Introduction to R for Distance

Topic 1: Introduction to R and RStudio

Most users will use a graphical user interface (GUI) to R. While the basic R installation comes with a simple GUI, here we adopt the use of RStudio, which considerably facilitates an introduction to R by providing many shortcuts and convenient features which we introduce next.

By default, RStudio has the following layout:

- the R console (left pane)
- scripts, if open (top left pane)
- the workspace objects (top right pane)
- the loaded packages (bottom right pane)
- the created plots (bottom right pane)
- the help files (bottom right pane)
- a file navigator system (bottom right pane)

Note that you can customize the aspect of RStudio (e.g. font size and colours of the smart syntax highlighting scheme) via `Tools|Global options`.

You know that R is ready to receive a command when you see the R prompt in the console (on the bottom left tab by default in RStudio): ">". If you type a line of code that is not complete, R presents the "+" character, so that the user knows it expects the conclusion of the current line. Important note: while the prompt ">" and "+" will be shown in this tutorial's code, you should not try to add either ">" nor "+" to the command line: this is something that R does for you and will complain if you try to do it yourself!

Exercise 1.1: R as a Calculator

R is a very powerful calculator! Mathematical functions like `sin()` and `log()` and numbers like `pi` are built in. Logical statements like `==`, `>`, and `!=` will return `TRUE` or `FALSE`. Try some simple maths, say for example (you need to press enter after each line so that the line is evaluated).

```
4 + 3
```

```
## [1] 7
```

```
5 == 2
```

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

```
log(8)
```

```
## [1] 2.079442
```

```
sin(pi)
```

```
## [1] 1.224647e-16
```

Task 1.1.1: St Andrews was founded in 1413. How old is the university? Is the university more than 500 years old?

```
2019 - 1413
```

```
## [1] 606
```

```
(2019 - 1413) > 500
```

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

Task 1.1.2: What is the sum of 5! and the natural log of 38?

```r
5 * 4 * 3 * 2 * 1 + log(38)
```

```
## [1] 123.6376
```

```r
# alternatively,

factorial(5) + log(38)
```

```
## [1] 123.6376
```

```r
sum(factorial(5), log(38))
```

```
## [1] 123.6376
```

Task 1.1.3: Calculate the square root of 29340 (hint: there is a function called `sqrt()`).

```r
sqrt(29340)
```

```
## [1] 171.2892
```

Task 1.1.4: What is the product of 12 and 74 divided by 8 plus 7?

```r
12 * 74 / (8 + 7)
```

```
## [1] 59.2
```

Task 1.1.5: What is the area of a circle with a radius of 23 km?

```r
pi * 23 ^ 2
```

```
## [1] 1661.903
```

Exercise 1.2: Saving and running code from scripts

So far, we've been typing commands directly into the console. Most of the time, you will write code in a script file and then execute it. Working with script files rather than typing code directly into the console is a key part of reproducibility when using R: the script file allows you to keep track of what you've done, and to re-run analyses as needed. Script files have the extension `.R`. You can add comments to your script file using `#`, which basically tells R to ignore whatever follows on the same line. For example:

```r
# this is a comment

3 + 5 # this is also a comment
```

```
## [1] 8
```

```r
# 3 + 5 + 8 this is also a comment

# each line of a comment
# needs its own #
```

Code can be run from your script file by copy-pasting, using the "Run" button at the top right of the script window, or using keyboard shortcuts: Command+Enter for Macs and Ctrl+Enter for Windows. Note that other keyboard shortcuts can be found under `Tools|Keyboard Shortcuts Help`.

Task 1.2.1: Open a script file and save it on your computer. Repeat the tasks from Exercise 1.1, but this time, type the code in the script file then send it to the console. Add comments to the script file describing what the code does.

Exercise 1.3: Assignments and operations using variables

Variables store values or objects so that they can be accessed later. In R, we use `<-` to assign values or objects to variables and `=` to set function arguments. For logical tests, as mentioned before, we use `==`. The shortcut for `<-` in R is `alt-`.

Create a variable called x and and assign to it the value 5.

```
x <- 5 # this is pronounced "x gets 5"
```

When you execute this line of code, nothing happens in the console. That's a good thing! the `>` symbol appears on the next line, indicating that R has completed your request and is ready for the next instruction.

Now, look in the `Environment` pane of your RStudio session. You can see that the variable x now exists in your R Environment and it has a value of 5. You can also see this by typing `ls()` into your console, which shows you all of the variables in your R Environment.

Type `x` into your console, then try a few operations on the variable x.

```
x
```

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

```
x*2
```

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

```
x == 10
```

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

Task 1.3.1: Translate the statement "y gets 10" to code. Verify that `y` exists in your R Environment. Then, add, subtract, multiply, and divide x and y and store the results as new variables.

```
y <- 10

add.xy <- x + y

sub.xy <- x - y

mul.xy <- x * y

div.xy <- x / y
```

To create variables with length greater than 1, we use the function `c()` which combines or concatenates objects.

```
z <- c(1, 2, 3)
```

Now, we should see the variable `z` in our Environment pane and/or when we call `ls()`. Whereas `x` and `y` contain single values, `z` is a *numeric vector* of length three. You can see this in the Environment pane or by checking the structure of `z` with `str(z)`.

You can find out what different functions do using `?` or searching in the `Help` pane of RStudio. For example, to find out what `ls()` does type `?ls()` into the console.

Task 1.3.2: There are two functions that can help you to create sequences of numbers: the colon operator `:` and the function `seq()`. Find the help pages for these functions and use them to create variables called `a` and `b` that contain sequences of numbers from 1 to 10 by one and 1 to 100 by ten, respectively. Are `a` and `b` the same length?

```
?seq()

?':'
```

```
a <- 1:10

b <- seq(1, 100, by = 10)

length(a) == length(b)
```

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

Task 1.3.3: R performs operations *element-wise*. To understand this concept, try the following operations with variables that we have already created and note what happens in each case. Which operations cause a warning message to appear and why?

```
a / b
```

```
##  [1] 1.0000000 0.1818182 0.1428571 0.1290323 0.1219512 0.1176471 0.1147541
##  [8] 0.1126761 0.1111111 0.1098901
```

```
a / y
```

```
##  [1] 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
a * z
```

```
## Warning in a * z: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  1  4  9  4 10 18  7 16 27 10
```

```
a + b
```

```
##  [1]   2  13  24  35  46  57  68  79  90 101
```

```
z + x
```

```
## [1] 6 7 8
```

```
b + z
```

```
## Warning in b + z: longer object length is not a multiple of shorter object
## length
```

```
##  [1]  2 13 24 32 43 54 62 73 84 92
```

Task 1.3.4: Another useful function is `rep()`. Read the help file for this function, and note the arguments `times`, `length.out`, and `each`. Create three vectors: `j` should contain the numbers 1 to 5 repeated three times, `k` should contain the numbers 1 to 5 repeated three times *each*, and `l` should contain the numbers 1 to 5 repeated and have length 15. Which two resulting vectors are the same?

```
j <- rep(1:5, times = 3)

k <- rep(1:5, each = 3)

l <- rep(1:5, length.out = 15)

j == k
```

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

```r
j == l
```

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

```r
k == l
```

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

Task 1.3.5: Use the remove function `rm()` to remove all objects from the Environment.

```r
rm(a)

rm(b, j, k, l)

rm(list = ls())
```

Topic 2: Data Types and Formats

Objects can have classes, which allow functions to interact with them. Objects can be of several classes. We already used the class `numeric`, which is used for general numbers, but there are also additional very commonly used classes

- `integer`, for integer numbers
- `character`, for character strings, which must be in quotes
- `factor`, used to represent levels of a categorical variable
- `logical`, the values TRUE and FALSE

While many others exist, these are the more commonly used. Outputs of some analyses have special classes, as an example, the output of a call of function `lm()` is an object of class `lm`, i.e., a linear model. Typically, functions behave differently according to the class of an object. As an example, note how `summary()` treats differently an object of class `factor` or one of class `numeric`, producing a table of counts per level for a factor but a 6 number summary for numeric values.

```r
obj1 <- factor(c(rep("a", 12), rep("b", 4), rep("c", 2)))

summary(obj1)
```

```
##  a  b  c
## 12  4  2
```

```r
obj2 <- c(2, 5, -0.2, 89, 12, -3, -5.4)

summary(obj2)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -5.4    -1.6     2.0    14.2     8.5    89.0
```

We can check the class of an object using function `class`, as in the following examples

```r
class(obj1)
```

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

```r
class(obj2)
```

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

```
class(TRUE)
```

```
## [1] "logical"
```

It is sometimes useful to coerce objects into different classes, but care should be used when doing so. Some examples are presented below. Can you describe in your own words what R did below?

```
as.integer(c(3, -0.3, 0.4, 0.6, 0.9, 13.2, 12))
```

```
## [1]  3  0  0  0  0 13 12
```

```
as.numeric(c(TRUE, FALSE, TRUE))
```

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

```
as.numeric(obj1)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 3 3
```

So far, we've been working exclusively with objects called vectors. Vectors are one-dimensional and can have a length of one or more than one. For more complex data, we can use matrices, data frames, and lists to store information in multiple dimensions.

- A *matrix* is a collection of elements of the same data type (numeric, character, or logical) arranged in a fixed number of rows and columns. An array is an n-dimensional matrix.
- A *data frame* has variables as columns and observations as rows and can contain multiple data types.
- A *list* can contain different kinds of objects (data frames, matrices, vectors, etc.) with different dimensions.

Exercise 2.1: Creating different types of arrays

Task 2.1.1: Use the `matrix()` function to create a matrix of the numbers 1 to 100 in ten rows and ten columns. Can you get the numbers to read from left to right instead of from top to bottom?

```
data <- 1:100

mat <- matrix(data, nrow = 10, ncol = 10)

mat <- matrix(data, nrow = 10, ncol = 10, byrow = TRUE)
```

Task 2.1.2: You just got back from a whale survey and saw 5 fin, 2 blue, 14 humpback, 0 minke, and 1 gray whale. Use the `data.frame()` function to create a data frame of this information. Check the structure of the data frame. What would happen if you stored this data as a matrix instead?

```
whales <- data.frame("Species" = c("Fin", "Blue", "Humpback", "Minke", "Gray"),
                     "Count" = c(5, 2, 14, 0, 1))
```

Task 2.1.3: Use the `list()` function to create an object containing your name, your matrix of numbers from 1:100, and your data frame of whale sightings.

```
my.list <- list("Eiren", mat, whales)

# Could also create a named list
my.list <- list(Name = "Eiren", Matrix = mat, Whales = whales)
```

Exercise 2.2: Indexing and subsetting arrays

To index or extract information from an object, we use square brackets: `[]`. For example, we can select the third element of a vector

```
x <- c(1, 3.5, 7, 8, -7, 0.43, -1)

x[3]
```

```
## [1] 7
```

but we can also select all *except* the second and third elements of the same vector

```
x[-c(2,3)]
```

```
## [1]  1.00  8.00 -7.00  0.43 -1.00
```

We can also select only the objects which follow a given condition, say only those that are positive

```
x[x > 0]
```

```
## [1] 1.00 3.50 7.00 8.00 0.43
```

or those between (-1,1)

```
x[(x > -1) & (x < 1)]
```

```
## [1] 0.43
```

When working on a matrix the indexing is done by row and column, therefore for selecting the value that is in the third row and second column of a matrix we use

```
mat[3, 2]
```

```
## [1] 22
```

but we can also select all the elements in the second row

```
mat[2,]
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

or the fourth column

```
mat[,4]
```

```
##  [1]  4 14 24 34 44 54 64 74 84 94
```

but note that if we provide a single value, like `mat[2]` R will extract the 2nd element of the matrix column-wise:

```
mat[2]
```

```
## [1] 11
```

You can use the same type of row-column indexing for data frames, or you can extract information using the name of the column you're interested in and the dollar sign operator `$`.

```
df <- data.frame("Type" = c("a", "b", "c"),
                 "Count" = c(83, 74, 31))

df[2, 2]
```

```
## [1] 74
```

```
df$Count[2]
```

```
## [1] 74
```

Task 2.2.1: What was the maximum number of whales of a species seen during the whale survey and which species were they? Did you see zero of any species? Which?

```
max(whales$Count)
```

```
## [1] 14
```

```
which.max(whales$Count)
```

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

```
whales$Species[which.max(whales$Count)]
```

```
## [1] Humpback
## Levels: Blue Fin Gray Humpback Minke
```

```
any(whales$Count == 0)
```

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

```
whales$Species[which(whales$Count == 0)]
```

```
## [1] Minke
## Levels: Blue Fin Gray Humpback Minke
```

Task 2.2.2: Extract the whale species names from the object `my.list` that you created in 2.1.3.

```
my.list$Whales$Species
```

```
## [1] Fin      Blue     Humpback Minke    Gray
## Levels: Blue Fin Gray Humpback Minke
```

Exercise 2.3: Operations on arrays

Task 2.3.1: Use the `head()` function to look at the built-in dataset `ToothGrowth`. What is the mean length of all guinea pig teeth? The standard deviation? What is the mean length of teeth of guinea pigs given a dose of 1 mg/day?

```
head(ToothGrowth)
```

```
##    len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

```
mean.tg <- mean(ToothGrowth$len)

sd.tg <- sd(ToothGrowth$len)

mean.1mg <- mean(ToothGrowth$len[which(ToothGrowth$dose == 1)])
```

Task 2.3.2: Use the `subset()` function to extract the data pertaining to guinea pigs treated with orange juice. How else could you extract this information?

```
oj <- subset(ToothGrowth, supp == "OJ")

oj <- ToothGrowth[ToothGrowth$supp=="OJ", ]
```

Task 2.3.3: You find an additional record from this experiment from a guinea pig treated with 2 mg of Vitamin C a day, given via orange juice, that had a tooth length of 20.3. Add this information to the `ToothGrowth` data frame. Use the function `nrow()` to determine how many records are in your updated data frame.

```
newdata <- data.frame(len = 20.3, supp = "OJ", dose = 2)

ToothGrowth <- rbind.data.frame(ToothGrowth, newdata)

nrow(ToothGrowth)
```

```
## [1] 61
```

Task 2.3.4: Add a column of tooth length divided by dose to `ToothGrowth`. Round this new data to the nearest integer.

```
ToothGrowth$LenDose <-ToothGrowth$len/ToothGrowth$dose

ToothGrowth$LenDose <- round(ToothGrowth$LenDose)
```

Topic 3: Using R in Context

Exercise 3.1: Setting up a project directory

Typically, when we're working on projects, we'll be working with several different types of files: raw data, R scripts, figures, and perhaps manuscripts and presentations. It's good practice to keep all of the files associated with a project in a single, organized directory. RStudio offers R Projects, which make it easier to associate files and folders with your R scripts.

Task 3.1.1: Create an R Project folder.

- First, somewhere on your computer, create a folder called "IntroR" or something similar.

- Within that folder, create folders for Data, Scripts, and Figures.

- Then, click `File|New Project...` within RStudio, and choose "Existing Directory"
- Navigate to the IntroR folder you created above and create your new project
- Open a new R script and save it to the Scripts folder you've created within your project directory

Exercise 3.2: Reading data in and out

Rather than importing data into R manually, typically the data we work with are imported from some external source. Typically this might be some simple file format, like a txt or a csv file, but while not covered here, direct import from say Excel files or Access data bases is possible. Such more specialized inputs often require additional packages.

RStudio includes a useful dedicated shortcut "Import dataset", by default available through the top right window of RStudio's interface. Note this shortcut essentially just calls the appropriate functions required for each import. Here we present a couple of examples using the dataset `iris`.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
summary(iris)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300  
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
##        Species  
##  setosa    :50  
##  versicolor:50  
##  virginica :50  
##                 
##                 
## 
```

Now we create a new data frame which we then modify to include a new variable

```
mydata <- iris
mydata$total <- mydata$Sepal.Length + mydata$Sepal.Width + mydata$Petal.Length + mydata$Petal.Width
```

Now, we are going to export this data set as a txt, named `mydatafile.txt`. First, use `getwd()` to check which working directory you are in (it should be the directory associated with your R project).

```
write.table(mydata, file = "./Data/mydatafile.txt", row.names = FALSE)
```

Note the use of the optional argument `row.names=FALSE`, otherwise some arbitrary row names would be added to the file. If you look in the folder you are working in, you should now have a new file there. Open it and check that it looks as you would expect. Next, we are going to import it back into R, into an object named `indat`.

We can now remove `mydata` from our workspace, then read it back in from the file.

```
rm(mydata)

indat <- read.table(file = "./Data/mydatafile.txt", header = TRUE)
```

So now we have our data back in R.

Task 3.2.1: Put the file `monthly_in_situ_co2_mlo.csv` in your Data folder. Use the function `read.csv()` to import the data. Replace any missing values with NA. Use the `save()` function to save the modified data as an `.RData` file. Remove the data from your workspace, then use the `load()` function to reload it from the `.RData` file. Use the `unique()` and `length()` functions to find the number of years included in this timeseries.

```
co2 <- read.csv("./Data/monthly_in_situ_co2_mlo.csv",
                header=TRUE, na.strings=-99.99)

save(co2, file = "./Data/co2.RData")
```

```
rm(co2)

load("./Data/co2.RData")

length(unique(co2$Yr))
```

Exercise 3.3: Using functions from packages

So far, all of the functions we've used have been available in base R. But often, we'll want to use functions from specialized packages (like Distance!). A package is just a set of functions and data sets (and the corresponding documentation plus some additional required files) which usually have some specific goal.

Installing a new package in R requires a call to function `install.packages()`. A RStudio shortcut is simply to follow the `Tools|Install packages...` shortcut.

After a package is installed it needs to be loaded to be available. In R this is done calling function `library()` with the package name as an argument. In RStudio this becomes simpler by checking the boxes under the RStudio tab packages (by default this tab is available on the bottom right window, along with the Files, Plots, Help and Viewer tabs).

Task 3.3.1: Install and load the `maps` package. Use the `map` function to plot a world map.

```
install.packages("maps")

library(maps)

map("world")
```

Task 3.3.2: Install and load the `Distance` package. Look at the help for the function `ds()`. What are the function arguments? Now run the first chunk of example code. Can you explain what this code is doing?

```
install.packages("Distance")

library("Distance")

?ds

data(book.tee.data)
tee.data <- book.tee.data$book.tee.dataframe[book.tee.data$book.tee.dataframe$observer==1, ]
ds.model <- ds(data = tee.data, truncation = 4)
summary(ds.model)
plot(ds.model)
```

Topic 4: Control Structures

Exercise 4.1: Conditional Statements

Conditional statements are constructed using `if` and `else`. In pseudocode, they have the structure:

```
# if(logical test here){
#  instruction for what to do if logical test is true
# } else {
#  thing to do if the logical test is false
#}
```

Here is a simple example of using the if-else construct:

```
a <- 5

if(a < 0){
  print("a is negative")
} else {
  print("a is positive")
}
```

```
## [1] "a is positive"
```

You can extend this construct to include a series of tests:

```
a <- 0

if(a < 0){
  print("a is negative")
} else if (a == 0){
  print("a is zero")
} else {
  print("a is positive")
}
```

```
## [1] "a is zero"
```

Task 4.1.1: Use the function `rbinom()` to generate the outcome of a single binomial trial with probability 0.5 (i.e., a coin flip). Write an if-else statement that prints "HEADS" when the outcome is 1 and "TAILS" when the outcome is 0.

```
coin <- rbinom(n = 1, size = 1, prob = 0.5)

if(coin == 0){
  print("HEADS")
} else {
  print("TAILS")
}
```

```
## [1] "HEADS"
```

Exercise 4.2: For Loops

For loops are a mechanism for iteration. Here is a very simple example:

```r
for(i in 1:10){
  print(i)
}
```

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

Here is another:

```r
x <- 25:35

for(i in 1:10){
  print(x[i])
}
```

```
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
## [1] 31
## [1] 32
## [1] 33
## [1] 34
```

Task 4.2.1: Modify the above examples to loop over values of i from 1 to 10. In each iteration, calculate the factorial of i and store it in the ith position of a new vector. How could you make the same calculations without using a for loop?

```r
ifac <- vector()

for(i in 1:10){
  ifac[i] <- factorial(i)
}

factorial(1:10)
```

```
##  [1]       1       2       6      24     120     720    5040   40320
##  [9]  362880 3628800
```

Task 4.2.2: A Fibonacci Series is a series of numbers in which each number is the sum of the two preceding numbers (i.e., 1, 1, 2, 3, 5, 8 . . .). Write a for loop to generate a Fibonacci Series of any length.

It may be helpful to break down this task into the following steps:
1. create a variable N to store the length of the series, using N = 10 as an example
2. initialize a vector in which to store your series
3. set the first two values of the vectors to 1
4. write a loop that begins with i = 3 and ends with i = N

```
N <- 10

Fib <- rep(NA, 10)

Fib[1:2] <- 1

for(i in 3:N){

  Fib[i] <- Fib[i-2] + Fib[i-1]

}
```

Bonus: Add a conditional statement to your for loop that sets positions 1 and 2 equal to 1 automatically. Modify the loop to iterate from 1:N.

Topic 5: Basic Functions

Exercise 5.1: Writing your own functions

If you will repeat a process often, it is worthwhile to write a function to do it. Functions in R have the syntax:

```
myfunction <- function(arg1, arg2, etc.){
  statements
  return(object)
}
```

Where `myfunction` is the name of the function, `arg1` and `arg2` are arguments you will pass to the function, `statements` are the operations you will perform on those arguments, and `object` is the object that you want to get out of your function.

Task 5.1.1: Write a function to calculate the mean of a vector.

```
average <- function(v){
  avg <- sum(v)/length(v)
  return(avg)
}
```

Task 5.1.2: Earlier, we wrote a for loop to generate a Fibonacci sequence. Now, write a function that takes `length` as an argument and returns a Fibonacci sequence of length = `length`.

```
FibFunc <- function(length){

  out <- rep(NA, length)

  for(i in 1:length){
    if(i == 1 | i == 2){out[i] <- 1} else
      out[i] <- out[i-1] + out[i-2]
  }

  return(out)
}
```

Exercise 5.2: Sourcing functions from scripts

As you begin to do more complex analyses, it will make sense to write functions, save them in their own scripts, and then `source()` them from whatever script you are working in. This way, you can use a custom function in multiple analyses without copy-pasting the code, which helps prevent errors and version control issues from creeping in.

Task 5.2.1: Store your Fibonacci sequence generating function in its own script file. Modify the function script to plot the Fibonacci sequence (i.e., include `plot(out)` after the for loop). Source the script and run the function with length = 10.

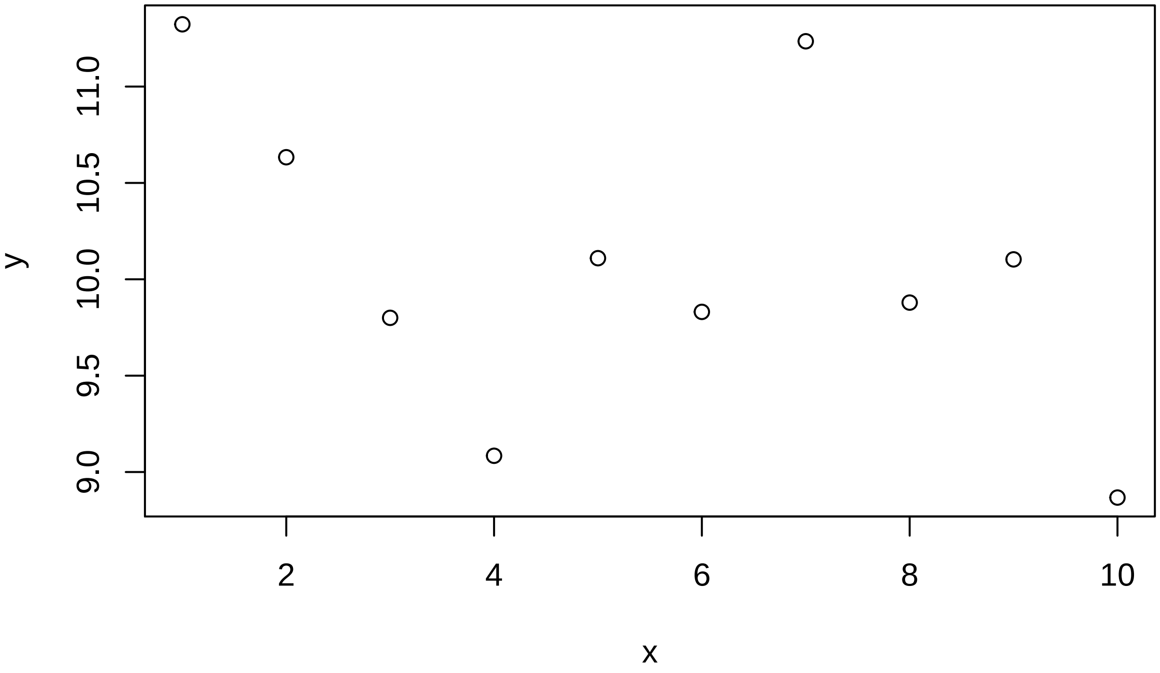
Topic 6: Basic Plots

Exercise 6.1: Scatterplots, Line Plots, Boxplots

The most basic plots in R are scatterplots, created using the `plot()` function.

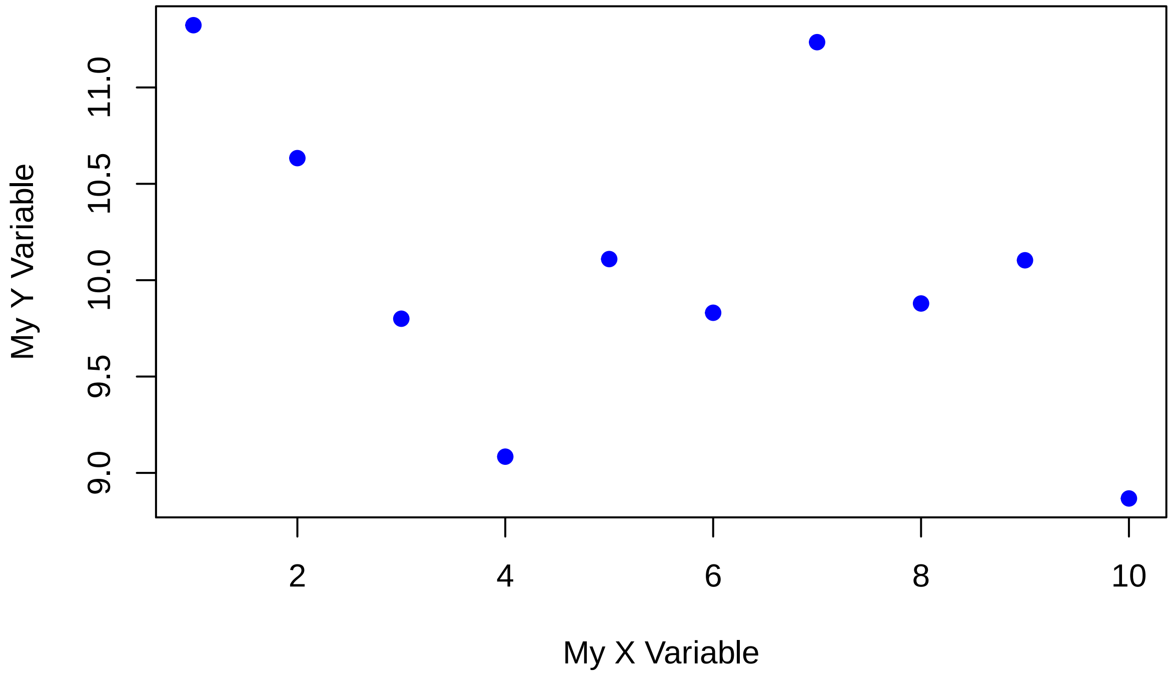
```
x <- 1:10
y <- rnorm(10, 10, 1)

plot(x, y)
```



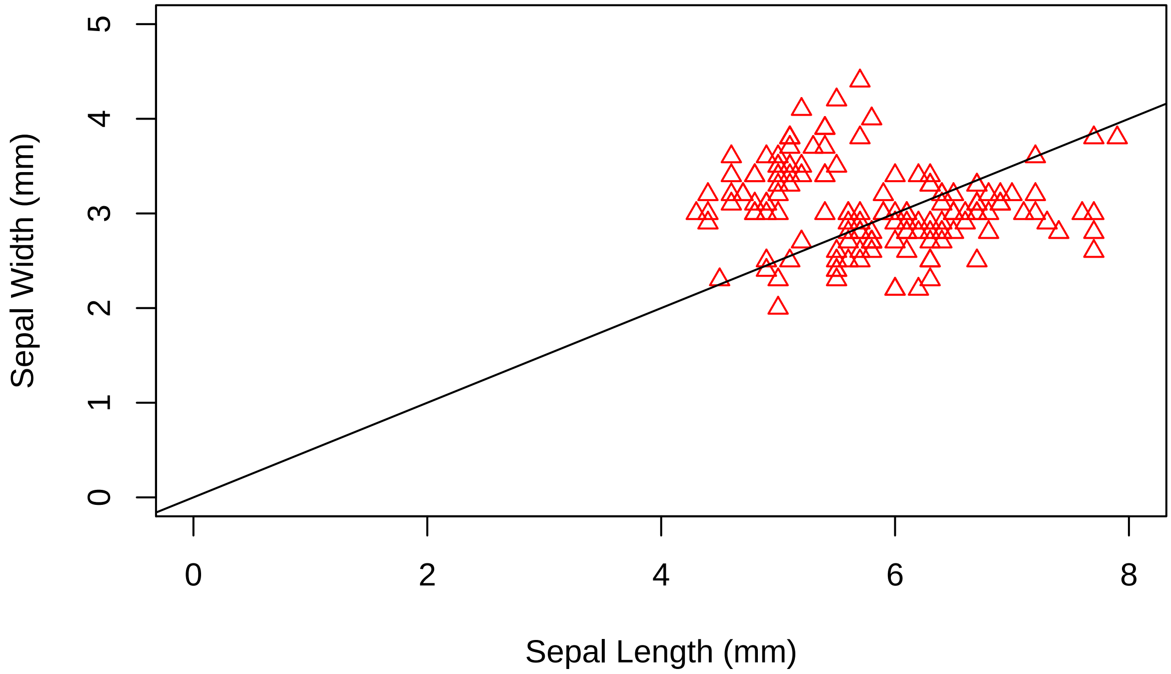
This function has arguments that can be used to e.g., include axis labels, change the colors of the points:

```
plot(x, y,
     xlab = "My X Variable", ylab = "My Y Variable",
     pch = 19, col = "blue")
```



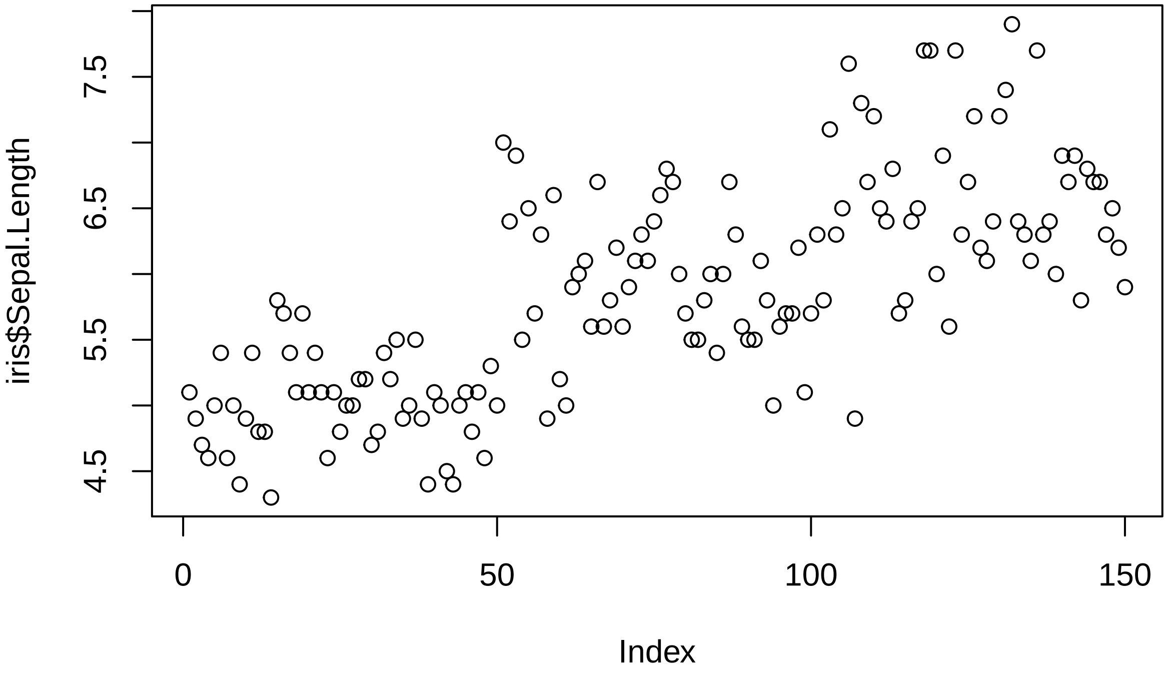
Task 6.1.1: Create a scatterplot of sepal length vs. sepal width, using the `iris` dataset. Change the points to be red triangles (hint: look at `?pch`). Change the axis titles to something sensible, and change the axes to so the origin of the plot is at 0, 0. Add an (arbitrary) diagonal line that goes through the data points (hint: look at `?abline`.

```
plot(iris$Sepal.Length, iris$Sepal.Width,
     pch = 24, col = "red",
     xlab = "Sepal Length (mm)", ylab = "Sepal Width (mm)",
     xlim = c(0, 8), ylim = c(0, 5))
abline(a = 0, b = 0.5)
```



The plot function is, as we have seen before for function `summary()`, a function that attempts to do something smart depending on the type of arguments used. Using the data set iris previously considered, plot examples are implemented below, with some optional arguments being used to show some of the possibilities to customize plots.

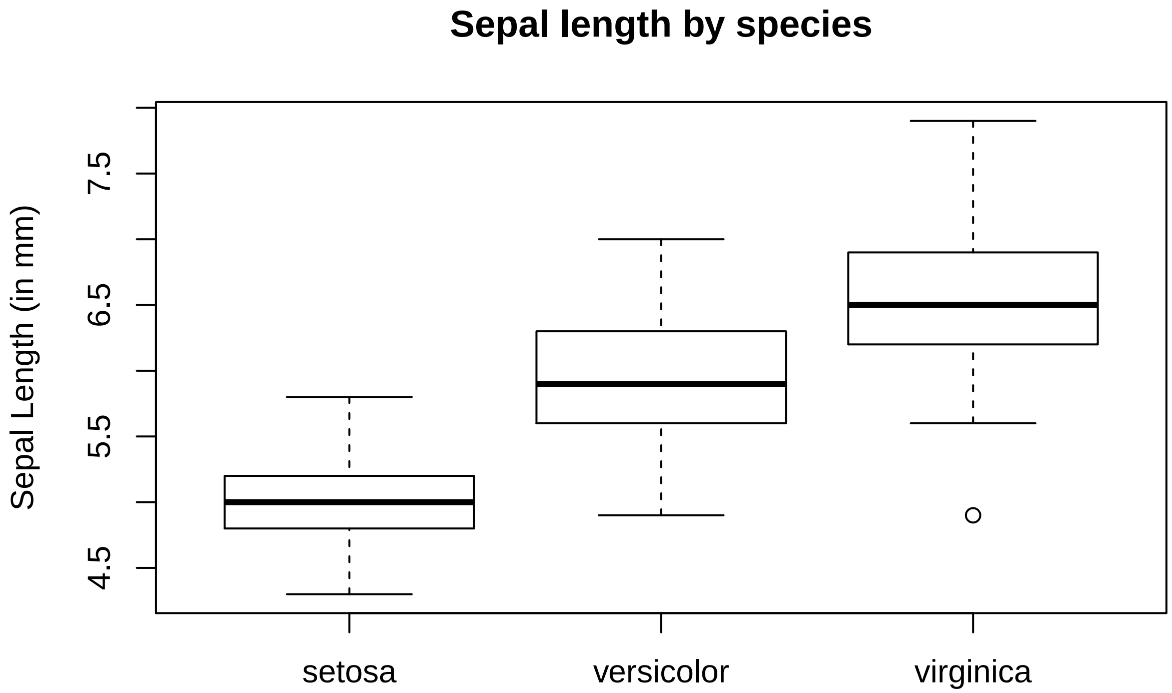
```
plot(iris$Sepal.Length)
```



We now add some labels to a boxplot of sepal length as a function of species. This can be done using either the `plot()` function or `boxplot()` function.

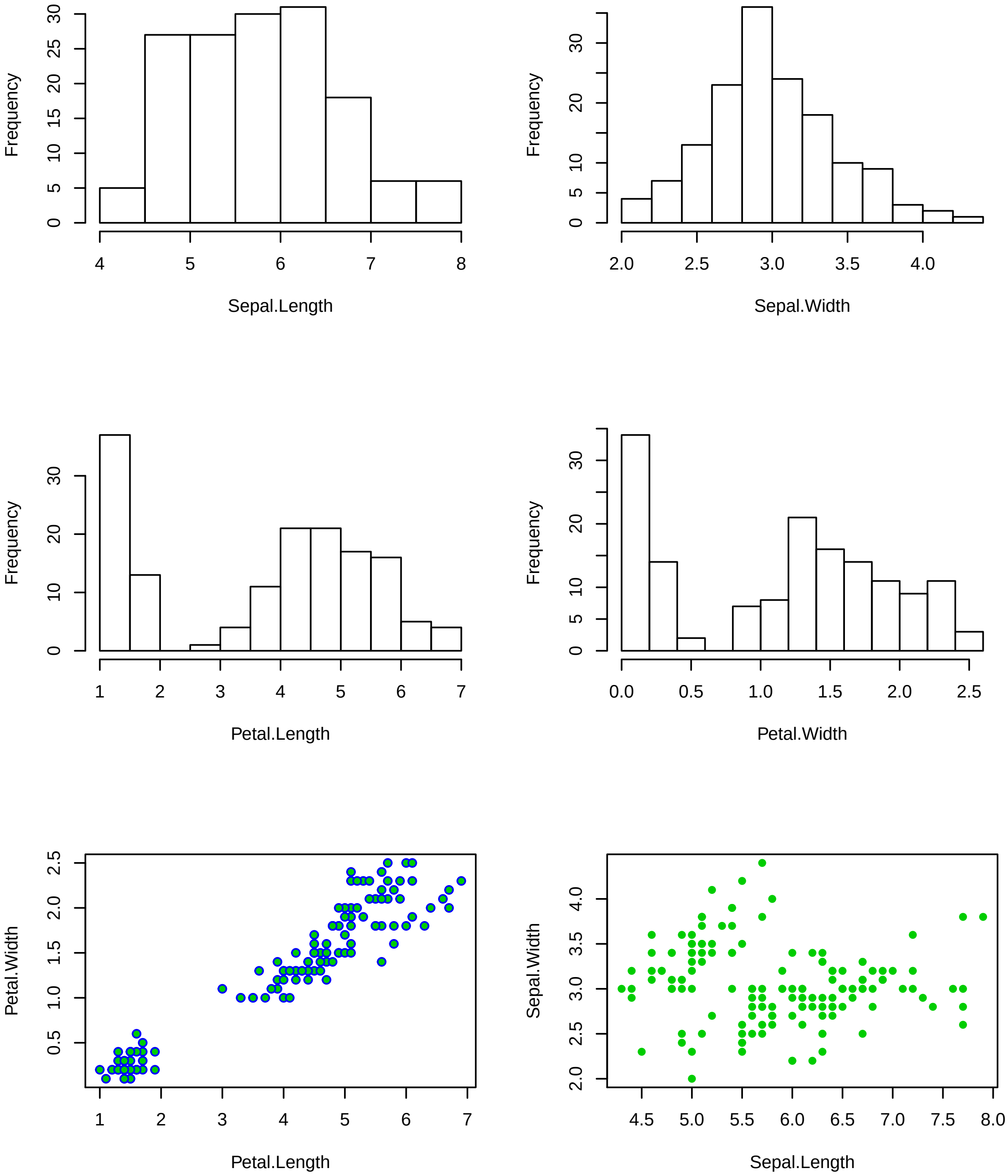
```
plot(iris$Species, iris$Sepal.Length,
     ylab="Sepal Length (in mm)",
     main="Sepal length by species")

boxplot(Sepal.Length ~ Species, data = iris,
        ylab="Sepal Length (in mm)",
        main="Sepal length by species")
```



We can also set the graphic window to hold multiple plots. This is done with the argument `mfrow`, one of the arguments in function `par`. An example follows, in which we leverage on the use of function `with` to avoid having to constantly use `indat$` to tell R where the data can be found.

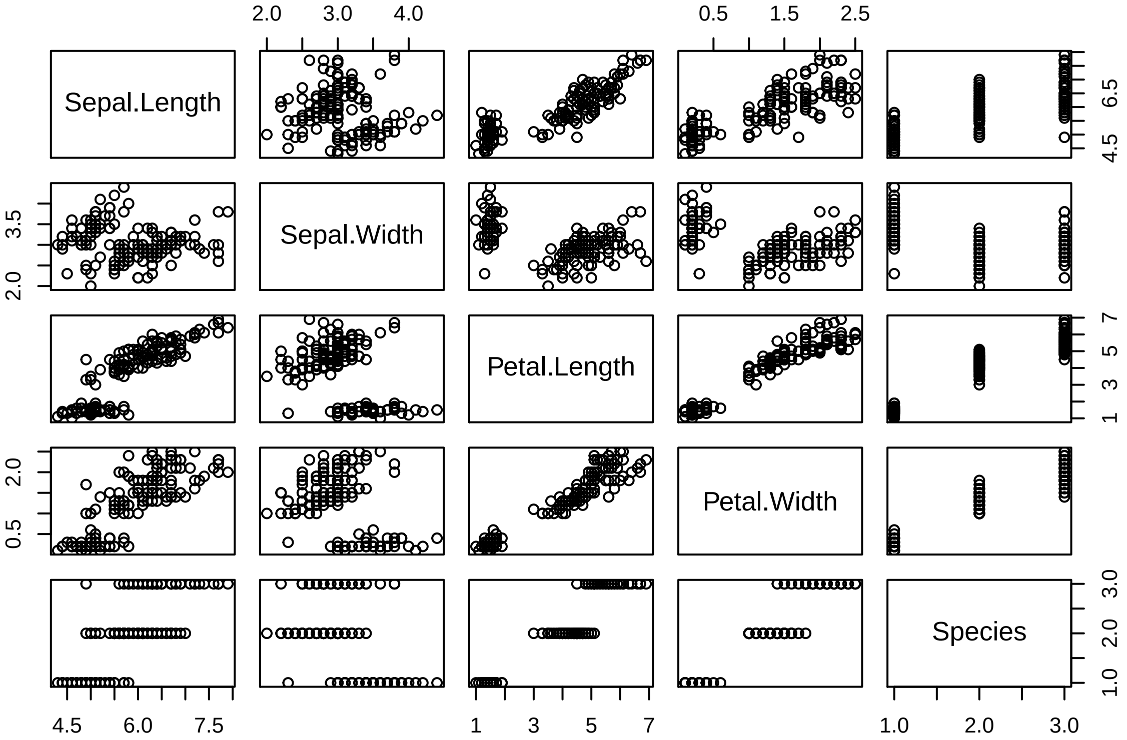
```
#define two rows and 2 columns of plots
par(mfrow=c(3,2))
with(iris, hist(Sepal.Length, main=""))
with(iris, hist(Sepal.Width, main=""))
with(iris, hist(Petal.Length, main=""))
with(iris, hist(Petal.Width, main=""))
with(iris, plot(Petal.Length, Petal.Width, pch=21, col=12, bg=3))
with(iris, plot(Sepal.Length, Sepal.Width, pch=16, col=3))
```



Looking at the help for function `par` gives you an insight to the level of customization one can reach with respect to these graphical parameters, via dozens of different arguments.

We can look at the correlation structure between all variables using function `pairs()`.

```r
par(mfrow=c(1,1))
pairs(iris)
```

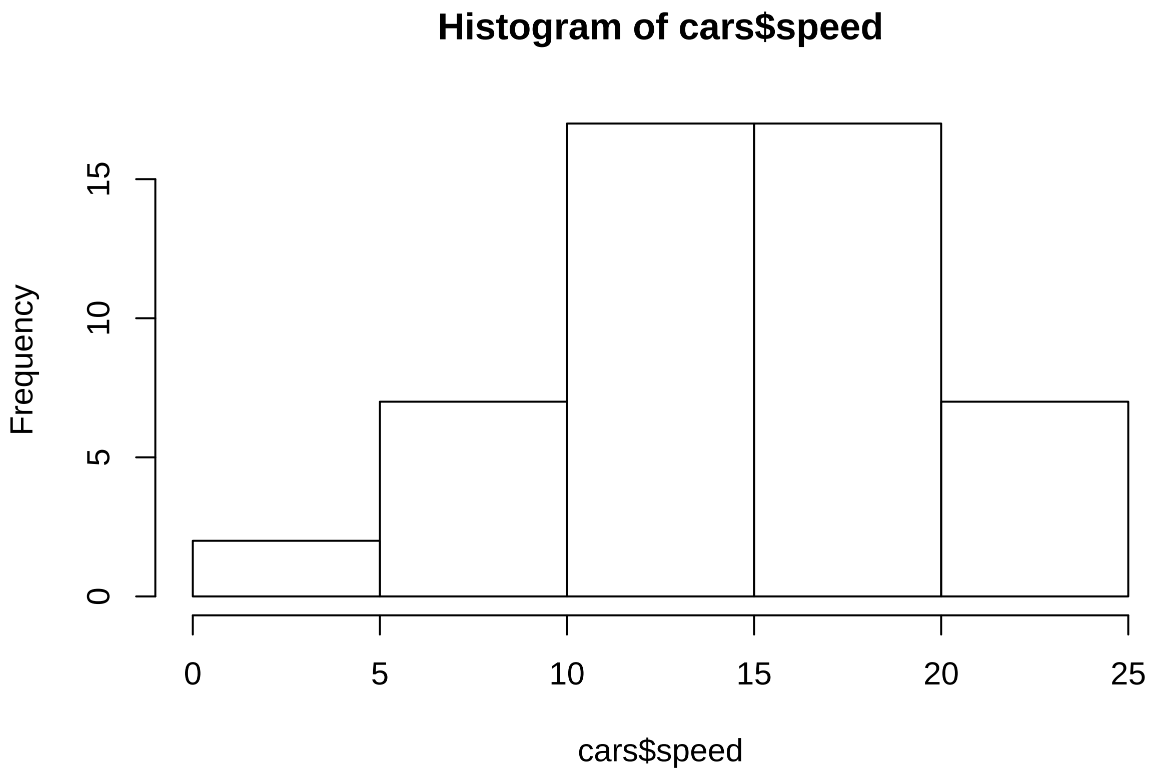


Task 6.1.2: Using data `cars`, create a plot that represents the stopping distances as a function of the speed of cars. Use the `points` function to add a special symbol to points corresponding to cars with speed lower than 15 mph, but distance larger than 70m. Check out the function `text` to add text annotations to plots. Customize axis labels.

Exercise 6.2: Histograms

A histogram is a type of plot where data are binned and the number of data in each bin are counted. In R, the function `hist()` computes the histogram and plots it. The stored histogram object contains useful information about the breakpoints and counts:

```
h <- hist(cars$speed)
```



```
h$counts
```

```
## [1]  2  7 17 17  7
```

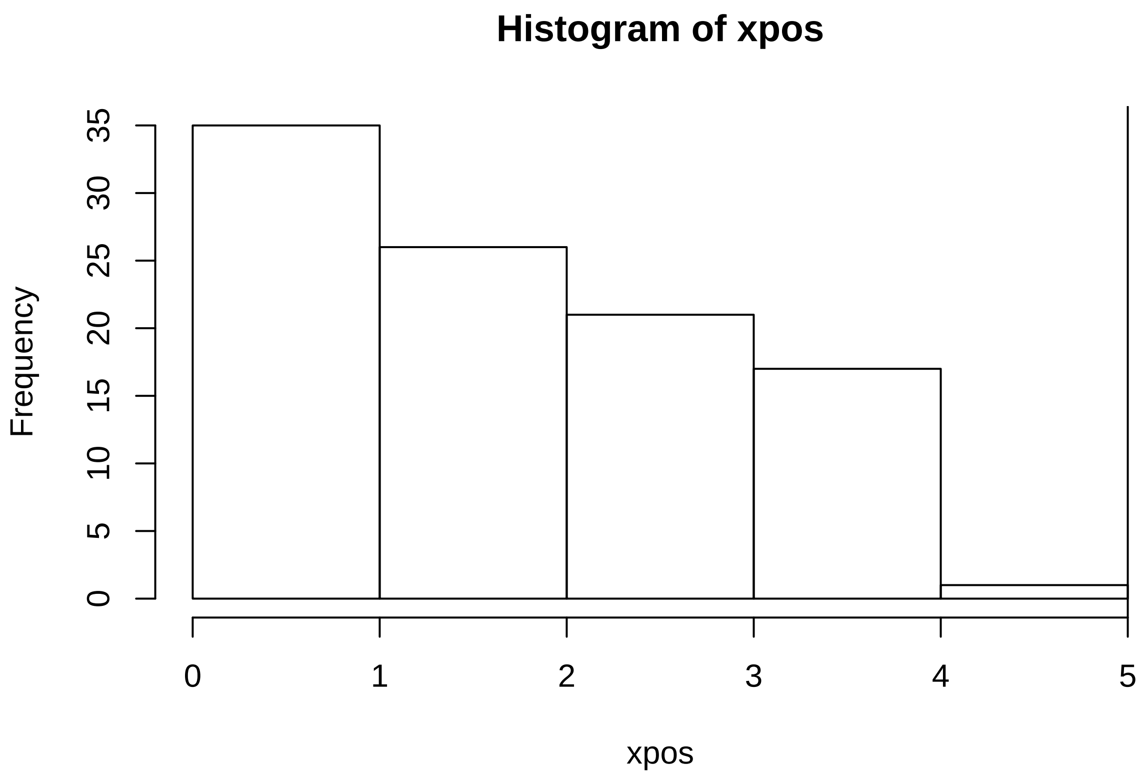
Task 6.2.1: Generate 100 random normal deviates with mean = 0 and sd = 2. Use the `abs()` function to make all of the deviates positive. Plot the deviates using a histogram with breakpoints at 0, 1, 2, etc. Add a vertical line at x = 5.

```
x <- rnorm(100, 0, 2)

xpos <- abs(x)

xmax <- ceiling(max(xpos))

hist(xpos, breaks = seq(0, xmax, by = 1))
abline(v = 5)
```



Exercise 6.3: Saving plots

You can save plots using the Export button in the plot pane of RStudio, or you can write code to save plots. The latter is preferable if you want to make sure your code is reproducible, but we'll demonstrate it both ways.

```
pdf(file = "./Figures/DemoHist.pdf", width = 6, height = 4) # open the device

hist(xpos, breaks = seq(0, xmax, by = 1))
abline(v = 5)

dev.off() # close the device
```

Topic 7: Intro to Stats in R

There are many, many statistical tests built in to R and to R packages. Here, we will show a few basic examples of using statistical tests and of fitting models and reviewing model results.

Exercise 7.1: T-Test

Task 7.1.1: Use the `rnorm()` function to generate two vectors of 100 numbers with means and standard deviations of your choice. Then, use the `t.test()` function to determine whether the means of your two vectors are significantly different. What is the p-value of your t-test?

```
x <- rnorm(100, mean = 5, sd = 1)
y <- rnorm(100, mean = 4.5, sd = 2)
```

```r
t <- t.test(x, y)

t$p.value
```

```
## [1] 0.08800098
```

Exercise 7.2: Linear Model

One of the most common type of data analysis is a regression model. Despite being conceptually simple, it is a very powerful way to understand which (and how) of a number of candidate variables, sometimes referred to covariates, independent or explanatory variables, might influence a dependent variable, also often referred as the response. There are many flavours of regression models, from a simple linear regression to complicated generalized additive mixed models. We do not wish to present these in any detail, but to introduce you to some functions that implement these models and the syntax that R uses to describe them.

Let's start with the basics. You have used the `cars` data set above. We use it here again to try to explain the distance a car takes to stop as a function of its speed. We start with a linear model using function `lm()`:

```r
data(cars)

mylm1 <- lm(dist ~ speed, data = cars)
```

We have stored the result of fitting the model in object `mylm1`. The function `summary()` can be used to print a summary of the fit:

```r
summary(mylm1)
```

```
##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -29.069  -9.525  -2.272   9.215  43.201
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) -17.5791     6.7584  -2.601   0.0123 *
## speed         3.9324     0.4155   9.464 1.49e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.38 on 48 degrees of freedom
## Multiple R-squared:  0.6511, Adjusted R-squared:  0.6438
## F-statistic: 89.57 on 1 and 48 DF,  p-value: 1.49e-12
```

Don't be frightened by all of the output. The coefficient associated with speed tells us what intuition alone would anticipate: the higher the speed, the larger the distance a car takes to stop. The easier way to see the relationship is by adding a line to the plot (note this is a similar plot to what you should have created in task 3 above!).

```r
xl <- "Speed (mph)"

yl <- "Distance (m)"

plot(cars$speed, cars$dist,
```

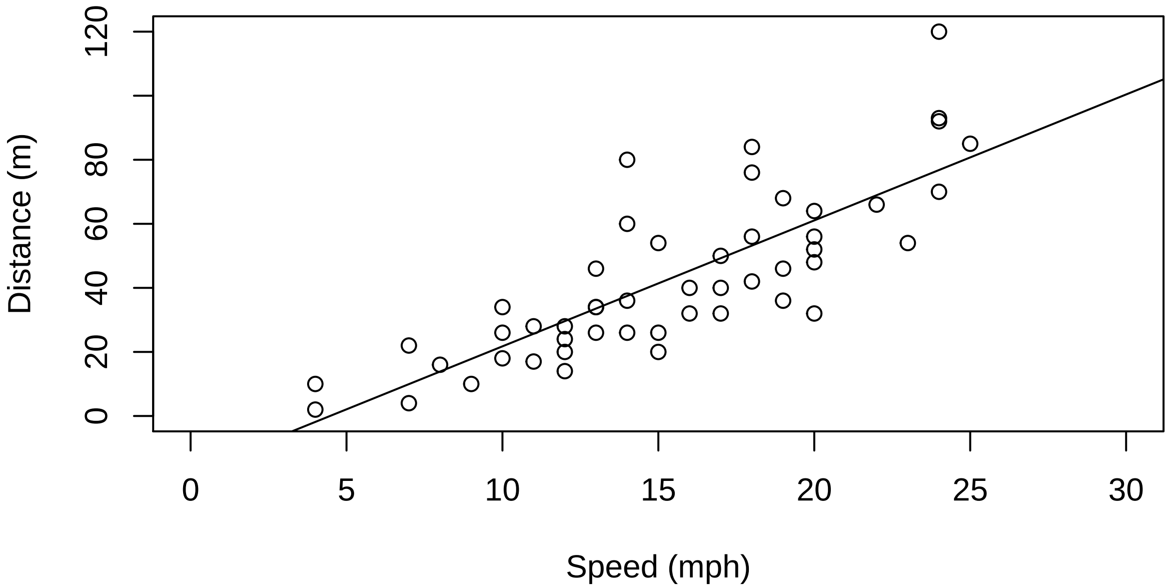


Figure 1: The data and the linear fit added to it.

```
     xlab = xl, ylab = yl,
     ylim = c(0, 120), xlim = c(0, 30))

abline(mylm1)
```

Note how function `abline()` is used with a linear model as its first argument and it uses the parameters in said object to add a line to the plot. The optional arguments `v` and `h` are often very useful to draw vertical and horizontal lines in plots.

Task 7.2.1: Use abline to draw dashed lines (tip, use optional argument `lty=2`) representing the estimated distance that a car moving at 16 mph would take to stop.

Note that the line added to the plot represents the distance a car would take to stop given its speed. Oddly enough, it seems like a car going at 3 mph might take a negative time to stop, which is just plain nonsense. Why? Because we used a model which does not respect the features of the data. A stopping distance can not be negative. However, implicit in the linear model we used, distance is a Gaussian (=normal) random variable. We can avoid this by using a generalized linear model (GLM). Now the response can have a range of distributions. An example of such distribution that takes only positive values is the gamma distribution. We implement a gamma GLM next.

Exercise 7.3: Genearlized Linear Model

```
#fit the glm
myglm1 <- glm(dist ~ speed, data = cars, family = Gamma(link = log))
```

To generate model predictions, we can use the function `predict()` or `predict.glm()`. As indicated in the help file, it is possible to simply predict over the original values of the covariates using `predict(myglm1)`. By default, the predictions are given on the scale of the link function; if we want to see predictions on the scale of the response we can include the argument `type = "response"`. We can supply the model with novel sets of covariates over which to predict. For example, the dataset didn't include any observations of cars at a speed of 5. If we wanted to know the predicted stopping distance at a speed of 5, we could use:

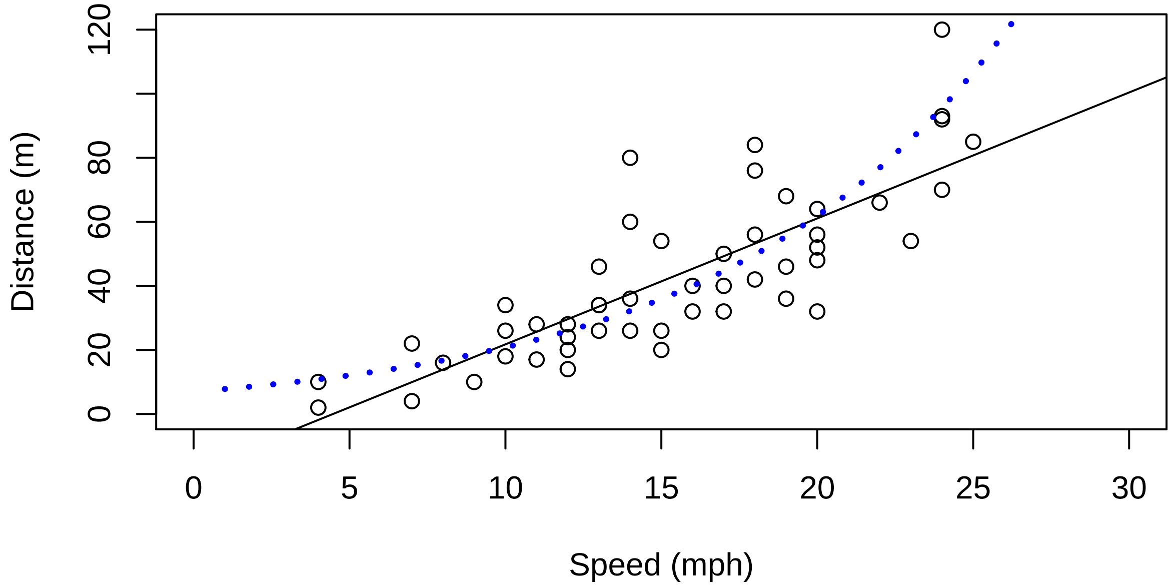


Figure 2: The data and the linear and Gamma glm fits added to it.

```r
predict.glm(myglm1,
            newdata = data.frame(speed = 5),
            type = "response")
```

```
##        1
## 12.07056
```

Notice that although we only want to know the predicted distance for a single speed, we still need to provide that speed within a data frame. This makes more sense when predicting over larger sets of covariates, for example, predicting for speeds from 1 to 30:

```r
predmyglm1 <- predict.glm(myglm1,
                          newdata = data.frame(speed = 1:30),
                          type = "response")
```

Our model now assumes the response has a gamma distribution, and the link function is the logarithm. The link function allows you to change how the mean value is related to the covariates. This becomes rather technical rather fast. Details about GLMs are naturally beyond the scope of this tutorial.

```r
#create a plot
plot(cars$speed,cars$dist,
     xlab = "Speed (mph)", ylab = "Distance (m)",
     ylim = c(0,120), xlim = c(0,30))

#add the linear fit
abline(mylm1)

#and now add the glm predictions
lines(1:30, predmyglm1, col="blue", lwd=3, lty=3)
```

However, this GLM still requires that the response is linear at some scale (in this case, on the scale of the link function). Sometimes, non-linear effects are present. These can be fitted using generalized additive models.

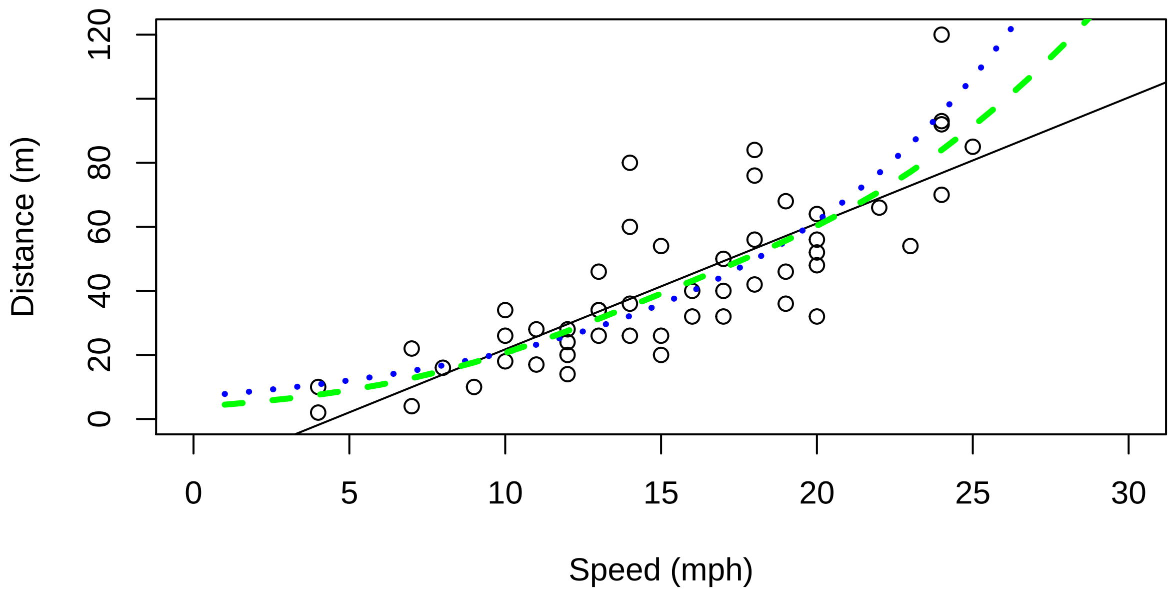


Figure 3: The data and the linear and Gamma glm and gam fits added to it.

Exercise 7.4: Generalized Additive Model

So finally we fit a GAM model to the same data set. For that we require library `mgcv`. Here the fit is not very different from the GLM fit, but under many circumstances a GAM might be more appropriate than a GLM

```
#load the mgcv library
library(mgcv)
```

```
## Loading required package: nlme

## This is mgcv 1.8-28. For overview type 'help("mgcv-package")'.
```

```
#fit the gam
mygam1 <- gam(dist ~ s(speed), data = cars,
              family = Gamma(link = log))

#predict using the glm for speeds between 1 and 30
predmygam1 <- predict(mygam1,
                      newdata = data.frame(speed = 1:30),
                      type="response")
```

```
#create a plot
plot(cars$speed, cars$dist,
     xlab = "Speed (mph)", ylab = "Distance (m)",
     ylim = c(0, 120), xlim = c(0, 30))

#add the linear fit
abline(mylm1)

#and now add the glm predictions
lines(1:30, predmyglm1, col = "blue", lwd = 3, lty = 3)
lines(1:30, predmygam1, col = "green", lwd = 3, lty = 2)
```

Material for this tutorial was contributed by Louise Burt, Jessica Carrière-Garwood, Danielle Harris, Eiren Jacobson, Tiago Marques, & Len Thomas