

Data Visualization and Modelling
Part 1: Introduction to R
in
Master in Modelling for Science and Engineering, UAB

Juan R Gonzalez
(jrgonzalez@creal.cat)

Center for Research in Environmental Epidemiology (CREAL)
Department of Mathematics, Universidad Aut3noma de Barcelona (UAB)

September 15, 2014

Outline of the course

1. **Part 1:** Introduction to R (6h) (Juan R Gonzalez).
2. **Part 2:** Data Simulation, Bootstrapping and Permutation testing (18h) (Pere Puig).
3. **Part 3:** Bayesian Networks (12h) (Rosario Delgado).

NOTE: Slides, papers, R code, software, etc. will be available at *Campus Virtual*.

Assessment

1. Continuous assessment - 3 assessments during the course, weighted as 15%, 50%, 35% corresponding to each part.
2. Each lecturer will explain their own type of assessment.
3. **Part 1 assessment:** Daily homework + final project (individual simple real data analysis with R).

Credits

Part of this material has been prepared by Dr. Norma Bargary, University of Limerick.

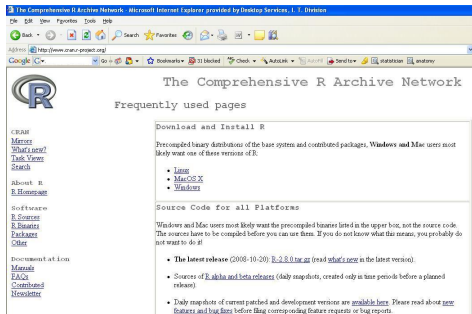
What is R?

R is

- a suite of software facilities for:
 - ▶ reading and manipulating data
 - ▶ computation
 - ▶ conducting statistical analyses
 - ▶ displaying the results
- open-source version (i.e. freely available version - no license fee) of the S programming language, a language for manipulating *objects*
- a programming environment for data analysis and graphics
- a platform for development and implementation of new algorithms
- Software and packages can be downloaded from `www.cran.r-project.org`

Installing R

R must be installed on your system! If it is not, go to `www.cran.r-project.org`



Click on

Windows > base > R-version-win32.exe > Run

and follow the instructions to install the programme.

Starting R

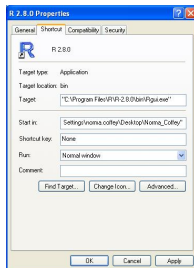
R can be started in the usual way by double-clicking on the R icon on the desktop.

R works best if you have a dedicated folder for each separate project - called the working folder.

- Create the directory/folder that will be used as the working folder, e.g. create a folder on your desktop titled `Your_name` by right-clicking, then clicking `New > Folder`.
- Right-click on an existing R icon and click `Copy`.
- In the working folder, right-click and click `Paste`. The R icon will appear in the folder.

Starting R

- Right-click on the R icon and click Properties.
- In the **Start in** box type the location of the working directory, e.g.
`"C:\Documents and Settings\norma.coffey\Desktop\Your_name"`



- Click Apply, then Ok.

Starting R

There are 3 ways to start R in the working folder:

- double-click on the R shortcut
- double-click on the .RData file in the folder, when it exists
- double-click any R shortcut and use `setwd(filepath)` command

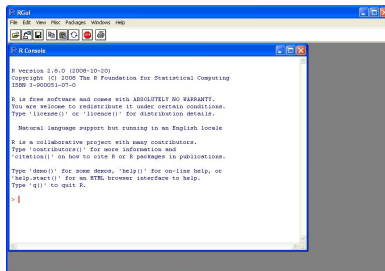


Figure: The R console (command line) window.

How does R work?

- R creates its objects in memory and saves them in a single file called `.RData` (by default)
- Commands are recorded in an `.Rhistory` file
- Commands may be recalled and reissued using up- and down-arrow
- Recalled commands may be edited
- Flawed commands may be abandoned by pressing `<Esc>`
- Copy-and-paste from a “script” file
- Copy-and-paste from the history window used for recalling several commands at once
- To end your session type `q()` or just kill the window.

How does R work?

There are a number of drop-down menus in the R Gui (File, Edit, View, Packages, Help).

Users are expected to type input (*commands*) into R in the console window. When R is ready for input, it prints out its prompt, a ">".

Commands:

- consist of *expressions* or *assignments*
- are separated by a semi-colon (;) or by a newline
- can be grouped together using braces ({ and })

Comments can be included and are indicated with a hash (#).

How does R work?

Users enter a line with a command after the prompt and press `Enter`.

The programme carries out the command and prints the result if relevant. For example, if the expression `2 + 2` is typed in, the following is printed in the R console:

```
> 2 + 2  
[1] 4  
>
```

The prompt `>` indicates that R is ready for another command. If a command is incomplete at the end of a line, the prompt `+` is displayed on subsequent lines until the command is syntactically complete.

Calculator

R can also evaluate other standard calculations:

```
> exp(-2)
[1] 0.1353353
```

```
> 2*3*4*5
[1] 120
```

```
> pi      # R knows about pi
[1] 3.141593
```

```
> 1000*(1 + 0.075)^5 - 1000
[1] 435.6293
```

Assignments

It is often required to store intermediate results so that they do not need to be re-typed over and over again. To assign a value of 10 to the variable `x` type:

```
> x <- 10
```

and press Enter.

Can also use the command

```
> assign("x", 10)
```

There is no visible result, however `x` now has the value 10 and can be used in subsequent expressions.

```
> x  
[1] 10
```

```
> x + x  
[1] 20
```

```
> sqrt(x)  
[1] 3.162278
```

Case sensitivity and variable names

R is a case-sensitive language, e.g. `x` and `X` do not refer to the same variable.

Variable names:

- can be created using letters, digits and the `.` (dot) symbol, e.g. `weight`, `wt.male`
- must not start with a digit or a `.` followed by a digit.
- Some names are used by the system, e.g.
`c`, `q`, `t`, `C`, `D`, `F`, `I`, `T`, `diff`, `df`, `pt` - AVOID!

Objects

You cannot perform much statistics on single numbers. R works by creating different *objects* and using various function calls that create and use those objects.

- Vectors of
 - ▶ numbers
 - ▶ logical values
 - ▶ character strings
 - ▶ complex numbers
- Matrices and general n -way arrays
- Lists - arbitrary collections of objects of any type, e.g. list of vectors, list of matrices, etc.
- Data frames - lists with a rectangular structure
- Connections - connection to files and similar things
- Functions

Objects

During an R session, objects are created and stored by name. The command

```
> ls()
```

displays all currently-stored objects (*workspace*). Objects can be removed using

```
> rm(x, a, temp, wt.males)
```

```
> rm(list=ls())
```

removes all of the objects in the workspace.

At the end of each R session, you are prompted to save your workspace. If you click Yes, all objects are written to the `.RData` file. When R is re-started, it reloads the workspace from this file and the command history stored in `.Rhistory` is also reloaded.

Getting help in R

R has a built-in help facility. To get more information on any specific function, e.g. `sqrt()`, the command is

```
> help(sqrt)
```

An alternative is

```
> ? sqrt
```

Can also obtain help on features specified by special characters. Must enclose in single or double quotes (e.g. `"["`)

```
> help("[")
```

Help is also available in HTML format by running

```
> help.start()
```

For more information use

```
> ? help
```

Packages

"R" contains one or more libraries of packages. Packages contain various functions and data sets for numerous purposes, e.g. survival package, genetics package, fda package, etc.

Some packages are part of the basic installation. Others can be downloaded from CRAN.

To access all of the functions and data sets in a particular package, it must be loaded into the workspace. For example, to load the fda package:

```
> library(fda)
```

One important thing to note is that if you terminate your session and start a new session with the saved workspace, you must load the packages again.

Packages

To check what packages are currently loaded into the workspace

```
> search()
[1] ".GlobalEnv"          "package:MASS"        "package:stats"
[4] "package:graphics"    "package:grDevices"   "package:utils"
[7] "package:datasets"    "package:methods"     "Autoloads"
[10] "package:base"
```

Can remove a package you have loaded use:

```
> detach("package:fda")
```

An interactive session

Create a folder called `Session 1` and copy an R shortcut into this folder. Right-click on this shortcut and go to `Properties`. Change the address in the **Start In** box to the location of your folder.

For the purposes of this session, a data set already stored in R will be used. To access this data, must first load the package containing the data. (R has many packages containing various functions that can be used to analyse data, e.g. if you want to analyse your data using splines, need to load the `splines` package). In this example, the data is stored in the `MASS` package. This is loaded with the command

```
> library(MASS)
```

Now have access to all functions and data sets stored in this package.

An interactive session

We will work with the data set titled “whiteside”. To display the data:

```
> whiteside
      Insul  Temp  Gas
1  Before -0.8  7.2
2  Before -0.7  6.9
3  Before  0.4  6.4
4  Before  2.5  6.0
5  Before  2.9  5.8
6  Before  3.2  5.8
7  Before  3.6  5.6
8  Before  3.9  4.7
9  Before  4.2  5.8
10 Before  4.3  5.2
```

This is a particular type of object called a data frame.

A full description of these data is found using

```
> ? whiteside
```

An interactive session

To remind ourselves of the names of the columns:

```
> names(whiteside)
[1] "Insul" "Temp"  "Gas"
```

Summary statistics for each column are determined using

```
> summary(whiteside)
```

Insul	Temp	Gas
Before:26	Min. :-0.800	Min. :1.300
After :30	1st Qu.: 3.050	1st Qu.:3.500
	Median : 4.900	Median :3.950
	Mean : 4.875	Mean :4.071
	3rd Qu.: 7.125	3rd Qu.:4.625
	Max. :10.200	Max. :7.200

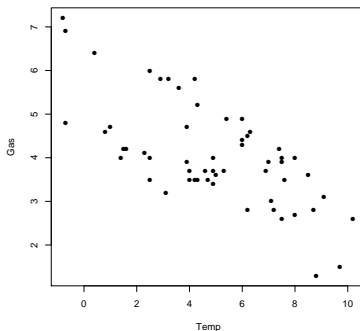
Access the data in a particular column

```
> whiteside$Temp
```

An interactive session

A plot of gas consumption versus temperature is now created.

```
> plot(Gas ~ Temp, data=whiteside, pch=16)
```

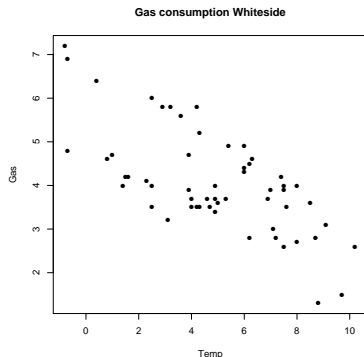


An interactive session

A title can be put on the graph

```
> plot(Gas ~ Temp, data=whiteside, pch=16, main="Gas consumption Whiteside")
```

Note: Do not need to re-type entire command. Press the up-arrow key to recall the last command. Edit this command to include `main="Gas consumption Whiteside"`, as above.



An interactive session

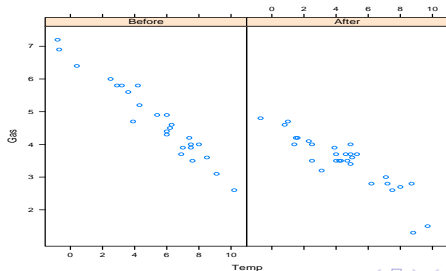
Can produce separate graphs for gas consumption versus temperature before insulation used and after insulation used.

Requires the use of `xyplot()` available in the `lattice` package.

Need to load this package into R before function can be used.

Then use `xyplot()`.

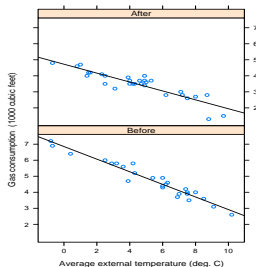
```
> library(lattice) # Loads the lattice package  
> ? xyplot # Gives more information on xyplot()  
> xyplot(Gas ~ Temp | Insul, whiteside)
```



An interactive session

More complex plot (can copy code from examples section of help file on whiteside data set obtained earlier)

```
> xyplot(Gas ~ Temp | Insul, whiteside, panel =  
+   function(x, y, ...) {  
+     panel.xyplot(x, y, ...)  
+     panel.lmline(x, y, ...)  
+   }, xlab = "Average external temperature (deg. C)",  
+   ylab = "Gas consumption (1000 cubic feet)", aspect = "xy",  
+   strip = function(...) strip.default(..., style = 1))  
>
```



An interactive session

Entry of data at the command line

Will now create a data frame with 2 columns. The following data gives, for each amount by which an elastic band is stretched over the end of a ruler, the distance that the band moved when released:

Stretch (mm)	Distance (cm)
46	148
54	182
48	173
50	166
44	109
42	141
52	166

An interactive session

Entry of data at the command line

Use the `data.frame()` command.

Name the data frame **elasticband**.

```
> elasticband <- data.frame(stretch = c(46,54,48,50,44,42,52),  
+ distance=c(148,182,173,166,109,141,166))  
>  
> elasticband  
  stretch distance  
1       46      148  
2       54      182  
3       48      173  
4       50      166  
5       44      109  
6       42      141  
7       52      166
```

Exercises

1. Create summary statistics for the elastic band data.
2. Create a plot of distance versus stretch.
3. Use the `help()` command to find more information about the `hist()` command.
4. Create a histogram of the distance using `hist()`.
5. The following data are on snow cover for Eurasia in the years 1970-1979.

year	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979
snow.cover	6.5	12.0	14.9	10.0	10.7	7.9	21.9	12.5	14.5	9.2

- (i) Enter the data into R. To save keystrokes, enter the successive years as `1970:1979`.
 - (ii) Take the logarithm of snow cover.
 - (iii) Plot snow cover versus year.
6. Display all objects in the workspace. Remove the data frame **elasticband**.

Objects and simple manipulations

Vectors

Vectors are the simplest type of object in R. There are 3 main types of vectors:

- Numeric vectors
- Character vectors
- Logical vectors
- (Complex number vectors)

To set up a numeric vector `x` consisting of 5 numbers, 10.4, 5.6, 3.1, 6.4, 21.7, use

```
> x <- c(10.4, 5.6, 3.1, 6.4, 21.7)
```

or

```
> assign("x", c(10.4, 5.6, 3.1, 6.4, 21.7))
```


Numeric Vectors

To print the contents of `x`:

```
> x  
[1] 10.4  5.6  3.1  6.4 21.7
```

The `[1]` in front of the result is the index of the first element in the vector `x`.

To access a particular element of `x`

```
> x[1]  
[1] 10.4
```

```
> x[5]  
[1] 21.7
```

Can also do further assignments:

```
> y <- c(x, 0, x)
```

Creates a vector `y` with 11 entries (two copies of `x` with a 0 in the middle)

Numeric Vectors

Computations

- Computations are performed element-wise, e.g.

```
> 1/x  
[1] 0.09615385 0.17857143 0.32258065 0.15625000 0.04608295
```

- Short vectors are “recycled” to match long ones

```
> v <- x + y
```

Warning message:

```
In x + y : longer object length is not a multiple  
of shorter object length
```

- Some functions take vectors of values and produce results of the same length:

sin, cos, tan, asin, acos, atan, log, exp, Arith, ...

```
> cos(x)  
[1] -0.5609843 0.7755659 -0.9991352 0.9931849 -0.9579148
```

Numeric Vectors

Computations

- Some functions return a single value:

`sum, mean, max, min, prod, ...`

```
> sum(x)
```

```
[1] 47.2
```

```
>
```

```
> length(x)
```

```
[1] 5
```

```
>
```

```
> sum(x)/length(x)
```

```
[1] 9.44
```

```
>
```

```
> mean(x)
```

```
[1] 9.44
```

- Some functions are a bit special:

`cumsum, sort, range, pmax, pmin, ...`

Numeric Vectors

Complex Numbers

Care must be taken when working with complex numbers. The expression

```
> sqrt(-17)
```

```
[1] NaN
```

Warning message:

```
In sqrt(-17) : NaNs produced
```

gives NaN (i.e. Not a Number) and a warning but

```
> sqrt(-17+0i)
```

```
[1] 0+4.123106i
```

performs the calculations as complex numbers.

Generating Sequences I

R has a number of ways to generate sequences of numbers. These include:

- the colon ":", e.g.

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

This operator has the highest priority within an expression, e.g. $2*1:10$ is equivalent to $2*(1:10)$.

- the `seq()` function. (Use `> ? seq` to find out more about this function).

```
> seq(1,10)
```

```
> seq(from=1, to=10)
```

```
> seq(to=10, from=1)
```

are all equivalent to `1:10`.

Generating Sequences II

Can also specify a step size (using `by=value`) or a length (using `length=value`) for the sequence.

```
> s1 <- seq(1,10, by=0.5)
> s1
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0
[12] 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
```

```
> s2 <- seq(1,10, length=5)
> s2
[1] 1.00 3.25 5.50 7.75 10.00
```

- the `rep()` function - replicates objects in various ways.

```
> s3 <- rep(x, 2)
> s3
[1] 10.4 5.6 3.1 6.4 21.7 10.4 5.6 3.1 6.4 21.7
[11] 10.4 5.6 3.1 6.4 21.7
```

```
> s4 <- rep(c(1,4),c(10,15))
> s4
[1] 1 1 1 1 1 1 1 1 1 1 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
```

Character Vectors

- To set up a character/string vector `z` consisting of 4 place names use

```
> z <- c("Canberra", "Sydney", "Newcastle", "Darwin")
```

```
> z <- c('Canberra', 'Sydney', 'Newcastle', 'Darwin')
```

- Can be concatenated using `c()`

```
> c(z, "Mary", "John")
```

```
[1] "Canberra" "Sydney" "Newcastle" "Darwin" "Mary" "John"
```

- Lots of in-built functions in R to manipulate character vectors.

Logical Vectors

- A logical vector is a vector whose elements are TRUE, FALSE or NA.
- Are generated by *conditions*, e.g.

```
> temp <- x > 13
```

Takes each element of the vector `x` and compares it to 13. Returns a vector the same length as `x`, with a value TRUE when the condition is met and FALSE when it is not.

- The logical operators are `>`, `>=`, `<`, `<=`, `==` for exact equality and `!=` for inequality.
- `&` and `|`
- `&&` and `||`

Missing Values

In some cases the entire contents of a vector may not be known. For example, missing data from a particular data set.

A place can be reserved for this by assigning it the special value NA.

Can check for NA values in a vector `x` using the command

```
> is.na(x)
```

Returns a logical vector the same length as `x` with a value TRUE if that particular element is NA.

```
> w <- c(1:10, rep(NA,4), 22)
```

```
> is.na(w)
```

Indexing Vectors

Have already seen how to access single elements of a vector.
Subsets of a vector may also be selected using a similar approach.

- `> ex1 <- w[!is.na(w)]`

Stores the elements of the vector `w` that do NOT have the value NA, into `ex1`.

- `> ex2 <- w[1:3]`

Selects the first 3 elements of the vector `w` and stores them in the new vector `ex2`.

- `> ex3 <- w[-(1:4)]`

Using the `-` sign indicates that these elements should be *excluded*. This command excludes the first 4 elements of `w`.

```
> ex4 <- w[-c(1,4)]
```

In this case only the 1st and 4th elements of `w` are excluded.

Modifying Vectors

To alter the contents of a vector, similar methods can be used.

- Remember `x` has contents

```
> x  
[1] 10.4  5.6  3.1  6.4 21.7
```

For example, to modify the 1st element of `x` and assign it a value 5 use

```
> x[1] <- 5  
> x  
[1]  5.0  5.6  3.1  6.4 21.7
```

- The following command replaces any NA (missing) values in the vector `w` with the value 0

```
> w[is.na(w)] <- 0
```

Modifying Vectors

- Let

```
> y <- c(-1, -2, rep(0, 3), 7, 8, 9)
> y
[1] -1 -2  0  0  0  7  8  9
```

The following replaces any elements of `y` with a negative value with the corresponding positive value. (Note this is equivalent to using the in-built `abs()` function).

```
> y[y < 0] <- -y[y < 0]
> y
[1] 1 2 0 0 0 7 8 9
```

Factors

A factor is a special type of vector used to represent categorical data, e.g. gender, social class, etc.

- Stored internally as a numeric vector with values $1, 2, \dots, k$, where k is the number of levels.
- Can have either *ordered* and *unordered* factors.
- A factor with k levels is stored internally consisting of 2 items
 - (a) a vector of k integers
 - (b) a character vector containing strings describing what the k levels are.

Factors

Example

Consider a survey that has data on 200 females and 300 males. If the first 200 values are from females and the next 300 values are from males, one way of representing this is to create a vector

```
> gender <- c(rep("female", 200), rep("male", 300))
```

To change this into a factor

```
> gender <- factor(gender)
```

The factor `gender` is stored internally as

1	female
2	male

Each category, i.e. female and male, is called a level of the factor.

To determine the levels of a factor the function `levels()` can be used:

```
> levels(gender)
[1] "female" "male"
```

Factors

Example

Five people are asked to rate the performance of a product on a scale of 1-5, with 1 representing very poor performance and 5 representing very good performance. The following data were collected.

```
> satisfaction <- c(1, 3, 4, 2, 2)
> fsatisfaction <- factor(satisfaction, levels=1:5)
> levels(fsatisfaction) <- c("very poor", "poor", "average",
"good", "very good")
```

The first line creates a numeric vector containing the satisfaction levels of the 5 people. Want to treat this as a categorical variable and so the second line creates a factor. The `levels=1:5` argument indicates that there are 5 levels of the factor. Finally the last line sets the names of the levels to the specified character strings.

Exercises I

Vectors and Factors

1. Create a vector `x` with the following entries

3 4 1 1 2 1 4 2 1 1 5 3 1 1 1 2 4 5 5 3

Check which elements of `x` are equal to 1 (Hint use `==` operator). Modify `x` so that all of the 1's are changed to 0's.

2. Create a vector `y` containing the elements of `x` that are greater than 1.
3. Create a sequence of numbers from 1 to 20 in steps of 0.2 and store.
4. Concatenate `x` and `y` into a vector called `newVec`.
5. Display all objects in the workspace and then remove `newVec` (see Lecture 1).

Exercises II

Vectors and Factors

6. Six patients were asked to rate their pain from 0 to 3, with 0 representing 'no pain', 1 representing 'mild' pain, 2 representing 'medium' pain and 3 representing 'severe' pain. The following results were obtained:

Patient	1	2	3	4	5	6
Pain level	0	3	1	2	1	2

Create a factor `fpain` to represent the above data.

Matrices

A matrix

- is a two-dimensional array of numbers;
- has rows and columns;
- is used for many purposes in statistics.

In R matrices are represented as vectors with dimensions.

```
> m <- rnorm(12) # Creates a vector of 12 random numbers
> m
[1] -0.32902981  0.64425299  0.91911540 -0.40675505  0.60745737 -0.06756709
[7]  0.38349915 -2.35291296  0.37126362 -1.33875464 -0.49121615 -1.55876372

> dim(m) <- c(3,4)
> m
      [,1]      [,2]      [,3]      [,4]
[1,] -0.3290298 -0.40675505  0.3834992 -1.3387546
[2,]  0.6442530  0.60745737 -2.3529130 -0.4912161
[3,]  0.9191154 -0.06756709  0.3712636 -1.5587637
```

Matrices

The `dim` function sets the *dimension* of `m`.

Causes R to treat the vector of 12 numbers as a 3×4 matrix.

Note that the storage is carried out by filling in the columns first, then the rows.

Another way to create a matrix is to use the `matrix()` function.

```
> n <- rnorm(10) # Generates a new vector of 10 random numbers
> matrix(n, nrow=5, ncol=2, byrow=T)
      [,1]      [,2]
[1,] 1.1202412 -0.6622302
[2,] 1.1286009  0.8751449
[3,] 1.2719938 -0.6243375
[4,] 0.7223669  0.8414961
[5,] 0.6330745  0.8950885
```

Matrices

The `byrow=T` command causes the matrix to be filled in row by row rather than column by column.

Re-call the last command and change `byrow=T` to `byrow="F"`. Notice the difference between the two outputs. This time the matrix is filled in column by column.

Useful functions for matrices include `nrow()`, `ncol()`, `t()`, `rownames()`, `colnames()`.

```
> nrow(m)
[1] 3
```

```
> rownames(m) <- c("A", "B", "C")
> m
```

	[,1]	[,2]	[,3]	[,4]
A	-0.3290298	-0.40675505	0.3834992	-1.3387546
B	0.6442530	0.60745737	-2.3529130	-0.4912161
C	0.9191154	-0.06756709	0.3712636	-1.5587637

Matrices

The `t()` function is the transposition function (rows become columns and vice versa).

```
> t(m)
```

	A	B	C
[1,]	-0.3290298	0.6442530	0.91911540
[2,]	-0.4067550	0.6074574	-0.06756709
[3,]	0.3834992	-2.3529130	0.37126362
[4,]	-1.3387546	-0.4912161	-1.55876372

Can merge vectors and matrices together, column-wise or row-wise using `rbind()` (add on rows) or `cbind()` (add on columns).

When using `rbind()` - if combining matrices with other matrices, the matrices must have the same number of columns. If combining vectors with other vectors or vectors with matrices the vectors can have any length but will be lengthened/shortened accordingly if of differing lengths.

Matrices

When using `cbind()` - if combining matrices with other matrices, the matrices must have the same number of rows. If combining vectors with other vectors or vectors with matrices, the vectors can have any length but will be lengthened/shortened accordingly if of differing lengths.

A warning message is printed.

```
> X1 <- matrix(1:12, nrow=3, ncol=4, byrow=T)
> X2 <- matrix(20:27, nrow=2, ncol=4)
> rbind(X1,X2,n)
```

	[,1]	[,2]	[,3]	[,4]
	1.000000	2.000000	3.000000	4.000000
	5.000000	6.000000	7.000000	8.000000
	9.000000	10.000000	11.000000	12.000000
	20.000000	22.000000	24.000000	26.000000
	21.000000	23.000000	25.000000	27.000000
n	1.120241	-0.6622302	1.128601	0.8751449

Warning message:

In `rbind(X1, X2, n)` :

number of columns of result is not a multiple of vector length (arg 3)

Matrices

If the vector is too short, the values are re-cycled

```
> rbind(X1,X2,c(1,2))
      [,1] [,2] [,3] [,4]
[1,]     1     2     3     4
[2,]     5     6     7     8
[3,]     9    10    11    12
[4,]    20    22    24    26
[5,]    21    23    25    27
[6,]     1     2     1     2
```

- Generate 2 new matrices X3 and X4 that have the same number of rows.

Use `cbind()` to combine the matrices column-wise.

Need to be careful when working with matrices. For example, if A and B are square matrices of the same size then

```
> A <- matrix(1:9, nrow=3, ncol=3)
> B <- matrix(10:18, nrow=3, ncol=3)
> C <- A %*% B # Calculates the product of two matrices, C = AB

> C <- A * B   # Calculates element by element products
```

Matrices

Other functions to work on matrices include:

```
crossprod(A, B)    # = t(A) %*% B
```

```
diag(n)            # Creates a diagonal matrix with  
                   # the values in the vector n on  
                   # the diagonal
```

```
solve(C)           # Calculates the inverse of A
```

```
C^(-1)             # Calculates 1/c_ij
```

```
eigen(C)           # Calculates the eigenvalues and  
                   # eigenvectors of C
```


Indexing Matrices

Say we have a 5×6 matrix

```
> X <- matrix(rnorm(30), nrow=5)
> dimnames(X) <- list(letters[1:5], LETTERS[1:6])
> X
```

	A	B	C	D	E	F
a	0.4233750	-0.6569585	0.67730663	0.4258022	-0.2003732	0.5934342
b	0.8045364	0.3983394	1.11778619	-0.2736120	0.5258172	-0.8595252
c	0.7479229	1.5591540	1.04614639	1.0419031	2.6454571	-0.2541819
d	0.6153824	-1.0451224	-0.03475772	0.9428584	0.3023099	0.2933696
e	0.8011791	-0.5668138	1.45136460	-0.3036993	0.2801525	-1.4701663

Can access the value in row 3, column 2 using

```
> X[3,2]           > X["c", "B"]
[1] 1.559154         [1] 1.559154
```

Indexing Matrices

Can also access multiple elements, e.g. we wish to extract

- elements in columns 2 and 4

```
> X[,c(2,4)]
```

	B	D
a	-0.6569585	0.4258022
b	0.3983394	-0.2736120
c	1.5591540	1.0419031
d	-1.0451224	0.9428584
e	-0.5668138	-0.3036993

- elements in rows 2 to 4

```
> X[2:4,]
```

	A	B	C	D	E	F
b	0.8045364	0.3983394	1.11778619	-0.2736120	0.5258172	-0.8595252
c	0.7479229	1.5591540	1.04614639	1.0419031	2.6454571	-0.2541819
d	0.6153824	-1.0451224	-0.03475772	0.9428584	0.3023099	0.2933696

Indexing Matrices

- elements $X[1,3]$, $X[2,2]$ and $X[3,1]$ - easiest to create an index array

```
> index <- array(c(1:3, 3:1), dim=c(3,2))
```

```
> index
```

```
      [,1] [,2]
[1,]     1     3
[2,]     2     2
[3,]     3     1
```

```
> X[index]
```

```
[1] 0.6773066 0.3983394 0.7479229
```

- want to replace these elements by zero

```
> X[index] <- 0
```

	A	B	C	D	E	F
a	0.4233750	-0.6569585	0.00000000	0.4258022	-0.2003732	0.5934342
b	0.8045364	0.0000000	1.11778619	-0.2736120	0.5258172	-0.8595252
c	0.0000000	1.5591540	1.04614639	1.0419031	2.6454571	-0.2541819
d	0.6153824	-1.0451224	-0.03475772	0.9428584	0.3023099	0.2933696
e	0.8011791	-0.5668138	1.45136460	-0.3036993	0.2801525	-1.4701663

Arrays

An array can have multiple dimensions.

A matrix is a special case of an array (a 2-d array).

Can construct an array from a vector `z` containing 300 elements using the `dim()` function (as for matrices).

```
> z <- rnorm(300)
> dim(z) <- c(10, 6, 5)
```

Creates a 3-d array with dimensions $10 \times 6 \times 5$ (like storing 5 matrices, each with 10 rows and 6 columns).

Can also use the `array()` function.

```
> A1 <- array(0, c(2, 2, 3))    # Creates an array of zeros

> a <- rnorm(50)
> A2 <- array(a, c(5, 5, 2))    # Creates an array from vector a
```

Use `? array` to find out more about arrays.

Indexing Arrays

Elements of multi-dimensional arrays can be extracted using similar techniques. For example

```
> arr.1 <- array(1:24, dim=c(4,2,3))  
> arr.1[2,,]      # Extracts the data in row 2 of the 3 'matrices'.  
  
> arr.1[,2,]      # Extracts the data in column 2 of the 3 'matrices'.  
  
> arr.1[, ,1]      # Extracts the data in the first 'matrix'.  
  
> arr.1[1,2,3]     # Extracts the data in the row 1, column 2 of the  
                    # third 'matrix'.
```

Exercises I

Matrices and Arrays

1. Construct a matrix A with values 10, 20, 30, 50 in column 1, values 1, 4, 2, 3 in column 2 and values 15, 11, 19, 5 in column 3, i.e. a 4×3 matrix. Also construct a vector B with values 2.5, 3.5, 1.75. Check your results to ensure that they are correct.
2. Combine A and B into a new matrix C using `cbind()`.
3. Combine A and B into a new matrix H using `rbind()`.
4. Determine the dimensions of C and H using `dim()` function.
5. Calculate the following:

$$\begin{pmatrix} 1 & 4 & 3 \\ 0 & -2 & 8 \end{pmatrix} \times \begin{pmatrix} 1 & 9 \\ 2 & 17 \\ -6 & 3 \end{pmatrix}$$

Exercises II

Matrices and Arrays

6. Create a $4 \times 4 \times 2$ array `arr` using the values 1 to 32.
7. Print out the value in row 1, column 3 of the first 'matrix'.
8. Print out the value in row 2, column 4 of the second 'matrix'.
9. Add these two values together.

Lists

Lists

- are an ordered collection of components;
- components may be arbitrary R objects (data frames, vectors, lists, etc.);
- single bracket notation for sublists;
- double bracket notation for individual components;
- construct using the function `list()`.

A simple example of a list is as follows:

```
> L1 <- list(name="Fred", wife="Mary", no.children=3, child.ages=c(4,7,9))
```

Each component of the list is given a name (i.e. `name`, `wife`, `no.children`, `child.ages`).

Lists

Construct a second list omitting the component names:

```
> L2 <- list("Fred", "Mary", 3, c(4,7,9))
```

What is the difference between the two?

Three equivalent ways of accessing the first component

```
> L1[["name"]]
```

```
[1] "Fred"
```

```
> L1$name
```

```
[1] "Fred"
```

```
> L1[[1]]
```

```
[1] "Fred"
```

A sublist consisting of the first component only

```
> L1[1]
```

```
$name
```

```
[1] "Fred"
```

Lists

The names of each component of the list can be accessed using

```
> names(L1)
[1] "name"      "wife"      "no.children"  "child.ages"
```

```
> names(L2)
NULL
```

Can set the names for the list components after the list has been created.

```
> names(L2) <- c("name.hus", "name.wife", "no.child", "child.age")
```

```
> names(L2)
[1] "name.hus"      "name.wife"      "no.child"      "child.age"
```

Can also concatenate lists:

```
> L3 <- c(L1, L2)
```

Exercises

Lists

1. Create 4 vectors `Year`, `mean_weight`, `Gender` and `mean_height` with the following entries:

Year	1980	1988	1996	1998	2000	2002
mean_weight	71.5	72.1	73.7	74.3	75.2	74.7
Gender	M	M	F	F	M	M
mean_height	179.3	179.9	180.5	180.1	180.3	180.4

2. Create a list called `mylist` consisting of the above vectors.
Give each component of the list a name.
3. Use 3 different ways to access the 4th element of the list.

Data Frames

A data frame

- can be thought of as a data matrix or data set;
- is a generalisation of a matrix;
- is a list of vectors and/or factors of the same length;
- has a unique set of row names.
- Data in the same position across columns come from the same experimental unit.

Can create data frames from pre-existing variables:

```
> d <- data.frame(mean_weight, Gender)
```

This is the same as re-typing

```
> d <- data.frame(mean_weight=c(71.5,72.1,73.7,74.3,75.2,74.7),  
+ Gender=c("M", "M", "F", "F", "M", "M"))
```

Data Frames

Can also convert other objects (e.g. lists, matrices) into a data frame.

In the previous exercises you created a list called `mylist`.

To convert this to a data frame:

```
> new.data <- as.data.frame(mylist)
```

Note that the data in each row are related, that is the same person is male, has a birth date of 1980, a weight of 71.5 kg and a height of 179.3 cm.

As with lists, individual components (columns) can be accessed using the `$` notation.

```
> new.data$year  
[1] 1980 1988 1996 1998 2000 2002
```

Indexing Data Frames

Have already seen how to access individual/sets of values in vectors, matrices and arrays.

It is possible to use the same methods to access values of a data frame. Makes use of the matrix-like structure.

```
> new.data
  year mean_weight gender mean_height
1 1980         71.5      M        179.3
2 1988         72.1      M        179.9
3 1996         73.7      F        180.5
4 1998         74.3      F        180.1
5 2000         75.2      M        180.3
6 2002         74.7      M        180.4
```

```
> new.data[3,2]
[1] 73.7
```

gives the value in the 3rd row and 2nd column of new.data.

Indexing Data Frames

```
> new.data[,2]  
[1] 71.5 72.1 73.7 74.3 75.2 74.7
```

returns all the measurements in the 2nd column (same as using `new.data$mean_weight`).

```
> new.data[3,]  
  year mean_weight gender mean_height  
3 1996         73.7      F         180.5
```

returns all measurements for the 3rd individual.

NB - Comma!!

Indexing Data Frames

Other indexing techniques also apply. For example, selecting all data for cases that satisfy some criterion, such as the data for all males.

```
> new.data[new.data$gender == "M",]
```

Selects the rows of the data frame where `gender` is male. Note that the row names are the same as those in the original data frame.

To select only the weight and height of females born after 1996 use:

```
> new.data[new.data$gender == "F" & new.data$year > 1996, c(2,4)]  
  mean_weight mean_height  
4          74.3       180.1
```


Indexing Data Frames

The first two logical commands

`new.data$gender == "F" & new.data$year > 1996` dictate which rows to select from the data frame (the `&` tells R that BOTH conditions must be satisfied).

The `c(2,4)` dictates which columns to select (in this case columns 2 and 4).

Replacing the `&` with a `|` selects the rows that satisfy EITHER condition.

Re-type the last command and replace the `&` with a `|`. Notice how the results differ.

```
> new.data[new.data$gender == "F" | new.data$year > 1996, c(2,4)]
```

	mean_weight	mean_height
3	73.7	180.5
4	74.3	180.1
5	75.2	180.3
6	74.7	180.4

`attach()` and `detach()`

It can be time consuming to access the variables in a data frame if you need to repeatedly use long commands like

```
> new.data[new.data$gender == "F" | new.data$year > 1996, c(2,4)]
```

The `attach()` command

```
> attach(new.data)
> search()
```

places the data frame `new.data` onto the search path and forces R to look for objects among the variables in `new.data`.

No need to use the `$` notation.

```
> new.data[gender == "F" | year > 1996, c(2,4)]
```

`attach()` **and** `detach()`

Must be careful that there are no other objects in the workspace with the same names as the names of the columns in the data frame - R will get confused.

If you want to make changes to the data frame, must still use the `$` notation.

To detach the data frame

```
> detach(new.data)
> search()
```

Exercises I

Data Frames

1. Create a data frame called `club.points` with the following data.

Firstname	Lastname	Age	Gender	Points
Alice	Ryan	37	F	278
Paul	Collins	34	M	242
Jerry	Burke	26	M	312
Thomas	Dolan	72	M	740
Marguerite	Black	18	F	177
Linda	McGrath	24	F	195

2. Store the points for every person into a vector called `pts`, then calculate the average number of points received. (Hint use `mean()` function).
3. Store the data for the females only into a data frame called `fpoints`.

Exercises II

Data Frames

4. The age for Jerry Burke was entered incorrectly. Change his age to 28.
5. Determine the maximum age of the males.
6. Extract the data for people with more than 100 points and are over the age of 30.

The apply Family

Sometimes want to apply a function to each element of a vector/data frame/list/array.

Four members: `lapply`, `sapply`, `tapply`, `apply`

- `lapply`: takes any structure and gives a list of results (hence the 'l')
- `sapply`: like `lapply`, but tries to simplify the result to a vector or matrix if possible (hence the 's')
- `apply`: only used for arrays/matrices
- `tapply`: allows you to create tables (hence the 't') of values from subgroups defined by one or more factors.

Used for

- efficiency relative to explicit loops
- convenience

The apply Family

lapply and sapply

Will use an in-built data set called `trees`. Gives girth, height and volume measurements for 31 trees.

To calculate the mean of each variable in `trees`

```
> lapply(trees, mean)
```

```
$Girth
```

```
[1] 13.24839
```

```
$Height
```

```
[1] 76
```

```
$Volume
```

```
[1] 30.17097
```

```
> sapply(trees, mean)
```

```
  Girth  Height  Volume  
13.24839 76.00000 30.17097
```

The apply Family

apply

The second argument is the function that we want to compute (can also write your own functions) and we can also specify other arguments, e.g. `na.rm=T` to remove NA values from the calculations.

To apply a function to either the row/columns of a matrix

```
> X1 <- matrix(1:12, nrow=3)
> apply(X1, 1, sum)
[1] 22 26 30
```

Calculates the sum of the values in each row. The second argument is an index (or vector of indices) dictating the dimension to apply the function to.

```
> apply(X1, 2, mean)
[1] 2 5 8 11
```

Calculates the mean of the values in each column.

The apply Family

tapply

Type the following code into R

```
> library(MASS)           # Allows R to use stored functions and data in this
                           # library
> Cars93                   # Data set stored in this library
```

	Manufacturer	Model	Type	Min.Price	Price	Max.Price	MPG.city
1	Acura	Integra	Small	12.9	15.9	18.8	25
2	Acura	Legend	Midsize	29.2	33.9	38.7	18
3	Audi	90	Compact	25.9	29.1	32.3	20
4	Audi	100	Midsize	30.8	37.7	44.6	19
5	BMW	535i	Midsize	23.7	30.0	36.2	22

Manufacturer is a factor

```
> is.factor(Cars93$Manufacturer)
[1] TRUE
```

and we want to calculate the average price of a car for each manufacturer.

```
> tapply(Cars93$Price, Cars93$Manufacturer, mean)
```

Acura	Audi	BMW	Buick	Cadillac
24.90000	33.40000	30.00000	21.62500	37.40000

Reading Data from Files

Sometimes data can be stored in external files like text files, Excel files etc. R provides several functions to read data in from such files.

- `scan()` - offers a low-level reading facility
- `read.table()` - can be used to read data frames from formatted text files
- `read.csv()` can be used to read data frames from comma separated variable files.
- When reading from Excel files, the simplest method is to save each worksheet separately as a .csv file and use `read.csv()` on each. A better way is to open a data connection to the excel file directly and use the ODBC facilities.

Reading Data from Files

General rules for storing data in external files:

- Use tabs as separators
- Each row has to have the same number of columns
- Missing data is NA, not empty
- As .txt: The ideal format!
- As .xls: save as tabbed text

Reading Data from Files

`read.table()`

Save the file `example.txt` into the same directory/folder as your R session (`.RData` file).

```
> example.data <- read.table("example.txt",header=TRUE)
```

Note that the variable names, `"x1"`, `"x2"`, and `"y"`, were in the first line of the data file, hence the `header=TRUE` command. This tells R that the column names in the text file should be used as the variable names in `example.data`.

Check whether `example.data` is a matrix, data frame, list... (Hint: use `is.matrix()`, `is.data.frame()`, etc.)

If the file is in a different directory/folder than your R session, specify a full file path

```
> example.data <- read.table("C:/.../Desktop/example.txt",header=TRUE)
```

Writing Data to Files

May also want to save your output in an external file. Use `write.table()` or `write.csv()`.

```
> write.table(example.data, "Ex1.txt", row.names=FALSE, sep=" ")
```

Writes the data in `example.data` to a text file called "Ex1.txt" which is in the same folder as your R session. (Can also specify a filepath)

The `row.names=FALSE` command ensures that the row numbers are not saved in the file. (Exercise: re-call the last command and change `row.names=FALSE` to `row.names=TRUE`.)

The `sep=" "` command ensures that the output is separated by a space. Can change this using

```
> write.table(example.data, "Ex1.txt", row.names=FALSE, sep=", ")
```

Output is now separated by commas.

Writing Data to Files

To write data to an Excel file

```
> write.csv(example.data, "Ex2.csv", row.names=FALSE)
```

For more information on importing and exporting data in R refer to the *R Data Import/Export* manual available online.

Exercises

Reading Data from Files

1. Download the example2 data and save.
2. Read this data into R.
3. Print out the data for cases 10 to 18.
4. Print out the data for column 2, cases 23, 2, and 5 (in that order).
5. Find the mean, standard deviation, minimum and maximum for each variable using the smallest number of commands possible.

Scripting

So far we have entered commands into R on a line by line basis. This is often not very practical, e.g. if you enter a command over 5 lines you need to use the up-arrow 5 times to re-enter it.

As a result, R scripts are used. These are essentially text files containing collections of R commands that can be copied and pasted into the R console.

Can use Notepad, Wordpad, etc. but there are some specialist text editors that work specifically for R, e.g. TinnR (can be downloaded from <http://sourceforge.net/projects/tinn-r>) and RWinEdt (used with LaTeX and WinEdt - can be downloaded from CRAN).

Scripting

We will use RWinEdt since WinEdt is available to us.

Go to <http://cran.r-project.org/>.

Click on the Packages link on the left hand side of the page. This gives a list of all packages available for download.

Find RWinEdt package and click on the link. Then click on RWinEdt_1.8-0.zip and click Save. Choose a place to save the zip file.

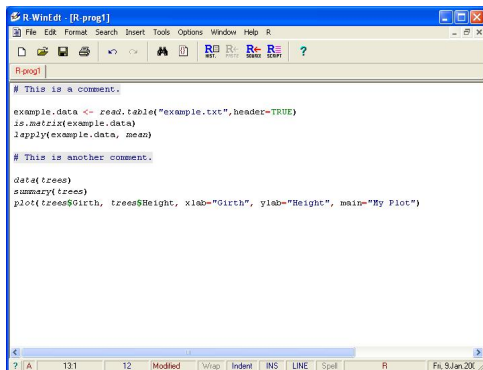
Go into the R console and click on Packages > Install package(s) from local zip files.. and navigate to where you saved the zip file. Click on it and click Open.

```
> utils:::menuInstallLocal()  
package 'RWinEdt' successfully unpacked and MD5 sums checked  
updating HTML package descriptions
```

Scripting

To load RWinEdt

```
> library(RWinEdt)
```



The screenshot shows the RWinEdt application window titled "RWinEdt - [R-prog1]". The menu bar includes File, Edit, Format, Search, Insert, Tools, Options, Window, Help, and R. The toolbar contains icons for file operations, editing, and running code. The editor area shows the following R code:

```
# This is a comment.  
  
example.data <- read.table("example.txt",header=TRUE)  
is.matrix(example.data)  
lapply(example.data, mean)  
  
# This is another comment.  
  
data(trees)  
summary(trees)  
plot(trees$Girth, trees$Height, xlab="Girth", ylab="Height", main="My Plot")
```

The status bar at the bottom displays various settings: ? A 13:1 12 Modified Wrap Indent INS LINE Spell R Fri, 9 Jan 200...

To begin typing in commands click on File > New and type in the commands you want to use.

Scripting

The file can be saved to some location on your computer for use later. No need to re-type all commands you used during your session. Gets a .R extension.

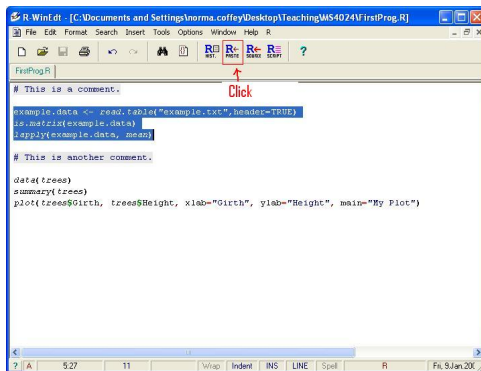
Can copy and paste commands from the script file to the R console.
OR can get R to carry out commands directly from RWinEdt.

To send all commands in the script file, click on



Scripting

To send only some selected commands, highlight the lines you want to use and then click the R Paste button.



Only the highlighted commands are evaluated by R.

Simulation and Probability Distributions

Save the file `ProbDist.R` to your current working folder.

Load `RWinEdt` if it is not already open (`library(RWinEdt)`).

Open the file `ProbDist.R`.

It is possible in R to create random samples from a particular population.

To pick a random sample from a set of numbers use the `sample()` command. See `ProbDist.R`.

Can also evaluate the density function, the cumulative distribution, the quantile function and create a random sample from a particular distribution distribution.

Simulation and Probability Distributions

Distribution	R name	Additional Args
beta	beta	shape1, shape2, ncp
binomial	binom	size, prob
Cauchy	cauchy	location, scale
chi-squared	chisq	df, ncp
exponential	exp	rate
F	f	df1, df2, ncp
...
normal	norm	mean, sd
Poisson	pois	lambda
Student's t	t	df, ncp
uniform	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

Simulation and Probability Distributions

- The prefix:
 - ▶ p: probabilities (cumulative distribution)
 - ▶ q: quantiles (percentage points)
 - ▶ d: density functions (probability for discrete RVs)
 - ▶ r: random (or simulated) values
- The stems:
 - ▶ norm: Normal (Gaussian)
 - ▶ t, chisq, f: Normal test distributions
 - ▶ unif: Uniform distribution, by default $[0, 1]$ range
 - ▶ gamma, cauchy, etc. : various specials
 - ▶ binom, pois, negbin, etc. : various discrete

Simulation and Probability Distributions

One problem when generating random numbers using `sample` or the prefix `r`, is that each time you take a sample, different numbers will be produced. If you want to sample the same numbers, use a command called `set.seed`.

For example,

```
set.seed(9)
sample(1:10, 4)
[1] 3 1 2 8
```

If we want to draw the same numbers again, need to set the seed and then re-draw the sample.

```
set.seed(9)
sample(1:10, 4)
[1] 3 1 2 8
```

If you do not use the `set.seed` function before re-drawing the sample, different numbers will be produced.

```
sample(1:10, 4)
[1] 5 2 4 3
```


Exercises

Simulation and Probability Distributions

1. Draw a random sample of size 100 from the interval $[0,2]$ which contains 200 values. Sample without replacement.
2. Use `dt` to evaluate the density function of the t distribution with 13 degrees of freedom at 20 values in the range -1 to 1.
3. Find $P[X \leq x] = 0.01$ for a t distribution with 9 degrees of freedom.
4. IQ scores are known to have a normal distribution with mean 100 and standard deviation 15. What IQ would you have if you were in the 80th percentile?
5. What IQ would you have if you were in the top 10 percent?
6. What is the probability of having an IQ above 142?
7. Set the seed to "0" and create two samples of size 20 from the standard normal distribution with the same values. Repeat the process but set the seed to your ID number.

Plotting

Plotting

- Simple plotting:
 - ▶ `plot`, `hist`, `pairs`, `boxplot`, ...
- Adding to existing plots:
 - ▶ `points`, `lines`, `abline`, `legend`, `title`, `mtext`, ...
- Interacting with graphics:
 - ▶ `locator`, `identify`
- Three dimensional data:
 - ▶ `contour`, `image`, `persp`, ...
- To see the many possibilities that R offers
 - > `demo(graphics)`

Plotting

Basic plotting function is `plot()`. Possible arguments to `plot()` include:

- `x`, `y` – basic arguments (`y` may be omitted)
- `xlim = c(lo, hi)`, `ylim = c(lo, hi)` – make sure the plotted axes ranges include these values
- `xlab = "x"`, `ylab = "y"` – labels for `x`- and `y`-axes respectively
- `type = "c"` – type of plot (`p`, `l`, `b`, `h`, `S`, ...)
- `lty = n` – line type (if lines used)
- `lwd = n` – line width
- `pch = v` – plotting character(s)
- `col = v` – colour to be used for everything.

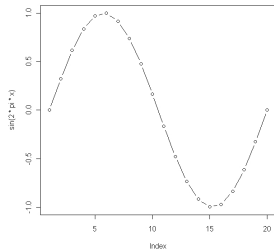
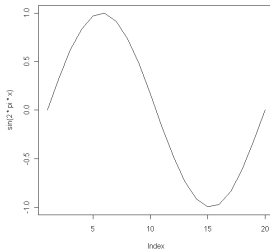
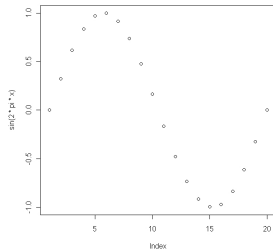
Various examples can be found in the file `Plots.R` in `RWinEdt`.

Plotting

```
x <- seq(0,1,length=20)  
plot(sin(2*pi*x))      # Points
```

```
plot(sin(2*pi*x), type="l")    # Lines
```

```
plot(sin(2*pi*x), type="b")    # Points and lines
```

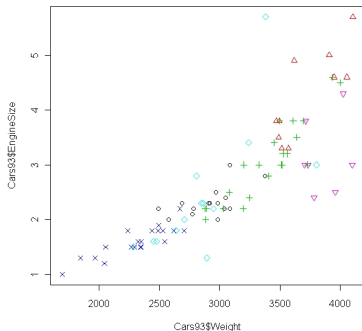


Plotting

Adding titles, lines, points

```
library(MASS)

# Colour points and choose plotting symbols according to a
# levels of a factor
plot(Cars93$Weight, Cars93$EngineSize, col=as.numeric(Cars93$Type),
     pch=as.numeric(Cars93$Type))
```



Plotting

Adding titles, lines, points

```
# Adds x and y axes labels and a title.
plot(Cars93$Weight, Cars93$EngineSize, ylab="Engine Size",
      xlab="Weight", main="My plot")

# Add lines to the plot.
lines(x=c(min(Cars93$Weight), max(Cars93$Weight)), y=c(min(Cars93$EngineSize),
max(Cars93$EngineSize)), lwd=4, lty=3, col="green")

abline(h=3, lty=2)
abline(v=1999, lty=4)

# Add points to the plot.
points(x=min(Cars93$Weight), y=min(Cars93$EngineSize), pch=16, col="red")
```

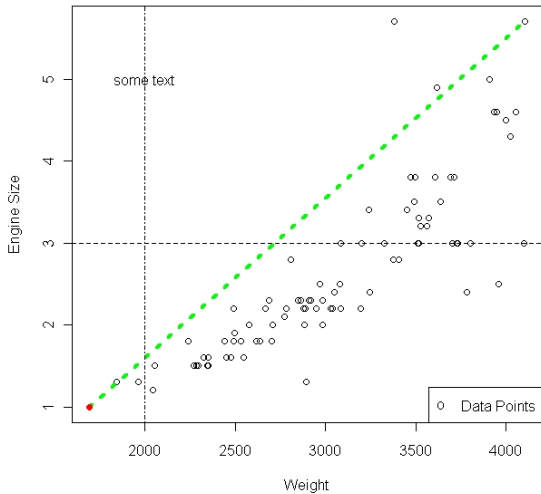
Plotting

Adding titles, lines, points

```
# Add text to the plot.  
text(x=2000, y=5, "some text")  
  
# Add text under main title.  
mtext(side=3, "sub-title", line=0.45)  
  
# Add a legend  
legend("bottomright", legend=c("Data Points"), pch="o")
```


My plot

sub-title



Plotting

Adding regression lines

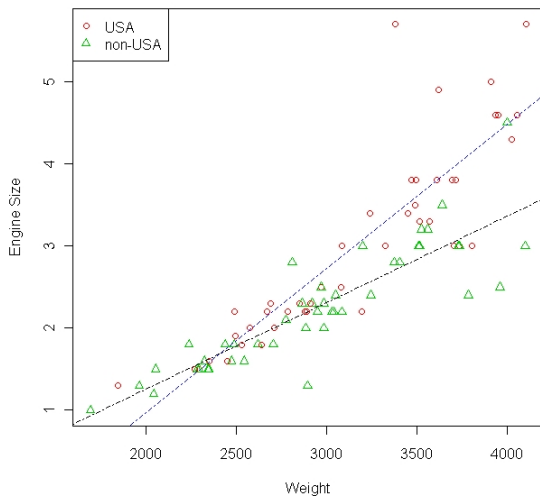
```
levels(Cars93$Origin)
[1] "USA"      "non-USA"

plot(Cars93$Weight, Cars93$EngineSize, pch = (1:2)[Cars93$Origin],
     col = (2:3)[Cars93$Origin], xlab="Weight", ylab="Engine Size")
legend("topleft", legend=levels(Cars93$Origin), pch=1:2, col=2:3)

fm1 <- lm(EngineSize ~ Weight, Cars93, subset = Origin == "USA")
abline(coef(fm1), lty=4, col="blue")

fm2 <- lm(EngineSize ~ Weight, Cars93, subset = Origin == "non-USA")
abline(coef(fm2), lty=4, col="black")
```

Plotting



Plotting

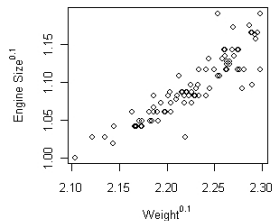
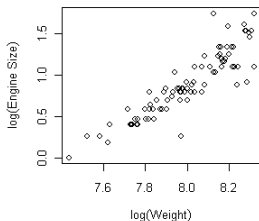
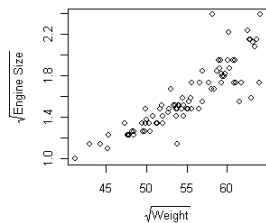
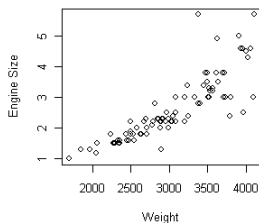
Multiple graphs

```
par(mfrow=c(2,2))          # Will create 4 plots on the same page.  
                             # Two in each row and two in each column.  
  
plot(Cars93$Weight, Cars93$EngineSize, xlab="Weight", ylab="Engine Size")  
  
plot(sqrt(Cars93$Weight), sqrt(Cars93$EngineSize),  
xlab=expression(sqrt(Weight)), ylab=expression(sqrt("Engine Size")))  
  
plot(log(Cars93$Weight), log(Cars93$EngineSize),  
xlab=expression(log(Weight)), ylab=expression(log("Engine Size")))  
  
plot(Cars93$Weight^0.1, Cars93$EngineSize^0.1,  
xlab=expression(Weight^0.1), ylab=expression("Engine Size"^0.1) )  
  
par(mfrow=c(1,1))          # Resets to create a single plot per page.
```

The expression command plots mathematical symbols on the x and y axes. For more information on expression

? plotmath

Plotting



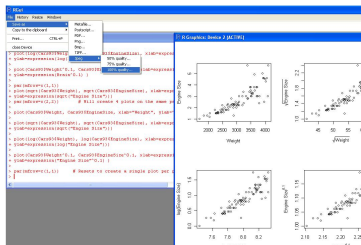
Saving plots

First need to make sure the graphics device is active by clicking on it.

Then click File > Save As > .

Get a number of options...

Mostly use Postscript, PDF and Jpeg.



Plotting

Histograms

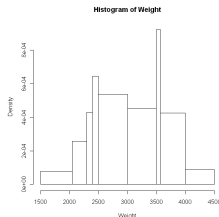
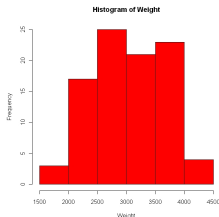
Histograms can be created using the `hist` command.

To create a histogram of the car weights from the `Cars93` data set

```
hist(Cars93$Weight, xlab="Weight", main="Histogram of Weight", col="red")
```

R automatically chooses the number and width of the bars. Can change this by specifying the location of the break points.

```
hist(Cars93$Weight, breaks=c(1500, 2050, 2300, 2350, 2400, 2500, 3000, 3500, 3570, 4000, 4500), xlab="Weight", main="Histogram of Weight")
```



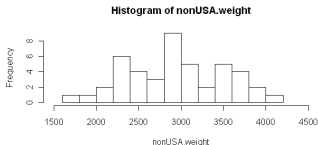
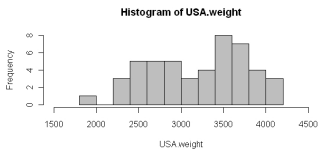
Plotting

Histograms

Histograms for multiple groups.

```
USA.weight <- Cars93$Weight[Cars93$Origin == "USA"]  
nonUSA.weight <- Cars93$Weight[Cars93$Origin == "non-USA"]
```

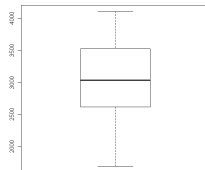
```
par(mfrow=c(2,1))  
hist(USA.weight, breaks=10, xlim=c(1500,4500), col="grey")  
hist(nonUSA.weight, breaks=10, xlim=c(1500,4500))  
par(mfrow=c(1,1))
```



Plotting

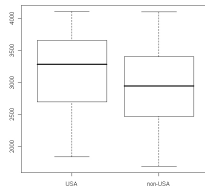
Boxplots

```
boxplot(Cars93$Weight)
```



```
boxplot(Cars93$Weight ~ Cars93$Origin)
```

```
boxplot(USA.weight, nonUSA.weight,  
names=c("USA", "non-USA"))
```

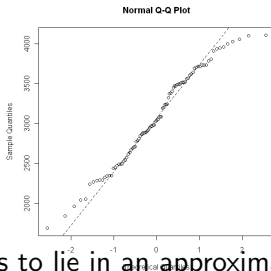


Plotting

Normal probability (Q-Q) plots

To check that data are normally distributed:

```
qqnorm(Cars93$Weight)  
qqline(Cars93$Weight)
```



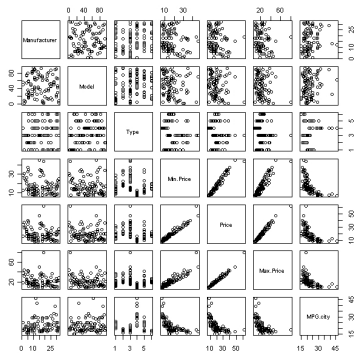
Want all of the points to lie in an approximate straight line (along the dotted line) for a normal distribution.

Plotting

Plots for multivariate data

If your data are stored in a data frame with several columns, the `pairs` command produces pairwise plots of the data in each column, i.e. the data in column 1 vs the data in column 2, column 1 vs column 3, and so on.

```
pairs(Cars93[,1:7])
```

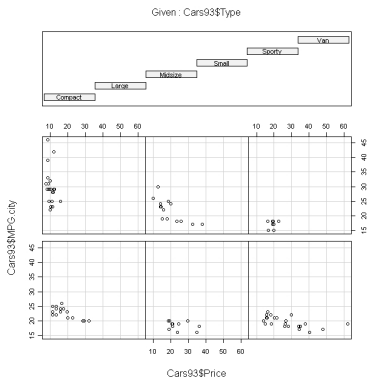


Plotting

Plots for multivariate data

`coplot` is also useful if you have 3 or 4 variables. For example if `a` and `b` are numeric vectors and `c` is a numeric vector or factor, the command `coplot(a ~ b | c)` produces plots of the values of `a` versus `b` for every level of `c`.

```
coplot(Cars93$MPG.city~Cars93$Price|Cars93$Type)
```



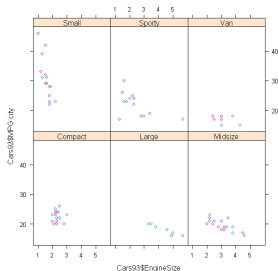
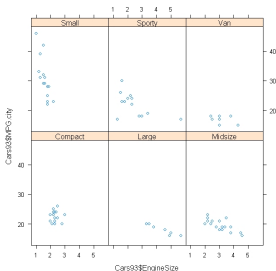
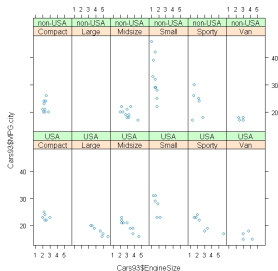
Plotting

Lattice graphs

```
xyplot(Cars93$MPG.city~Cars93$EngineSize|Cars93$Type)
```

```
xyplot(Cars93$MPG.city~Cars93$EngineSize|Cars93$Type*Cars93$Origin)
```

```
xyplot(Cars93$MPG.city~Cars93$EngineSize|Cars93$Type,  
panel=panel.superpose, groups=Cars93$Origin)
```



Plotting

Lattice graphs

Can examine the plotting parameters in xyplot.

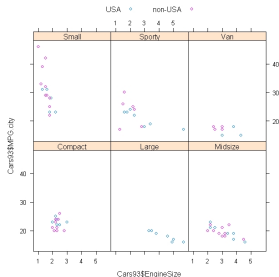
```
pars <- trellis.par.get("superpose.symbol")
pars
$alpha
[1] 1 1 1 1 1 1 1
$cex
[1] 0.8 0.8 0.8 0.8 0.8 0.8 0.8
$col
[1] "#0080ff"    "#ff00ff"    "darkgreen"  "#ff0000"    "orange"     "#00ff00"
[7] "brown"
$fill
[1] "#CCFFFF" "#FFCCFF" "#CCFFCC" "#FFE5CC" "#CCE6FF" "#FFFFCC" "#FFCCCC"
$font
[1] 1 1 1 1 1 1 1
$pch
[1] 1 1 1 1 1 1 1
```

Plotting

Lattice graphs

Can use these to add a key to the plot:

```
# Adds a key  
xyplot(Cars93$MPG.city~Cars93$EngineSize|Cars93$Type,  
panel=panel.superpose, groups=Cars93$Origin, key =  
list(columns = 2, text = list(levels(Cars93$Origin)),  
points = Rows(pars,1:2)))
```



Plotting

Other lattice plots

```
sploM( ~ data.frame) # Scatterplot matrix  
bwplot(factor ~ numeric, ...) # Boxplot  
qqmath(factor ~ numeric, ...) # Q-Q plot  
dotplot(factor ~ numeric, ...) # 1-D display  
stripplot(factor ~ numeric, ...)  
barchar(character ~ numeric, ...)  
histogram( ~ numeric, ...)  
densityplot( ~ numeric, ...) # Smoothed version of histogram
```


Plotting

2-D and 3-D plots

```
? volcano
data(volcano)
x <- 10*(1:nrow(volcano))
y <- 10*(1:ncol(volcano))

# Creates a 2-D image of x and y co-ordinates.
image(x, y, volcano, col = terrain.colors(100),
axes = FALSE)

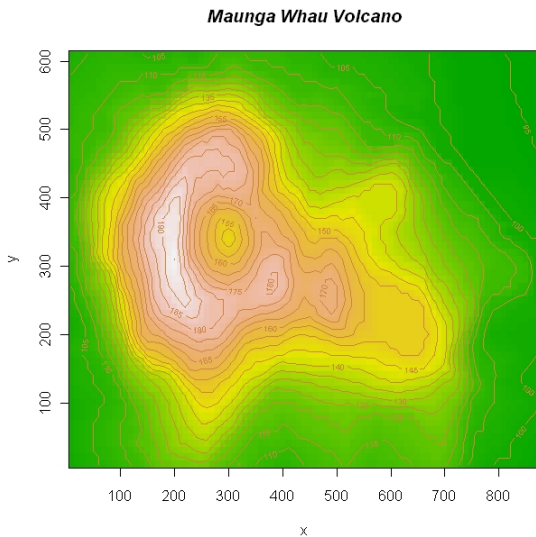
# Adds contour lines to the current plot.
contour(x, y, volcano, levels = seq(90, 200, by=5),
add = TRUE, col = "peru")

# Adds x and y axes to the plot.
axis(1, at = seq(100, 800, by = 100))
axis(2, at = seq(100, 600, by = 100))

# Draws a box around the plot.
box()

# Adds a title.
title(main = "Maunga Whau Volcano", font.main = 4)
```

Plotting



Plotting

2-D and 3-D plots

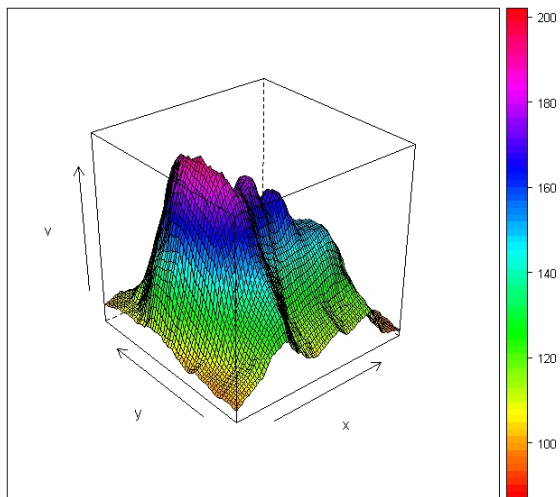
```
dim(volcano)
[1] 87 61
```

```
# Creates a data frame from all combinations of the
# supplied vectors or factors.
```

```
vdat <- expand.grid(x = x, y = y)
vdat$v <- as.vector(volcano)
```

```
wireframe(v ~ x*y, vdat, drape=TRUE, col.regions = rainbow(100))
```

Plotting



Exercises I

Plotting

Type the answers to the following exercises into a script file and run the commands from there.

1. Create a vector `x` of the values from 1 to 20.
2. Create a vector `w <- 1 + sqrt(x)/2`.
3. Create a data frame called `dummy`, with columns `x = x` and `y = x + rnorm(x)*w`. To ensure we all get the same values, set the seed to 0.
4. Create a histogram and a boxplot of `y` and plot them side-by-side on the same graphing region. Label the axes accordingly. Save your results as a Jpeg file.
5. Plot `y` versus `x` using an appropriate plotting command. Put a title on the graph and labels on the axes.

Exercises II

Plotting

6. Enter the command `fm <- lm(y ~ x, data=dummy)` to fit a linear regression model. Add the estimated regression line to the current plot and make it the colour blue.
7. Extract the values of the residuals using `resids <- resid(fm)`. Check that the residuals are normally distributed by creating a Q-Q plot.
8. The `airquality` data set in the base library has columns `Ozone`, `Solar.R`, `Wind`, `Temp`, `Month` and `Day`. Plot `Ozone` against `Solar.R` for each of THREE temperature ranges and each of THREE wind ranges. (Hint: Use `coplot`.)
9. Construct a histogram of `Wind` and overlay the density curve. (Hint: Need `hist`, `lines` and `density`.)

Looping and Functions

Functions

- Have already seen functions in R, e.g.

```
mean(x)
sd(x)
plot(x, y, ...)
lm(y ~ x, ...)
```

- Functions have a name and a list of *arguments* or input objects. For example, the argument to the function `mean()` is the vector `x`.
- Functions also have a list of output objects, i.e. objects that are returned once the function has been run (called).
- A function must be written and loaded into R before it can be used.

Functions are typically written if we need to compute the same thing for several data sets and what we want to calculate is not already implemented in the commercial software yet.

Writing Basic Functions

A simple function can be constructed as follows:

```
function_name <- function(arg1, arg2, ...){  
  
  commands  
  
  output  
}
```

You decide on the name of the function.

The `function` command shows R that you are writing a function.

Inside the parenthesis you outline the input objects required and decide what to call them.

The commands occur inside the `{ }`.

The name of whatever output you want goes at the end of the function.

Comments lines (usually a description of what the function does is placed at the beginning) are denoted by `#`.

Writing Basic Functions

Example

```
sf1 <- function(x){  
  x^2  
}
```

This function is called `sf1`.

It has one argument, called `x`.

Whatever value is inputted for `x` will be squared and the result outputted to the screen.

This function must be loaded into R and can then be called.

Writing Basic Functions

Example

Load the function into R by highlighting the code and clicking the Paste button in RWinEdt.

Type `ls()` into the console. Note that the function now appears.

Can call the function using:

<code>sf1(x = 3)</code>	<code>sf1(3)</code>
<code>[1] 9</code>	<code>[1] 9</code>

To store the result into a variable `x.sq`

<code>x.sq <- sf1(x = 3)</code>	<code>x.sq <- sf1(3)</code>
<code>> x.sq</code>	
<code>[1] 9</code>	

Writing Basic Functions

Example

```
sf2 <- function(a1, a2, a3){  
  x <- sqrt(a1^2 + a2^2 + a3^2)  
  return(x)  
}
```

This function is called `sf2` with 3 arguments.

The values inputted for `a1`, `a2`, `a3` will be squared, summed and the square root of the sum calculated and stored in `x`. (There will be no output to the screen as in the last example.)

The `return` command specifies what the function returns, here the value of `x`.

Will not be able to view the result of the function unless you store it.

```
sf2(a1=2, a2=3, a3=4)
```

```
sf2(2, 3, 4) # Can't see result.
```

```
res <- sf2(a1=2, a2=3, a3=4)
```

```
res <- sf2(2, 3, 4) # Need to use this.
```

```
res  
[1] 5.385165
```

Argument Matching

How does R know which arguments are which? Uses *argument matching*.

Argument matching is done in a few different ways.

- The arguments are matched by their positions. The first supplied argument is matched to the first formal argument and so on, e.g. when writing `sf2` we specified that `a1` comes first, `a2` second and `a3` third. Using `sf2(2, 3, 4)` assigns 2 to `a1`, 3 to `a2` and 4 to `a3`.
- The arguments are matched by name. A named argument is matched to the formal argument with the same name, e.g. `sf2` arguments have names `a1`, `a2` and `a3`. Can do things like `sf2(a1=2, a3=3, a2=4)`, `sf2(a3=2, a1=3, a2=4)`, etc.
- Name matching happens first, then positional matching is used for any unmatched arguments.

Argument Matching

Can also give some/all arguments *default* values.

```
mypower <- function(x, pow=2){  
  x^pow  
}
```

If a value for the argument `pow` is not specified in the function call, a value of 2 is used.

```
mypower(4)  
[1] 16
```

If a value for `pow` is specified, that value is used.

```
mypower(4, 3)  
[1] 64  
  
mypower(pow=5, x=2)  
[1] 32
```

More Complex Functions

The following function returns several values in the form of a list:

```
myfunc <- function(x)
{
  # x is expected to be a numeric vector
  # function returns the mean, sd, min, and max of the vector x

  the.mean <- mean(x)
  the.sd <- sd(x)
  the.min <- min(x)
  the.max <- max(x)

  return(list(average=the.mean,stand.dev=the.sd,minimum=the.min,
              maximum=the.max))
}
```

More Complex Functions

```
x <- rnorm(40)
res <- myfunc(x)
res
$average
[1] 0.29713
```

```
$stand.dev
[1] 1.019685
```

```
$minimum
[1] -1.725289
```

```
$maximum
[1] 2.373015
```

To access any particular value use:

```
res$average
[1] 0.29713
```

```
res$stand.dev
[1] 1.019685
```


Exercises

Functions

1. Write a function that when passed a number, returns the number squared, the number cubed, and the square root of the number.
2. Write a function that when passed a numeric vector, prints the value of the mean and standard deviation to the screen (Hint: use the `cat()` function in R.) and creates a histogram of the data.
3. Management requires a function that calculates the sum of the lengths of 3 vectors. Write the function.

if Statement

Have already encountered *implicit looping* when using the apply family of functions.

Conditional execution: the if statement has the form

```
if (condition){           # Brackets can be omitted if only one command
  expr_1                  # to be carried out.
}
else {
  expr_2
}
```

The condition must evaluate to a logical value, i.e. TRUE or FALSE. If the condition == TRUE, expr_1 is carried out, which can consist of a single command or multiple commands. If the condition == FALSE, expr_2 is carried out.

if Statement

Can also have longer if statements:

```
if (condition1){  
    expr_1  
}  
else if (condition2){  
    expr_2  
}  
...  
else {  
    expr_n  
}
```

If `condition1 == TRUE`, `expr_1` is executed and the checking stops. If `condition1 == FALSE`, moves on to `condition2` and checks if that condition is met. If `condition2 == TRUE`, `expr_2` is executed and checking stops. If `condition2 == FALSE`, moves on to the next condition and so on until all conditions have been checked.

The final `else` is executed if none of the previous conditions have returned a value of `TRUE`.

if Statement

Usually the logical operators `&&`, `||`, `==`, `!=`, `>`, `<`, `>=`, `<=` are used as the conditions in the `if` statement.

The following function gives a demonstration of the use of `if... else`.

```
comparisons1 <- function(number)
{
  # if ... else
  if (number != 1)
  {
    cat(number,"is not one\n")
  }
  else
  {
    cat(number,"is one\n")
  }
}

> comparisons1(1)      > comparisons1(20)
1 is one               20 is not one
```

if Statement

The following demonstrates the use of

if ... else if ... else

```
comparisons2 <- function(number)
{
  if (number == 0)
  {
    cat(number,"equals 0\n")
  }
  else if (number > 0)
  {
    cat(number,"is positive\n")
  }
  else
  {
    cat(number,"is negative\n")
  }
}
```

```
> comparisons2(0)
0 equals 0
```

```
> comparisons2(-15)
-15 is negative
```

```
> comparisons2(1)
1 is positive
```

if Statement

This function demonstrates the use of `&&` in the condition. This means that both conditions must be met before a value of `TRUE` is returned.

```
comparisons3 <- function(number)
{
  if ( (number > 0) && (number < 10) )
  {
    cat(number,"is between 0 and 10\n")
  }
}
```

```
> comparisons3(-1)
> comparisons3(9)
9 is between 0 and 10
```

```
> comparisons3(10)
```

ifelse Statement

A vectorised version of the if statement is `ifelse`. This is useful if you want to perform some action on every element of a vector that satisfies some condition.

The syntax is

```
ifelse( condition, true expr, false expr )
```

If `condition == TRUE`, the `true expr` is carried out. If `condition == FALSE`, the `false expr` is carried out.

```
x <- rnorm(20, mean=15, sd=5)
```

```
x  
[1] 23.608513 14.424667 12.306040 14.291568 18.522846 14.514071 22.004400  
[8] 24.658249 11.697999 16.344976 22.110389 8.455789 19.672274 22.393680  
[15] 11.449034 17.288859 14.839597 14.484774 18.636589 22.670548
```

```
ifelse(x >= 17, sqrt(x), NA)
```

```
[1] 4.858859      NA      NA      NA      4.303818      NA      4.690885  
[8] 4.965707      NA      NA      4.702169      NA      4.435344      4.732196  
[15]      NA      4.157987      NA      NA      4.317012      4.761360
```

for Loops

Repetitive execution: for loops, while loops and repeat loops.

To loop/iterate through a certain number of repetitions a for loop is used. The basic syntax is

```
for(variable_name in sequence) {  
  command  
  command  
  command  
}
```

A simple example of a for loop is:

```
for(i in 1:5){  
  print(sqrt(i))  
}  
[1] 1  
[1] 1.414214  
[1] 1.732051  
[1] 2  
[1] 2.236068
```


for Loops

Another example is:

```
n <- 20
p <- 5
value <- vector(mode="numeric", length=n)
rand.nums <- matrix(rnorm(n*p), nrow=n)
for(i in 1:length(value)){
  value[i] <- max(rand.nums[i,])
  print(sum(value))
}
```

The first four lines create variables `n` and `p` with values 20 and 5 respectively, a numeric vector called `value` with length 20 and a matrix of $20 \times 5 = 100$ random numbers, called `rand.nums`, with 20 rows.

The `for` loop performs 20 loops and stores the maximum value from each row of `rand.nums` into position `i` of the vector `value`.

The sum of the current numbers in `value` is also printed to the screen.

for Loops

Can also have *nested* for loops. Indenting your code can be useful when trying to “match” brackets.

```
for(variable_name1 in sequence) {  
  command  
  command  
  for(variable_name2 in sequence) {  
    command  
    command  
    command  
  } # ends inner for loop  
} # ends outer for loop
```

It should be noted that `variable_name2` should be different from `variable_name1`, e.g. use `i` and `j`. Using the same name will reset the counter each time and result in an infinite loop!!

for Loops

Load the function `simple.nesting` from `Loops.R` and call the function using

```
simple.nesting(num.fam=5, num.child=3).
```

The file `nest.dat` will be created in your current working directory. Open this file and explore the contents.

for loops and multiply nested for loops are generally avoided when possible in R as they can be quite slow. We will use in simulation examples later in the course.

while Loops

The while loop can be used if the number of iterations required is not known beforehand. For example, if we want to continue looping until a certain condition is met, a while loop is useful. The following is the syntax for a while loop:

```
while (condition){  
    command  
    command  
}
```

The loop continues while `condition == TRUE`.

```
niter <- 0  
num <- sample(1:100, 1)  
while(num != 20) {  
    num <- sample(1:100, 1)  
    niter <- niter + 1  
}  
niter
```

apply, lapply, sapply

However, the use of `for` structures is not the optimal option in R. There is a set of functions, called `apply`, that optimise the computation time when repeat a calculation in a vector, list, matrix or `data.frame`:

```
apply(vector, way, function)
```

`apply` is a function requiring three arguments. The first one must be a matrix or `data.frame` object, the second one indicates wheather computation will be performed by rows (1) or by columns (2). The third argument indicates the function to be applied. Extra arguments from that function can be passed through other arguments.

apply, lapply, sapply

Let's illustrate how to use apply in a simple case. Let's imagine we have a matrix:

```
> rmatrix <- matrix(rexp(200, rate=.1), ncol=20)
> dim(rmatrix)
```

```
[1] 10 20
```

To get the mean of each row we can do:

```
> apply(rmatrix, 1, mean)
```

```
[1] 7.271715 9.426293 13.923954 7.619087 7.267820 5.485142 8.817405
[8] 12.130877 9.918925 11.726472
```

And to get the mean of each column:

```
> apply(rmatrix, 2, mean)
```

```
[1] 4.725450 12.449909 11.025990 16.567596 6.481833 8.225318 7.596624
[8] 12.153077 7.037020 12.420729 6.286882 9.569098 6.345784 8.501153
[15] 7.899592 6.485849 7.600887 15.124320 15.394828 5.283442
```

apply, lapply, sapply

The same idea applies for `lapply` and `sapply` in the case of analyzing list or vectors, respectively.

`lapply` applies a function to each element of a list (NOTE: a `data.frame` can also be seen as a list) and returns a list:

```
> lapply(c(1,2,3), function(x){ return(x*2) })
```

```
[[1]]
```

```
[1] 2
```

```
[[2]]
```

```
[1] 4
```

```
[[3]]
```

```
[1] 6
```

The same works for `sapply`, but returning a vector:

```
> sapply(c(1,2,3), function(x){ return(x*2) })
```

```
[1] 2 4 6
```

Exercises

Looping and Functions

1. For each of the following code sequences, predict the result. Then do the computation:

(a)

```
answer <- 0
for(j in 3:5) { answer <- j + answer }
```

(b)

```
answer <- 10
j <- 0
while(j < 5) {
  j <- j + 1
  if(j == 3)
    next
  else
    answer <- answer + j*answer
}
```

2. Add up all the numbers for 1 to 100 in two different ways: using a for loop and using sum.
3. Create a vector `x <- seq(0, 1, 0.05)`. Plot x versus x and use `type="l"`. Label the y-axis "y". Add the lines x versus x^j where j can have values 3 to 5 using either a for loop or a while loop.