R: A Quick Start Guide

Table of Contents

R: A Quick Start Guide[2]

1 Why R?

R provides functionality that exceeds other statistical packages; and, it is free with a large user community and thousands of available packages. R is a programming language designed for the manipulation, analysis, description and visual presentation of data. It is command-based, not menu driven. New users who have limited experience with other programming languages face a steep learning curve. Trial and error will be necessary. When you do something wrong in **R**, it will give you an error message. These messages can be cryptic requiring research. Learning **R** necessitates use of available resources such as text books, other students, the internet and instructors. This document provides an introduction to the **R** language. It is not intended to be comprehensive. It addresses basics of data manipulation, computation and data visualization. Use this document as an introductory workbook. Examples and solved problems are provided.

2 Installing R

R is open-source software and may be downloaded freely at *http://cran.r-project.org/*. Everything required by this document can be accomplished in the **R** environment with a plain text editor. However, many users choose to work in RStudio, an integrated development environment (IDE) for R. RStudio is used for the examples in this document. It is convenient and easy to use. Install the most current version of **R** and RStudio.

Step 1: Download and install R from http://cran.r-project.org/. Documentation is readily available on CRAN and on many R community and resource sites.

Step 2: Download and install RStudio. http://www.rstudio.com/products/rstudio/download/

There are many facilities in **R** and thousands of software packages that can be downloaded. There is also the facility to 'bolt-on' additional libraries of functions that have a specific utility. Typing

> library() will give a list and short description of the libraries available. Typing
> library(libraryname), where libraryname is the name of the required library will give you access to the functions in that library.

---

[1]We adopt the convention of using typeface font to denote things typed in **R**. The > sign is not typed in **R** script or the console. It denotes the prompt symbol in the examples which follow. In this document, when the prompt symbol appears in an example, that example was copied from output in the **R** console.

[2]Adapted from https://www.nceas.ucsb.edu/files/scicomp/Dloads/RProgramming/BestFirstRTutorial.pdf.

3 Getting Started with R

R stores information and operates on ***objects***. The simplest objects are ***scalars, vectors*** and ***matrices***. But there are many others such as ***lists*** and ***dataframes***. It is also possible to define new types of objects, specific for a particular application. Open up a workspace, and work through this document at the computer. Type in all of the commands in the following examples making sure you understand how they operate. The examples are from RStudio. Then try the exercises at the end of each section. Solutions are provided separately.

When you start a session in **R**, the workspace is the current **R** working environment. Commands may entered interactively in the console at the **R** user prompt. It is also possible to write script and execute this script. **R** is case-sensitive. Commands written in **R** are saved in memory throughout the session. You can scroll back to previous commands typed by using the 'up' arrow key (and 'down' to scroll back again). You can also 'copy' and 'paste' using standard windows editor techniques (for example, using the 'copy' and 'paste' dialog buttons). As an alternative you might copy and paste commands manually into a notepad editor or something similar. At the end of an **R** session, you can save an image of the current workspace that is automatically reloaded the next time **R** is started.

4 Managing Your Workspace and Files

It is important to be aware of where your work is kept. It is possible to keep different projects in different physical directories. For the purposes of this guide, and subsequent course work, it is advisable to establish one working directory. Let's assume you name your working directory Predict401. The following commands are examples that allow you to set this directory, verify the working directory, and list the contents. These and other examples can be found at http://www.statmethods.net/index.html thanks to Rob Kabacoff, author of *R in Action.*

```
setwd(Predict401)     # change to Predict401
setwd("c:/docs/Predict401")  # note / instead of \ in windows
getwd() # print the current working directory or cwd
ls()   # list the objects in the current workspace
rm(list=ls())   #removes all objects in the current workspace
```

When you work in **R,** objects created are stored in the current workspace (image). Each object created remains in the image unless you explicitly delete it. The workspace will be lost at the session's end unless you save it. Here are commands to help you manage your workspace.

```
# save the workspace to the file .RData in the cwd
save.image()

# to save specific objects, if you don't specify the path, the cwd is assumed
```

```
save(object list,file="mywork.RData")
```

```
# load a workspace into the current session
# if you don't specify the path, the cwd is assumed
```

```
load("mywork.RData")
```

```
q() # quit R. You will be prompted to save the workspace.
```

There are a variety of ways to import data into R depending on whether the file is from Excel, SAS, SPSS or somewhere else. In this course we will be dealing with comma delimited text files denoted by .csv. If the file is resident on your computer, here are two ways to load such a file into your workspace. A special case of the read.table() command is the read.csv() command shown below. Either can be used in this course.

```
# first row contains variable names, comma is separator
# assign the variable id to row names
# note the / instead of \ on mswindows systems
```

```
mydata <- read.table("c:/mydata.csv", header=TRUE,  sep=",", row.names="id")
```

```
mydata <- read.csv(file.path("c:/R401/", "mydata.csv"), sep=",")
```

At some point you will need to save a data file. You can save it to your working directory using the first commands shown below, or to some other file location. These statements are intended for comma delimited files.

```
write.table(mydata, "mydata.csv") # Saves to current directory
```

```
write.table(mydata, file.path("c:/R401/", "mydata.csv"))
```

```
write.csv(mydata, file.path("c:/R401/", "mydata.csv"))
```

5 Working with Vectors

For example, type and enter: 4+6. **R** does scalar arithmetic returning the scalar value 10.

```
[1] 10
```

In actual fact, **R** returns a vector of length 1, hence the [1] denoting first element of the vector. We can assign values to objects for subsequent use:

```
x <- 6
y <- 4
z <- x+y
```

We can look at the contents of the object z by typing and entering its name (or using print(z)):

```
z
[1]  10
```

At any time we can list the objects we have created by using the list objects function ls(). **R** will return the following along with other objects you have defined:

```
>  ls()
[1]  "x"                "y"                 "z"
```

Notice that ls() is a function as well as an object. Typing ls would result in a display of the contents of this object, in this case, the commands of the function. The use of parentheses, ls(), ensures that the function is ***executed*** and its result, a list of the objects in the directory, is displayed. To completely clear the workspace use the command rm(list=list()). Enter ls() to verify character(0) is displayed which indicates the workspace has been cleared.

V ectors can be created in **R** in a number of ways as shown with z. We can print all of the elements to verify. The commands above result in the following output appearing in the console. The statement print(z) can also be used.

```
> z <- c(5,9,1,0)
> z
[1] 5 9 1 0
```

Note the use of the function c() to ***concatenate*** or 'glue together' individual elements. The following example leads to the same result by gluing together two vectors to create a single vector.

```
> x <- c(5,9)
> y <- c(1,0)
> z <- c(x,y)
> z
[1] 5 9 1 0
```

The length, or number of elements in z, can be determined using the length() function. Since the elements of z are numeric, their arithmetic total can be obtained using sum().

```
> length(z)
[1] 4
> sum(z)
[1] 15
```

Sequences can be generated.

```
> x <- 1:10
> x
```

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

More general sequences can be generated using the seq() command. For example:

```
> seq(1,9,by=2)
[1] 1 3 5 7 9
```

and

```
> seq(8,20,length=6)
[1]  8.0 10.4 12.8 15.2 17.6 20.0
```

These examples illustrate that many functions in **R** have optional arguments. The help() function is useful for determining what these options are. In this case, either the step length or the total length of the sequence works. If you leave out both of these options, **R** will make its own default choice, in this case assuming a step length of 1. So, for example,

```
> x <- seq(1,10)
> x
[1]  1  2  3  4  5  6  7  8  9 10
```

Another useful function for building vectors is the rep command for repeating things.

```
> rep(0,20)
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
> rep(1:3,6)
 [1]  1  2  3  1  2  3  1  2  3  1  2  3  1  2  3  1  2  3
```

Notice a variation on the use of this function,

```
> rep(1:3,c(6,6,6))
 [1]  1  1  1  1  1  1  2  2  2  2  2  2  3  3  3  3  3  3
```

which we could also simplify as:

```
> rep(1:3,rep(6,3))
 [1]  1  1  1  1  1  1  2  2  2  2  2  2  3  3  3  3  3  3
```

As explained above, **R** will often adapt to the objects it is asked to work on. For example:

```
> x <- c(6,8,9)
> y <- c(1,2,4)
>  x + y
```

```
[1] 7 10 13

>  x * y
[1]    6 16 36
```

The previous example shows that **R** uses element-wise arithmetic on vectors. **R** will also try to make sense if objects are mixed. For example,

```
> x <- c(6,8,9)
> x + 2
[1]  8 10 11
```

Care should be taken to make sure that **R** is doing what you wish. A function will also operate element-wise on an object. For example, sqrt() calculates the square root of the vector z.

```
> z <- c(16,4)
> sqrt(z)
[1] 4 2
```

The append() command can be used to merge vectors by introducing elements. Put 3 and 2 between 9 and 1. The position of 9 in the sequence must be designated. It is 2 in this example.

```
> z
[1] 5 9 1 0
> append(z, c(3,2), after = 2)
[1] 5 9 3 2 1 0
```

Non-numeric elements may also be merged using the append() command. Put “A” and “B” between “y” and “z”. The position of 1 in the sequence must be designated (i.e. after = 2).

```
> s <- c("x", "y", "z")
> append(s, c("A", "B"), after = 2)
[1] "x" "y" "A" "B" "z"
```

If you don’t know how to use a function, or don’t know what the arguments (options) or default values are, type help(**functionname**) where **functionname** is the name of the function you are interested in. You can also type **?functionname()** into the console and obtain information. This information may not be sufficient at which point further research will be necessary.

Exercises

1. Define

   ```
   x <- c(4,2,6)
   y <- c(1,0,-1)
   ```

   Decide what the result will be of the following and use **R** to check your answers.

   (a) length(x)
   (b) sum(x)
   (c) sum(x^2)
   (d) x + y
   (e) x * y
   (f) x - 2
   (g) x^2

2. Decide what the following sequences are and use **R** to check your answers:

   (a) seq(2,9)
   (b) seq(4,10,by=2)
   (c) seq(3,30,length=10)
   (d) seq(6,-4,by=-2)

3. Determine what the result will be of the following **R** expressions, and then use **R** to check:

   (a) rep(2,4)
   (b) rep(c(1,2),4)
   (c) rep(c(1,2),c(4,4))
   (d) rep(1:4,4)
   (e) rep(1:4,rep(3,4))

4. Use the rep function to define the following vectors in **R**.

   (a) 6,6,6,6,6,6
   (b) 5,8,5,8,5,8,5,8
   (c) 5,5,5,5,8,8,8,8

6 Summary Statistics

Let's suppose we've collected some data from an experiment and stored them in an object x. Some simple summary statistics of these data can be produced. The which.max() and which.min() functions identify the location of the minimum and maximum values. The quantile function is useful for identifying the associated percentile location, in this case the 25th percentile and the 75th percentile (known as the first and third quartiles. The median is the second quartile):

```
x <- c(7.5,8.2,3.1,5.6,8.2,9.3,6.5,7.0,9.3,1.2,14.5,6.2)

> mean(x)
[1] 7.216667
> median(x)
[1] 7.25
> min(x)
[1] 1.2
> max(x)
[1] 14.5
> which.min(x)
[1] 10
> which.max(x)
[1] 11
> quantile(x, probs = c(0.25,0.75))
  25%   75% 
6.050 8.475 
```

The summary() function determines many of these values directly.

```
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  1.200   6.050   7.250   7.217   8.475  14.500 
```

Note that summary() rounded the mean from 7.216667 to 7.217. The round() function will round to a specified number of decimal places, whereas signif() will round to a specified number of digits.

```
> round(mean(x), digits = 3)
[1] 7.217
> signif(mean(x), digits = 3)
[1] 7.22
```

There are other functions that can be useful such as ceiling() which finds the smallest integer greater than a value, floor() which finds the largest integer less than a value. For example:

```
> mean(x)
[1] 7.216667
> ceiling(mean(x))
[1] 8
> floor(mean(x))
[1] 7
```

It may be, that we subsequently learn that the first 6 data values in x correspond to measurements made on one machine, and the second six on another machine. Summarizing the two sets of data separately, we would need to extract from x the two relevant subvectors. This is achieved by subscripting:

```
x <- c(7.5, 8.2, 3.1, 5.6, 8.2, 9.3, 6.5, 7.0, 9.3, 1.2, 14.5, 6.2)

> x[1:6]
[1] 7.5 8.2 3.1 5.6 8.2 9.3
> x[7:12]
[1]  6.5  7.0  9.3  1.2 14.5  6.2

> summary(x[1:6])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.100   6.075   7.850   6.983   8.200   9.300
> summary(x[7:12])
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.200   6.275   6.750   7.450   8.725  14.500
```

Other subsets can be created by identifying the subscripts for the values of interest.

```
>   x[c(2,4,9)]
[1] 8.2 5.6 9.3
```

Negative integers can be used to *exclude* particular elements. For example x[-(1:6)] has the same effect as x[7:12].

```
> x[-(1:6)]
[1]  6.5  7.0  9.3  1.2 14.5  6.2
```

Exercises

1. x <- c(5,9,2,3,4,6,7,0,8,12,2,9) Decide what each of the following is. Check with **R.**

    (a) x[2]
    (b) x[2:4]
    (c) x[c(2,3,6)]
    (d) x[c(1:5,10:12)]
    (e) x[-(10:12)]

2. The data y <- c(33,44,29,16,25,45,33,19,54,22,21,49,11,24,56) contain sales of milk in litres for 5 days in three different shops (the first 3 values are for shops 1,2 and 3 on Monday, etc.) Produce a statistical summary of the sales for each day of the week and also for each shop.

7 Logical Comparisons

Logical values (sometimes called ***logicals***) are a non-numeric data type often created via comparison between variables. Relational operators are typically used on numeric values to determine if a comparison is TRUE or FALSE. The table below lists these relational operators.

Operator	**Interpretation**
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
!x	Not x
x | y	x or y
x & y	x and y

Logicals allow you to test the equivalence of different R objects in typical mathematical fashion to produce TRUE or FALSE results. Such comparisons are often used as conditions for performing further operations using if-then, else statements. While some programming languages require you to type “then” in order to actually execute the second part of an if-statement, R does not require this. Suppose you want to determine if a variable is less than or equal to zero and take note of this condition.

```
> x <- -1
> if (x > 0)  signal <- "FAIL" else
+ if (x <= 0) signal <- "SUCCESS"
> signal
[1] "SUCCESS"
```

These logicals can also be combined with the AND & or OR | operators to produce logicals that meet more than one criterion. Note that the symbol for OR is a vertical bar |, and not is the exclamation symbol !. In the following example, we introduce the & symbol with an additional condition.

```
> x <- -1
> if (x > 0)  signal <- "FAIL" else
+   if (x <= 0 & x > -2) signal <- "SUCCESS" else
+     signal <- "FAIL"
> signal
[1] "SUCCESS"
```

One application of logicals is determining the location of an element in a vector. Suppose we wish to locate the value 8.2 in the object x, and to count the number of occurrences. The sum() function treats TRUE as 1 and FALSE as 0. The sum() function can be used as a counter showing 8.2 appears twice. Substituting location into x produces 8.2 twice. We can also use the which() function to return the locations(s), element-wise, of a value.

```
> x <- c(7.5,8.2,3.1,5.6,8.2,9.3,6.5,7.0,9.3,1.2,14.5)
> location <- x == 8.2
> location
 [1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
> sum(location)
[1] 2
> x[location]
[1] 8.2 8.2

 > which(x == 8.2)
[1] 2 5
```

Similar operations are possible with non-numeric elements.

```
 > x <- rep(c("A","B","C"), 3)
 > x
```

```
[1] "A" "B" "C" "A" "B" "C" "A" "B" "C"
> location <- x == "B"
> location
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

R is capable of handling missing values. Missing data in **R** appears as NA. NA is not a string or a numeric value, but an indicator of a missing value. We can create vectors with missing values.

```
x1 <- c(1, 4, 3, NA, 7)
x2 <- c("a", "B", NA, "NA")
```

NA is one of the few non-numbers that does not generate an error. In x2, the third value is missing while the fourth value is the character string "NA". To see which values in each of these vectors **R** recognizes as missing, we can use the **is.na** function.

```
> is.na(x1)
[1] FALSE FALSE FALSE TRUE FALSE

> is.na(x2)
[1] FALSE FALSE TRUE FALSE
```

R distinguishes between the NA and "NA" in x2, NA is seen as a missing value, "NA" is not.

Exercises

1. For the data y <- c(33,44,29,16,25,45,33,19,54,22,21,49,11,24,56),   Find the minimum and maximum of y and their location in the vector.
2. Find the median of y and use logicals to split y into two subsets. One subset will have all values in y strictly less than the median, and the other subset all values strictly greater than the median. Print the resulting subsets.

8 Matrices

Matrices can be created in **R** in a variety of ways. Perhaps the simplest is to create the columns and then glue them together with the function cbind(). (There is a similar function, rbind(), for building matrices by gluing rows together.) For example,

```
> x <- c(5,7,9)
> y <- c(6,3,4)
> z <- cbind(x,y)
```

The dimension of a matrix can be checked with the dim command showing 3 rows and 2 columns.

```
> dim(z)
[1]  3  2
```

The functions cbind and rbind can also be applied to matrices themselves (provided the dimensions match) to form larger matrices. Forexample,

```
> rbind(z,z)

     x y
[1,] 5 6
[2,] 7 3
[3,] 9 4
[4,] 5 6
[5,] 7 3
[6,] 9 4
```

Matrices can also be built by explicit construction via the function matrix. Notice that the dimension of the matrix is determined by the size of the vector and the requirement that the number of rows is 3, as specified by the argument nrow=3. As an alternative we could have specified the number of columns with the argument ncol=2 (obviously, it is unnecessary to give both). Notice that the matrix is 'filled up' column-wise.

```
z <- matrix(c(5,7,9,6,3,4),nrow=3)
```

If instead you wish to fill up row-wise, add the option byrow=T. For example, with z as above,

```
> z <- matrix(c(5,7,9,6,3,4), nr=3, byrow=T)
> z
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
```

Notice that the argument nrow has been abbreviated to nr. Such abbreviations are always possible for function arguments provided it induces no ambiguity. If in doubt always use the full argument name. The number of rows and columns can be determined by nrow() and ncol(). Note how they work on z.

```
> nrow(z)
[1] 3
> ncol(z)
[1] 2
```

R will try to interpret operations on matrices in a natural way. For example, with z as above,

```
> y <- matrix(c(1,3,0,9,5,-1),nrow=3,byrow=T)
> y
     [,1] [,2]
[1,]    1    3
[2,]    0    9
[3,]    5   -1

> z + y
     [,1] [,2]
[1,]    6   10
[2,]    9   15
[3,]    8    3

> z*y
     [,1] [,2]
[1,]    5   21
[2,]    0   54
[3,]   15   -4
```

Multiplication here is element-wise rather than conventional matrix multiplication. Indeed, conventional matrix multiplication is undefined for y and z as the dimensions fail to match.

```
> x <- matrix(c(3,4,-2,6),nrow=2,byrow=T)
> x
     [,1] [,2]
[1,]    3    4
[2,]   -2    6
```

Matrix multiplication is expressed using notation %*%:

```
> y%*%x
     [,1]  [,2]
[1,]   -3   22
[2,]  -18   54
[3,]   17   14
```

Useful functions on matrices are **t()** to calculate a transpose and solve() to calculate inverses which can be useful for solving systems of linear equations.

```
> t(z)
     [,1] [,2] [,3]
[1,]    5    9    3
[2,]    7    6    4
```

```
> solve(x)

           [,1]       [,2]
[1,] 0.23076923 -0.1538462
[2,] 0.07692308  0.1153846
```

Operations can be combined. For example, calculate the identity matrix with rounding.

```
> round(x %*% solve(x), digits = 2)
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

As with vectors it is useful to be able to extract sub-components of matrices. In this case, we may wish to pick out individual elements, rows or columns. The [ ] notation is used to subscript. The following examples using z should make things clear:

```
>
z[1,1]
[1]  5

> z[c(2,3),2]
[1]  6  4

> z[,2]
[1]  7  6 4

> z[1:2,]
     [,1] [,2]
[1,]    5    7
[2,]    9    6
```

It is necessary to specify which rows and columns are required, whilst omitting the integer for either dimension implies that every element in that dimension is selected.

We can also transform this matrix into a different type of R object called a ***data.frame***. Matrices and data.frames have some similar properties, but also differences that you will learn. Importantly, most data that is read in from an outside file (as opposed to typing in ourselves) will be in the form of a data.frame. If you want to know whether something is a matrix or a data.frame, you can also test it using the is.matrix()or is.data.frame()command.

Exercises

1. Create in **R** the matrices $x = \begin{bmatrix} 3 & 2 \\ -1 & 1 \end{bmatrix}$ and $y = \begin{bmatrix} 1 & 4 \\ 0 & 1 \end{bmatrix}$.

2. Calculate the following and check your answers in R.
   a) 2*x
   b) x*x
   c) x%*%x
   d) x%*%y
   e) solve(x)
   f) x[1,]
   g) x[,2]

9 Accessing Example Datasets

R includes a number of datasets that are convenient to use for examples. You can get a description of what's available by typing data(). To access any of these datasets, type **data(dataset**) where **dataset** is the name of the dataset.

```
> data(trees)
> trees[1:3,]

  Girth Height Volume
1  8.3    70   10.3
2  8.6    65   10.3
3  8.8    63   10.2
```

This gives the first 3 rows of these data, and we can now see that the columns represent measurements of girth, height and volume of trees (actually cherry trees: see help(trees)) respectively. If we want to work on the columns of these data, we can use the subscripting. For example, trees[,2] gives all of the heights. There are several alternatives.

```
> mean(trees[,2])
[1] 76
```

```
> mean(trees$Height)
[1] 76

>  Height <- trees$Height
>
mean(Height)
[1] 76
```

Exercises

1. Use the dataset mtcars and find the mean weight and mean fuel consumption for vehicles in the dataset.   (type help(mtcars) for a description of the variables available).

10 The apply() Function

A common situation is where we want to apply the same function to every row or column of a matrix. For example, we may want to find the mean value of each variable in the trees dataset. Obviously, we could operate on each column separately. This is inefficient. The function apply() simplifies things. It is easiest understood by example. First, we determine the structure of trees using str() and then check the first 5 lines of the data.frame.

```
> data(trees)
> str(trees)

'data.frame':	31 obs. of  3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
```

As shown below, apply() has the effect of calculating the median of each column (dimension 2) of trees. We'd have used a 1 instead of a 2 if we wanted the median of everyrow. Any function can be applied in this way, though if optional arguments to the function are required these need to be specified as well.

```
> apply(trees,2,median)
 Girth Height Volume
  12.9   76.0   24.2
```

Note the use of mean() in combination with round(). See help(apply) for further details.

```
> round(apply(trees,2,mean), digits = 2)

 Girth Height Volume
 13.25  76.00  30.17
```

Exercise

1. Repeat the analyses shown above using the datasets quakes and mtcars using apply().

11 The aggregate() and table() Functions

Different formats can take advantage of grouping variables. The data ToothGrowth has three variables: len (tooth length), supp (Supplement type (VC or OJ**)** and dose (numeric dose in milligrams/day). Suppose we want the mean values of len for each combination of dose and supp. Since dose enters as a numeric at three levels, these levels can be used as labels. For purposes of illustration, it will be converted to a factor variable with levels “0.5”, “1.0” and “2.0” using the factor() function. The quotation marks denote a non-numeric variable.

```
> data(ToothGrowth)
> ToothGrowth$dose <- factor(ToothGrowth$dose)
> str(ToothGrowth)

'data.frame':	60 obs. of  3 variables:
 $ len : num  4.2 11.5 7.3 5.8 6.4 10 11.2 11.2 5.2 7 ...
 $ supp: Factor w/ 2 levels "OJ","VC": 2 2 2 2 2 2 2 2 2 2 ...
 $ dose: Factor w/ 3 levels "0.5","1","2": 1 1 1 1 1 1 1 1 1 1 ...

> aggregate(ToothGrowth$len, by = list(ToothGrowth$supp, ToothGrowth$dose), mean)

  Group.1 Group.2     x
1      OJ     0.5 13.23
2      VC     0.5  7.98
3      OJ       1 22.70
4      VC       1 16.77
5      OJ       2 26.06
6      VC       2 26.14
```

The coding can be simplified using the with() function. The following code will produce the table shown above: with(ToothGrowth, aggregate(len, by = list(supp, dose), mean)). It is possible to rename the columns for clarity using the colnames() function as shown below.

```
> result <- with(ToothGrowth, aggregate(len, by = list(supp, dose), mean))
> colnames(result) <- c("supp", "dose", "mean")
> result
```

```
  supp dose  mean
1   OJ  0.5  13.23
2   VC  0.5   7.98
3   OJ  1.0  22.70
4   VC  1.0  16.77
5   OJ  2.0  26.06
6   VC  2.0  26.14
```

Sample sizes may be desired for each combination of supp and dose. The table() function is applicable here. The addmargins() and with() functions may also be used as desired.

```
> addmargins(table(ToothGrowth$supp, ToothGrowth$dose))

      0.5  1  2  Sum
OJ     10 10 10   30
VC     10 10 10   30
Sum    20 20 20   60
```

Exercises

1. Use aggregate to determine the median len for each combination of supp and dose.

2. Use with() and addmargins() to produce the table of counts for the ToothGrowth data.

12 Loops

An important piece of **R** programming is looping. Like apply() and aggregate(), loops allow you to perform repetitive tasks on the same or different data. Loops, apply() and aggregate set **R** apart from SPSS, SAS, and Stata. There are several different types of loops that can be used. In this section "for" loops, and "while" loops will be discussed

The "for" loop tells **R** that we want to conduct a task a specified number of times. The format is simple:

```
for (i in values){
        …program statements…
}
```

Here *values* is a vector that gives the identifier "i" particular values to use for looping. The code between the parentheses must be executed for each value "i" assumes. There is nothing special about using "i". Other symbols may be used as shown below. Suppose we want to sum the integers from 1 to 10 and print the sum. Each subsequent value of k is added to the sum until k exceeds 10.

```
> sum <- 0
> for (k in 1:10){
+   sum <- sum + k
+ }
> sum
[1] 55
```

A “while” loop is different. It continues computing as long as the loop condition is satisfied. In the example, when k exceeds 10, the loop condition is not satisfied and the iteration ceases.

```
> sum <- 0
> k <- 1
> while(k <= 10){
+   sum <- sum + k
+   k <- k + 1
+ }
> sum
[1] 55
```

Exercises

1. Write a “for” loop to compute the value of a factorial for integers greater than zero. Execute the loop for an integer equal to 5, and print the factorials for 1, 2, 3, 4 and 5. Repeat the above, but with a “while” loop.

13 Writing functions

R has the facility to extend the language with user-supplied functions. For example, there is no function for calculating the percent coefficient of variation. We can define such a function. (For simplicity, we assume the mean value in the calculation will never be zero.)

```
cv <- function(x){
  100*sd(x)/mean(x)
}
```

```
> girth <- trees[,1]
> cv(girth)
[1] 23.68695
```

User defined functions can be used the same as library functions. Suppose we want the percent coefficient of variation for Girth, Height, and Volume from the tree data. This can be done efficiently using the apply function with the cv function just defined.

```
> apply(trees,2,cv)

   Girth      Height    Volume
23.686948  8.383964 54.482331
```

Exercises

1. Write a simple function that computes the sample variance for a vector of numerical values. Use this function with apply() to compute the sample variance for the dimensions in the tree data. Compare your results to those computed with var().

14 Statistical Computation, Simulation and Random Sampling

Many of the tedious statistical computations that would once have had to have been done from statistical tables can be easily carried out in **R**. **R** provides a library of many statistical distributions. Some of the more commonly used distributions include the normal, binomial, Poisson, t, F, Chi-squared, uniform, and hypergeometric. There are functions in **R** to evaluate the density function, the distribution function and the quantile function (the inverse distribution function). There are also functions to generate random samples from these distributions.

Using the normal distribution as an example, these functions are, respectively, dnorm, pnorm, qnorm and rnorm. Using help() in **R** can serve as a reminder of the necessary arguments for the use of the various probability functions. Let's take as an example the normal distribution. Unlike with tables, there is no need to standardize the variables first. For example, suppose $X \sim N(3, 2^2)$ (this stands for a normal variable X with mean 3 and standard deviation 2). The density function dnorm(x, mean = 3, sd = 2) will evaluate the density at points contained in the vector x (note, dnorm will assume mean 0 and standard deviation 1 unless these are specified). Note also that the function assumes you will give the standard deviation rather than the variance. As an example, the following shows the density of the $N(3,9)$ distributiom at $x = 5$. The variance is 9 which results in a population standard deviation of 3.

```
> dnorm(5, mean = 3, sd = 3)
[1] 0.1064827
```

The following calculates the density function of the same distribution at intervals of 0.1 over the range $[-5, 10]$. The functions pnorm and qnorm work in an identical way.

```
x <- seq(-5,10,by=.1)
dnorm(x, mean = 3, sd = 2)
```

Similar functions exist for other distributions. For example, dt(), pt() and qt() for the *t*-distribution, though in this case it is necessary to give the degrees of freedom rather than the mean and standard deviation. Other distributions available include the binomial, exponential, Poisson and hypergeometric. Care is needed interpreting the functions for discrete variables. Here some background reading and experimentation are needed. Use help for further information.

One further important technique for many statistical applications is the simulation of data from specified probability distributions. **R** enables simulation from a wide range of distributions, using a syntax similar to the above. To simulate 100 random observations from the *N* (3, 4) distribution, for example, we write rnorm(100,3,2).

Similarly, rt, rpois for simulation from the *t* and Poisson distributions, etc. Closely related to this is sampling from a data file. Random samples and sequential samples can be drawn with similar approaches. A vector needs to be generated that identifies the rows being selected. Using trees as an example, if we wanted to pick 10 observations at random, the following statements using the sample() function would suffice. In this instance, sample() will randomly select 10 integers between 1 and nrow(trees) without replacement. These integers can then be used to select the corresponding rows from trees as shown below.

```
> set.seed(123)
> index <- sample(1:nrow(trees), 5, replace = FALSE)
> trees[index,]

    Girth Height Volume
 9   11.1     80     22.6
24   16.0     72     38.3
12   11.4     76     21.0
25   16.3     77     42.6
26   17.3     81     55.4
```

For a systematic sample, if we wanted every third observation in trees. The index would be:

```
> index <- seq(1,nrow(trees), by=10)
> trees[index,]

    Girth Height Volume
1     8.3    70   10.3
11   11.3    79   24.2
21   14.0    78   34.5
31   20.6    87   77.0
```

Exercises

1. Suppose $X \sim N(2, 0.25)$. Denote by f and F the density and distribution functions of X respectively. Use **R** to calculate:
   a) The density function at 0.5, $f(0.5)$ (use dnorm())
   b) The distribution function at 2.5, $F(2.5)$ (use pnorm())
   c) The 95th percentile $F^{-1}(0.95)$ (use qnorm())
   d) The probability that X is between 1 and 3, Pr(1 <= X <= 3)
2. Repeat question 1 in the case that X has a t-distribution with 5 degrees of freedom.

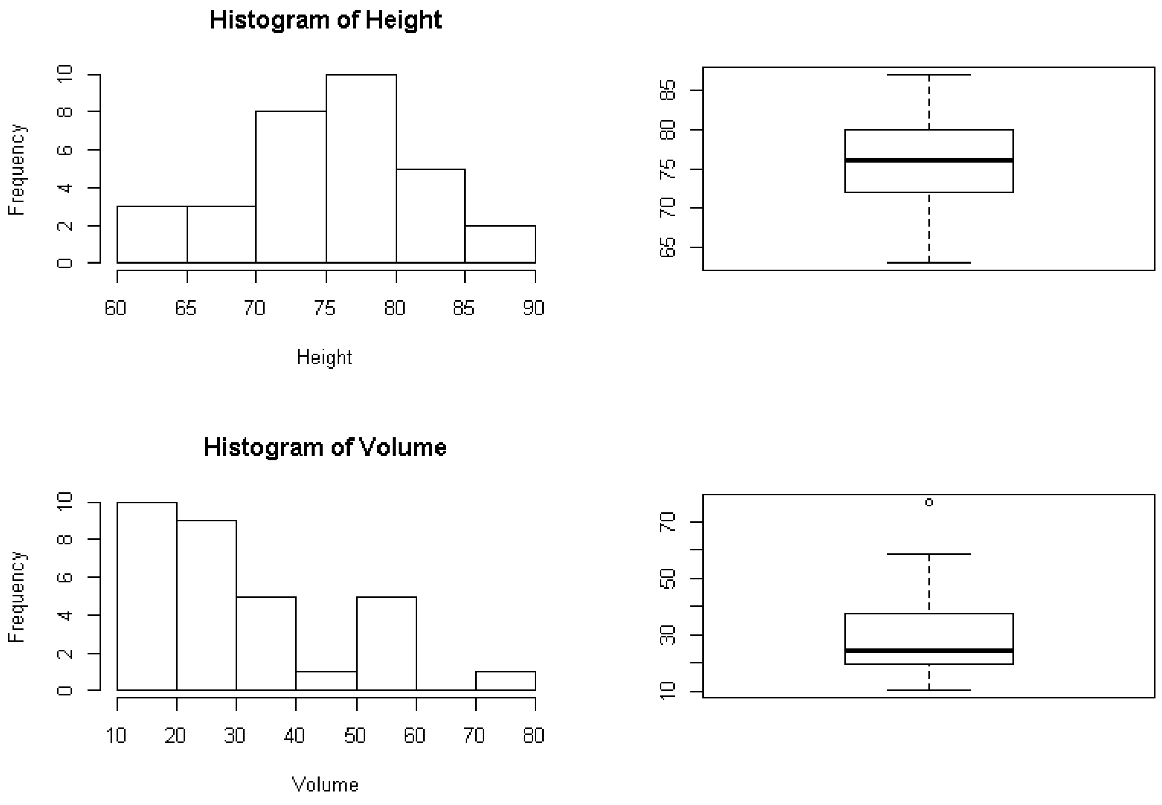
3. Use the function rpois to simulate 10,000 values from a Poisson distribution with a parameter of your own choice. Produce a statistical summary of the result and check that the mean and variance are in reasonable agreement with the true population values.

15 Graphics

R has many facilities for producing high quality graphics. A useful facility is to divide a page into smaller pieces so that more than one figure can be displayed. For example, par(mfrow=c(2,2)) creates a window of graphics with 2 rows and 2 columns. With this choice the windows are filled row-wise. Use mfcol instead of mfrow to fill column-wise. The function par is a general function for setting graphical parameters. There are many options. (See help(par).) Define variables Height and Volume for plotting purposes:

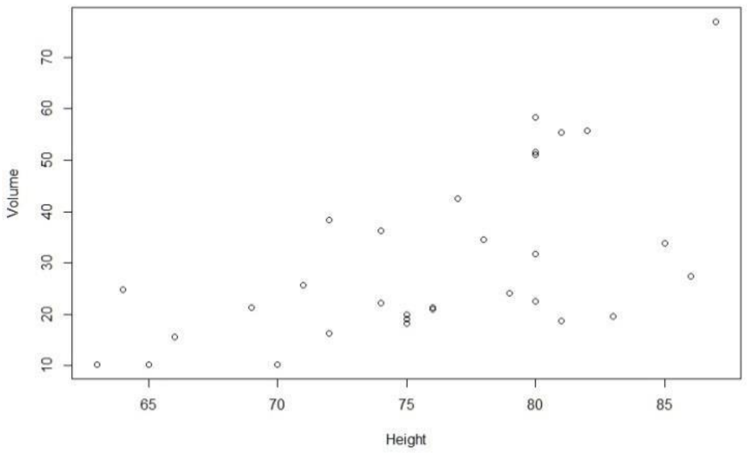
```
> data(trees)
> Height <-  trees$Height
> Volume <- trees$Volume
> summary(Height)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
     63      72      76      76      80      87 
> summary(Volume)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
  10.20   19.40   24.20   30.17   37.30   77.00 
>
> par(mfrow=c(2,2))
> hist(Height)
> boxplot(Height)
> hist(Volume)
> boxplot(Volume)
> par(mfrow=c(1,1))
```

The statements above produce the figure below and returns the graphics window to standard size. Compare the axes of the histograms and boxplots with the values produced by summary().



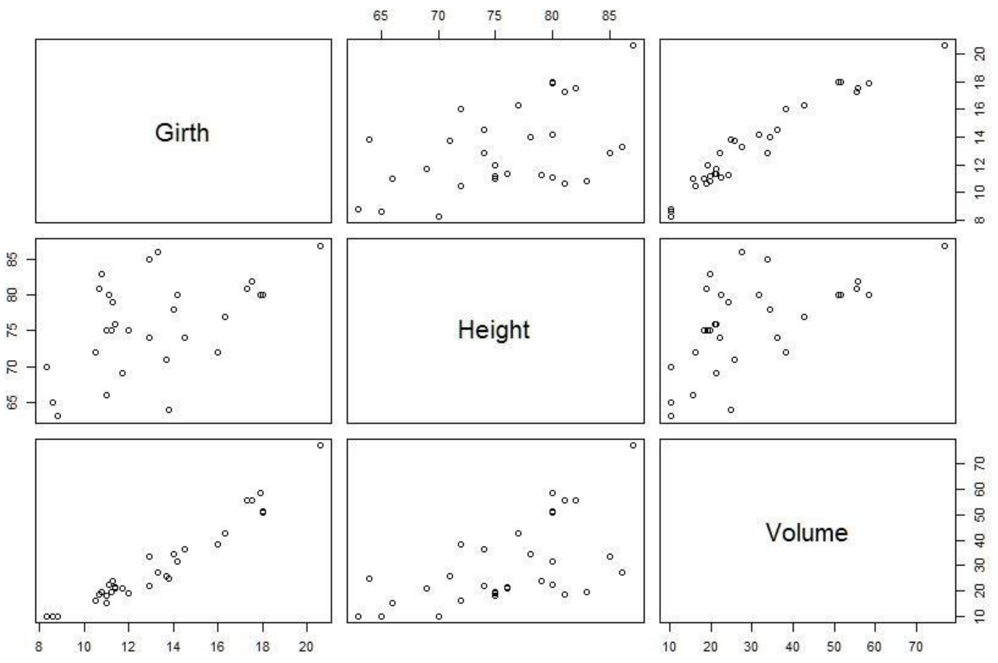
We can also plot one variable against another using the function plot().

```
>   plot(Height,Volume)
```



R can also produce a scatterplot matrix (a matrix of scatterplots for each pair of variables) using the function pairs. The plot() function will accomplish the same result.

```
>   pairs(trees)
```



The plot() function is object-specific: its behavior depends on the object being used. For example, if the object is a matrix, plot is identical to pairs: try plot(trees).

There are also many optional arguments in most plotting functions that can be used to control colors, plotting characters, axis labels, titles, etc.

Exercises

1. Use the data set "faithful" and create a scatter plot matrix of the variables.

2. Use x <-rnorm(250) to generate 250 standard normal random variables. Produce the histogram and compare to a stem-and-leaf plot of the data. The function stem() will produce the stem-and-leaf plot.

References:
1) Davies, Tilman M. (2016) *The Book of R,* San Francisco, CA: No Starch Press [ISBN-13: 978-1593276515]
2) Kabacoff, R. I. (2015) *R in Action, 2nd ed.* Shelter Island, NY: Manning Co. [ISBN-13: 978-1617291388]
3) Stowell, S. (2014). *Using R for* statistics. New York: Apress (distributed by Springer Science+Business Media) [ISBN-13: 978-1484201404]