

9.4 Helpful Resources

- Data Visualization with ggplot2 Cheat Sheet
- ggplot cookbook

Chapter 10

Wrapping Up

Congratulations! You've completed our introduction to R and are ready to start using it to conduct fishery analyses. We think you'll find, like us, that R will dramatically expand your ability to ask interesting and important questions about your data. Furthermore, by doing all your data processing, analysis, and plotting in R you can develop a streamlined work flow that is entirely reproducible and does not jeopardize the integrity of your raw data.

R is a very powerful and flexible tool. Once you become comfortable with the types of analyses outlined in this tutorial you can start using R to run fishery models or build your own.

10.1 R Packages for Fishery Analysis

Fortunately, there are already numerous R packages available that are specifically designed for different types of fisheries analyses. The following four packages all contain very useful functions, and numerous other packages are available. Before jumping into these packages, however, we recommend you become familiar with R and more basic analyses first. Also, keep in mind that **R is an open-source language**, which means that *anyone* can submit packages. You should therefore always be sure to read the package documentation and double check your work to be sure things are performing as expected.

- TropFishR - Fish stock assessment methods and fisheries models based on the FAO Manual "Introduction to tropical fish stock assessment" by P. Sparre and S.C. Venema. Focus is the analysis of length-frequency data and data-poor fisheries.
- DLMTool - Implementation of management procedures for data-limited fisheries
- LBSPR - Functions to run the Length-Based Spawning Potential Ratio (LBSPR) method. The LBSPR package can be used in two ways: 1) simulating the expected length composition, growth curve, and SPR and yield curves using the LBSPR model and 2) fitting to empirical length data to provide an estimate of the spawning potential ratio (SPR).
- fishmethods - Fishery science methods and models from published literature

All four of these packages are available on CRAN and can be downloaded directly in RStudio by running the following command:

```r
install.packages(c('TropFishR','DLMTool', 'LBSPR', 'fishmethods'))
```

10.2 Additional Resources

Throughout this guidebook we provided links to additional useful R resources. We now include all of these resources below for your convenience.

- R for Data Science
- Cookbook for R
- RStudio Cheat Sheets
- Jenny Bryan's Stat 545 Course Syllabus
- Simple Guidelines for Effective Data Management
- Research Data Management

Bibliography