A handbook on

R- Language: A programming software for statistical computing and graphics

.

Click **Next**, now we can see the component window (Fig.1. 9) Through which we can select the components to be installed (default is custom installation).

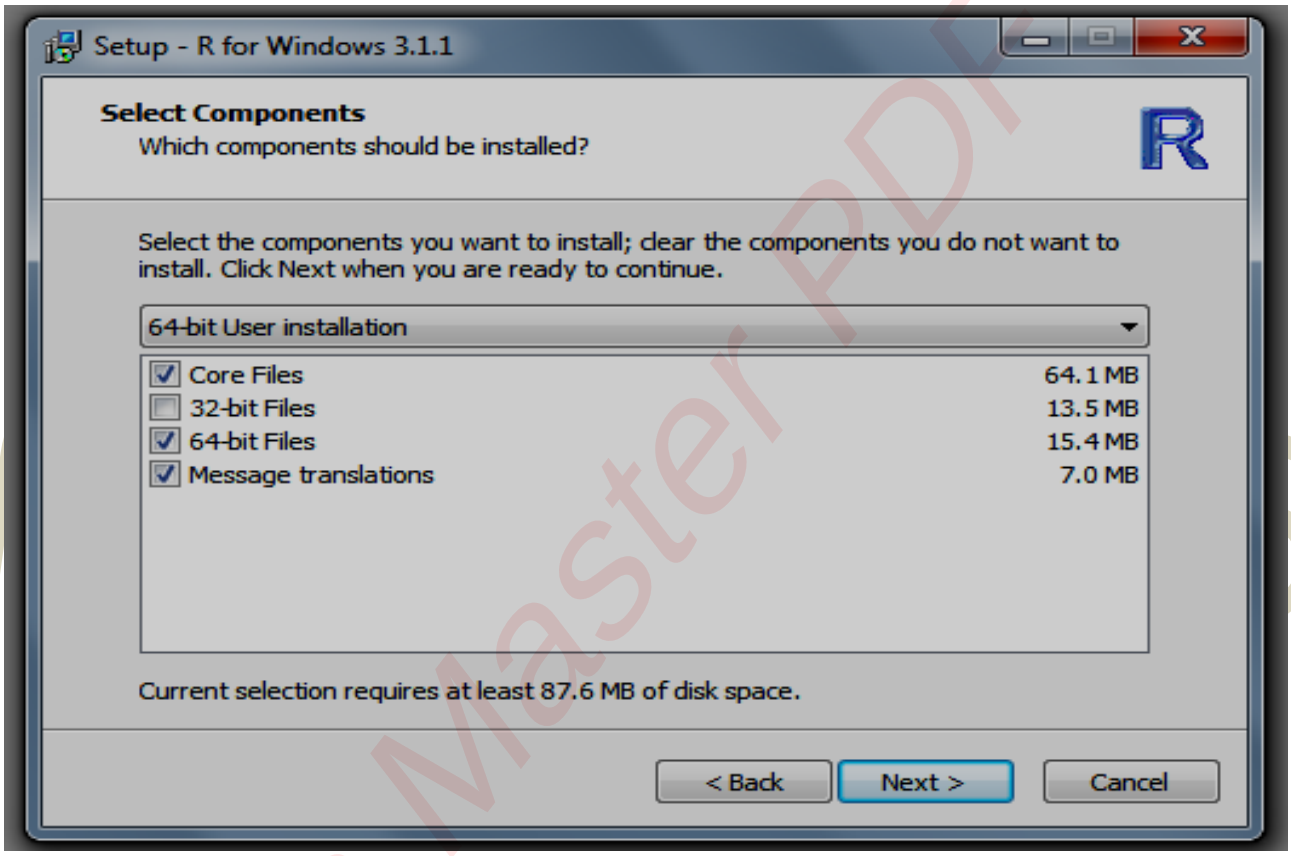


Fig.1. 9: Components Window

After selecting the files which is to be installed click **Next,** this opens the startup options window (Fig. 1.10). Select **Yes** or **No** (Default is No). "Yes" allows initialization by customized startup whereas "No" accepts defaults.

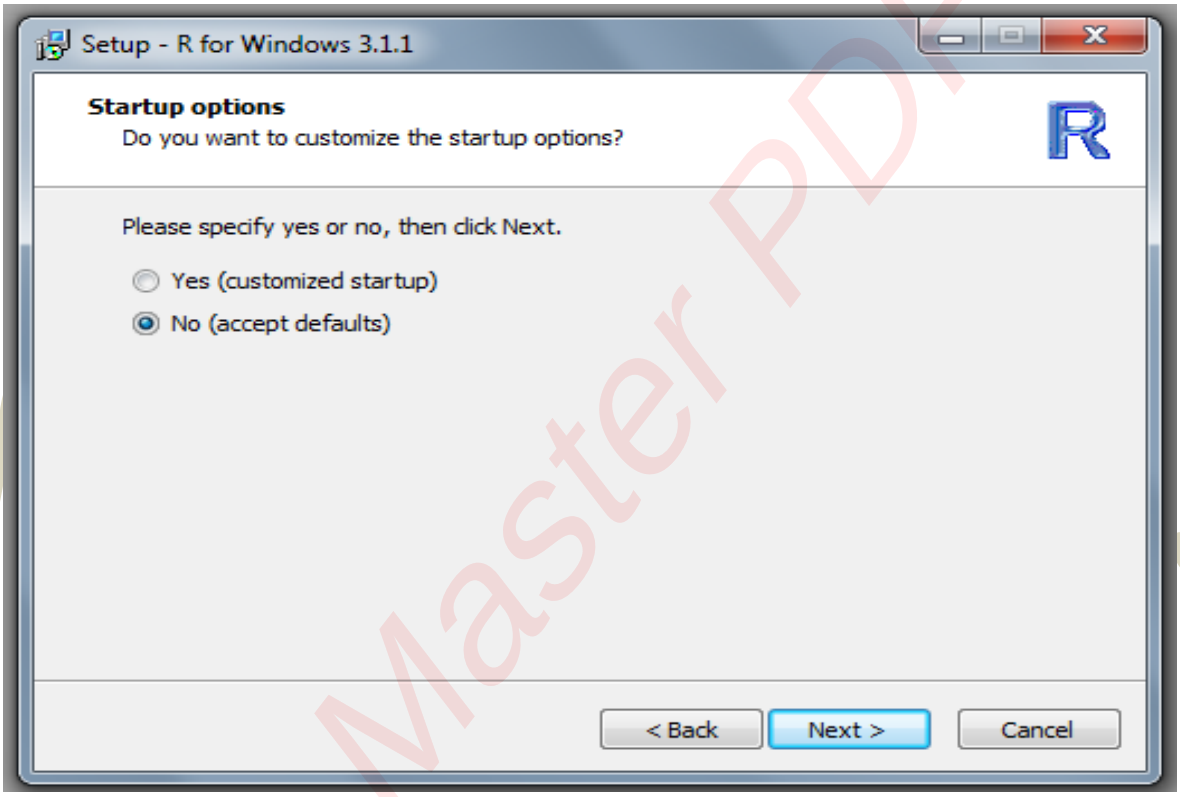


Fig 1.10: Startup Options Window

Click **Next** displays the start menu folder (Fig. 11) where all the program shortcuts will be stored. If one wants to select different folder, then click browse button and choose the folder where it should be stored.

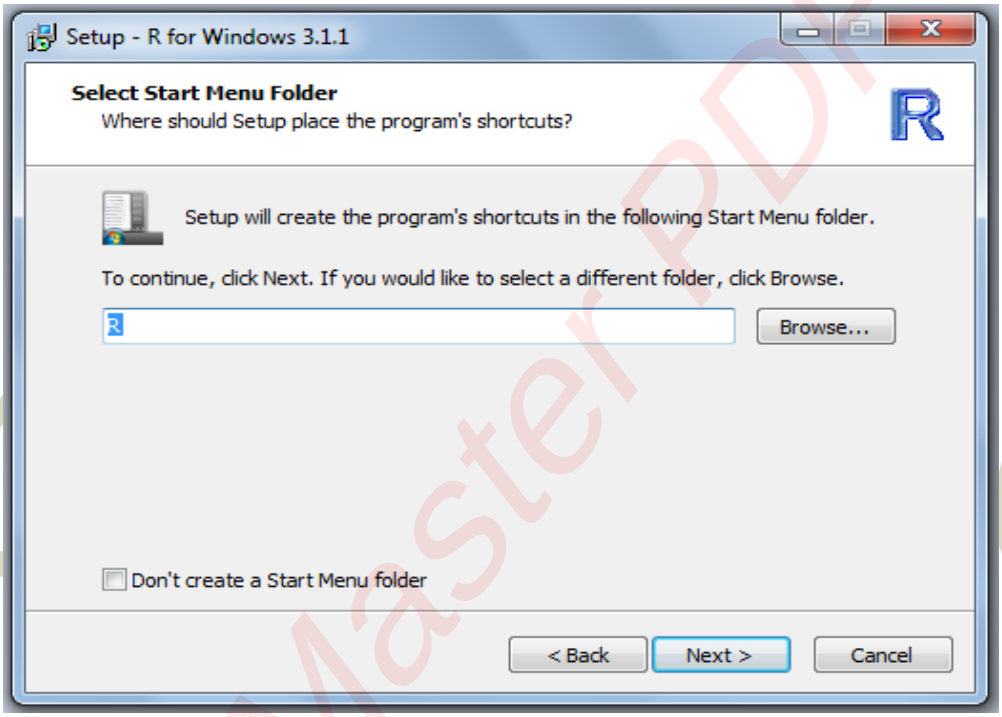


Fig. 1.11: Start Menu Folder Window

Once **Next** button is clicked, this opens another window for selecting additional tasks such as, creating desktop icon, creating a quick launch icon and registry entries (Fig.1.12). If one needs the R icon on desktop, then select the check box "create a desktop icon".

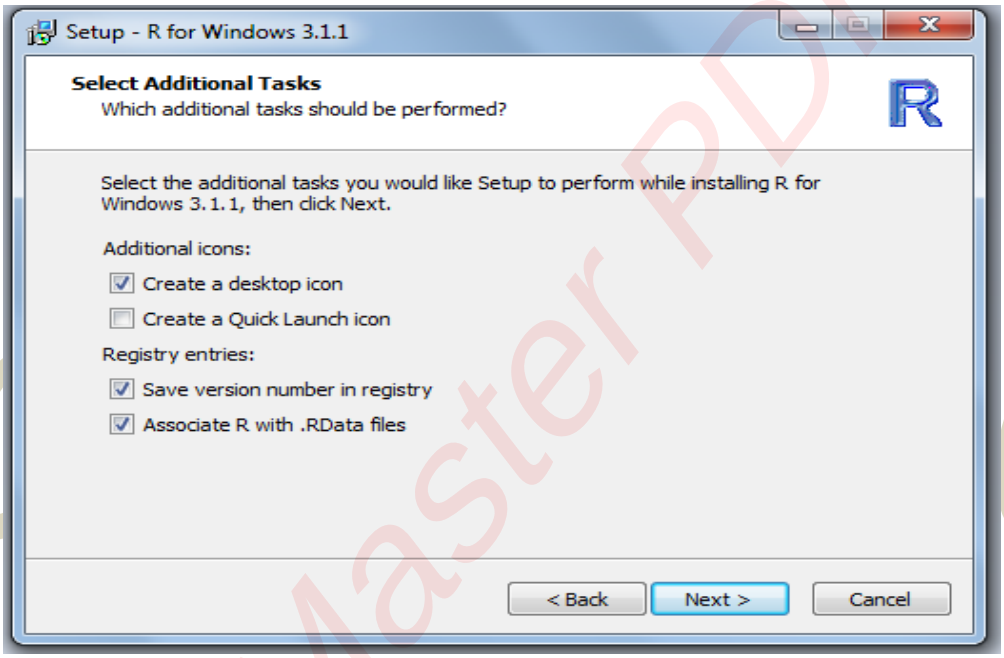


Fig. 1.12: Additional Tasks Windows

By Clicking Next, the installation process will start and within the next few minutes R software will be installed in the computer, in the selected location (Fig. 1.13).

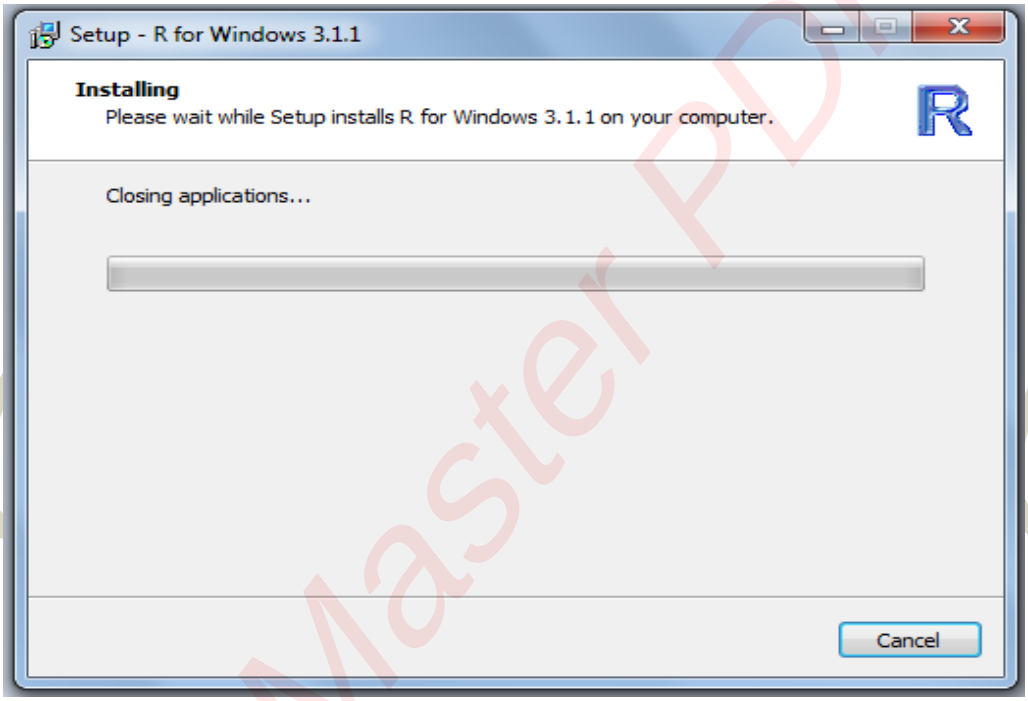


Fig. 1.13: Install Process Window

Once successfully installed, an exit setup window will appear as seen in Fig. 1.14. Click finish button to finish the installation.

Fig. 1.14: Exit setup Window

R- ELEMENTS

R package is a combination of different elements such as R-console, R-scripts, Graphical devices, R parse and R evaluator, workspace, R- import – Export and R packages. User interacts with R through R-console. When a user types a command (or statement or expression) at the prompt (>), the R evaluator reads, parses (i.e., analyzes the syntax), executes the corresponding R expressions and returns the value of the expression.

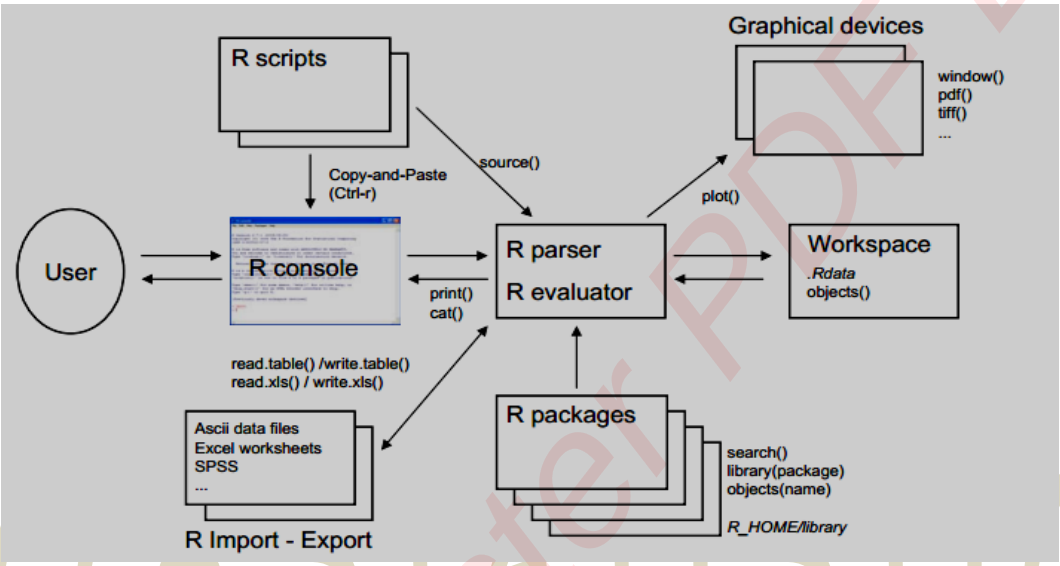


Fig. 15: Elements in R

R-Console

The R-console is a simple interface to interact with the R-evaluator. It interprets (or evaluates or executes) the typed-in command and prints the resulting value back in the R console.

R-Evaluator

The R evaluator is a core element of R. It reads the expressions written in R-language and interprets (evaluate) them.

The Workspace

The workspace represents the memory space that contains the user defined objects (e.g., variables, functions, data). It must be saved at the end of the session (.Rdata file). All the variables created in R are stored in a common workspace. To see which variables are defined in the workspace, one can use the function ls ( ).

```
> ls()
 [1] "Cluster_Data" "D"            "D1"           "Da"           "DC"
 [6] "DCF"          "DI"           "DS"           "hc"           "rf"
>
```

The Scripts (.R files)

The scripts contain the programs of the user (function definitions, analyses, etc.). They are simple text files and can be edited with any text processor. The expressions must be pasted in the R console to be executed.

Evaluation in R

Suppose user types the following text (R expression) in the console

```
> sqrt(7+2)
```

It is evaluated as follows:

R expressions are translated into an internal tree-like structure by the parser (Fig. 1.16). The R-Evaluator will evaluate the internal structure created by the parser. Typically, this evaluation will yield a value that is returned. The returned value is in general, printed by the R console, by an implicit call to the function print.

```
[1] 3
> |
```

Flowchart of the above process is shown in Fig. 1.16.

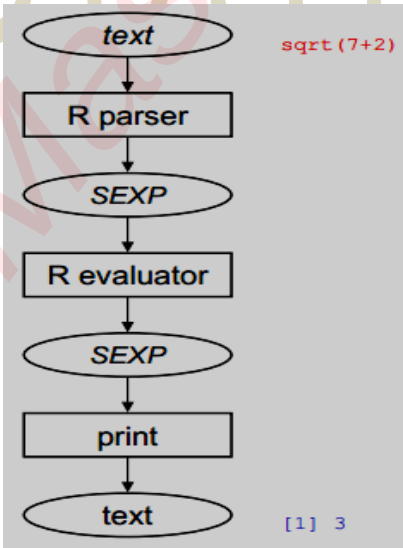


Fig. 1.16.: Flowchart of evaluation process

STARTING AND QUITTING R

In MS-Windows, the R installer will have created a start menu item **(Start -> All Programs -> R - > R x64 3.1.1)** and an icon for R on the desktop (If activated "create desktop icon" in Fig. 1.12). Double clicking the R icon, starts the R program by opening console.

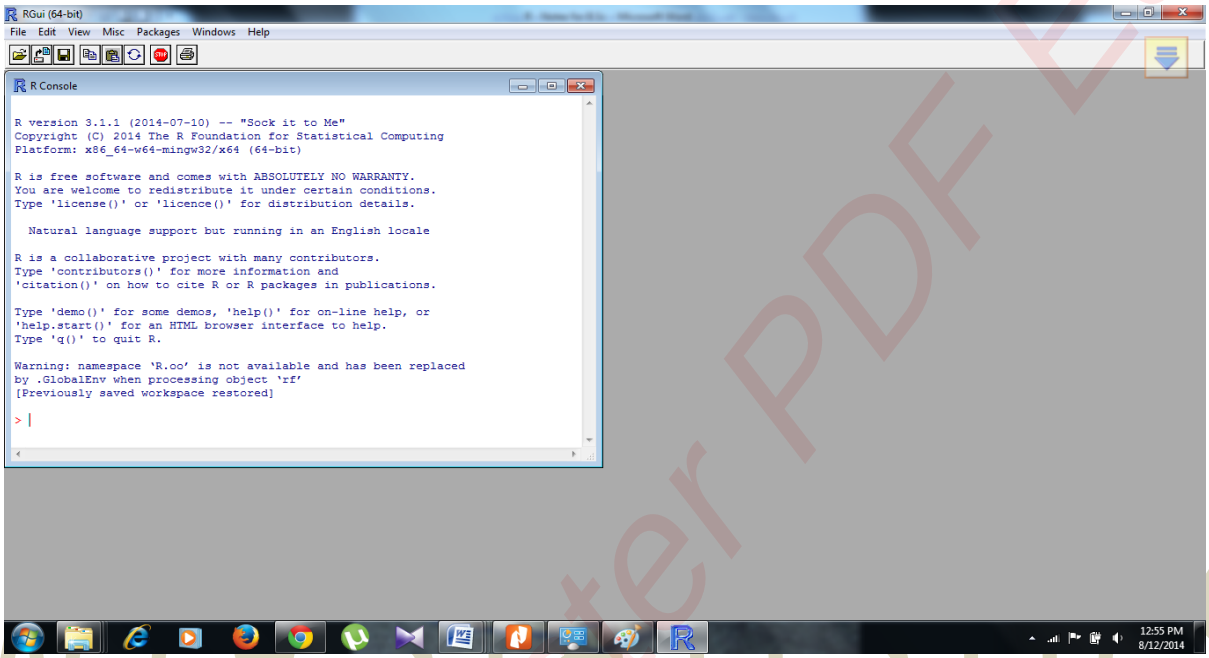


Fig. 1.17: RGui Interface to the Window version of R

Fig.1.17 shows the RGui (R Graphical User Interface) for windows versions of R. The most important element of RGui is the R console window, which initially contains an opening message followed by a line with a command prompt (greater than (>) symbol).

> *The R system is mainly command-driven, with the user typing in text and asking R to execute it. Menu-based interfaces (SPSS, SYSTAT, etc.,) are very convenient when applied to a limited set of commands, from a few, to one or two hundred. However, a command-line interface is open ended. If you want to program a computer to do something that no one has done before, you can easily do it by breaking down the task into the parts that make it up, and then building up a program to carry it out. This may be possible in some menu-driven interfaces, but it is much easier in a command-driven Interface.*

Interaction with R takes place at the command prompt. When ">" symbol appears, one can begin typing. R can be used simply like a calculator. For example

```
> 5 + 7
[1] 12
>
```

Right hand side of the prompt (>) typed 5 + 7 and by pressing enter key we get the result as 12 preceded by 1 inside the square brackets. The [1] indicates that this is the first (and in this case only) results from the command. In some commands multiple values will be displayed, and each line of the results will be labeled to help in interpreting the output. For example, a sequence of integers from 20 to 60 are displayed as follows

```
> seq(20:60)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
>
```

The first line starts with the first value, and so is labeled [1]; second line starts with 26, and so it is labeled as [26]. Few more examples, to show R works like a calculator

```
> 5 - 7
[1] -2
>
> 5 * 7
[1] 35
>
> 5 / 7
[1] 0.7142857
>
> # Exponential can be computed in two ways
>
> 5 ^ 7
[1] 78125
>
> 5 ** 7
[1] 78125
>
```

The pound sign (#) signifies a comment. Text followed by the # is ignored by the interpreter. R does not have options to comment more than one line. If the comment exists more than one line then we have to start the next line with #. For example

```
> # Exponential can be computed in two ways
> # by using ^ and **
>
```

R session can be quit by typing **q ( )** and hitting the enter key. Once it is done, Question dialog box will be displayed with three options **Yes**, **No** and **Cancel** (Fig. 1.18). By clicking "yes", the workspace image is saved in the default workspace area. If "No" button is clicked workspace image will not be saved. Hitting the Cancel

button allows us to continue current R session. The workspace image contains a record of the computation done by the user and may contain some saved results.

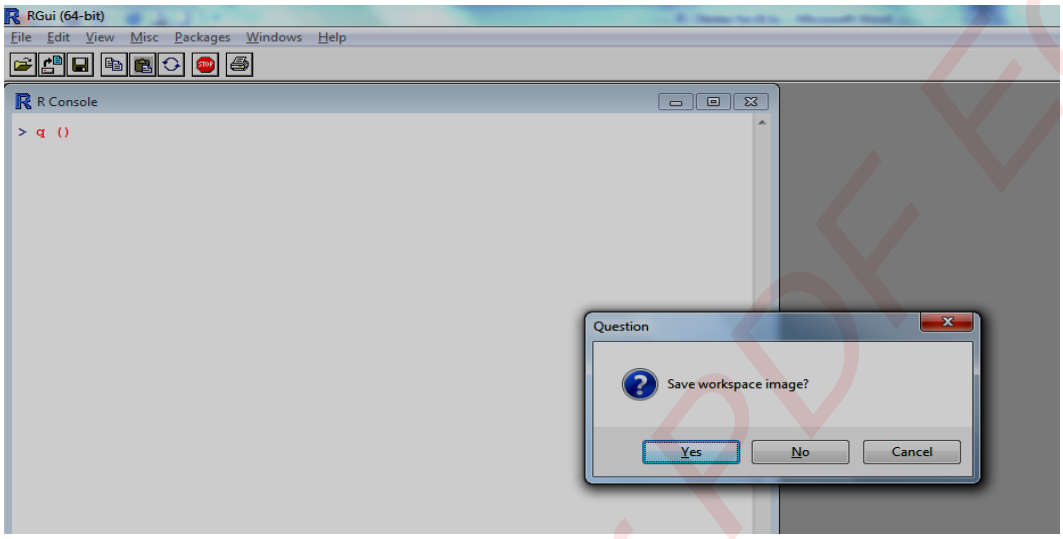


Fig.1.18 Exiting R session

While writing the quit command q ( ), if user omits the parentheses, then in the console window below result will be displayed.

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<bytecode: 0x000000001531ce48>
<environment: namespace:base>
>
```

This is because “q” is a function that is used to tell R to quit. Typing “q” alone tells the R compiler to show the contents of function q. By typing the function q ( ) telling R to call the function q, the function is about to quit R. Everything in R done through calling functions.

GETTING HELP IN R

R has a strong built-in help facility. There are a number of ways to invoke help in R. One of the simplest ways is by clicking the help button on the toolbar of the RGui window (Fig. 1.19) and can get help under the different headings. Suppose users want any study materials regarding R, it can be got from manual (in PDF) under help menu.

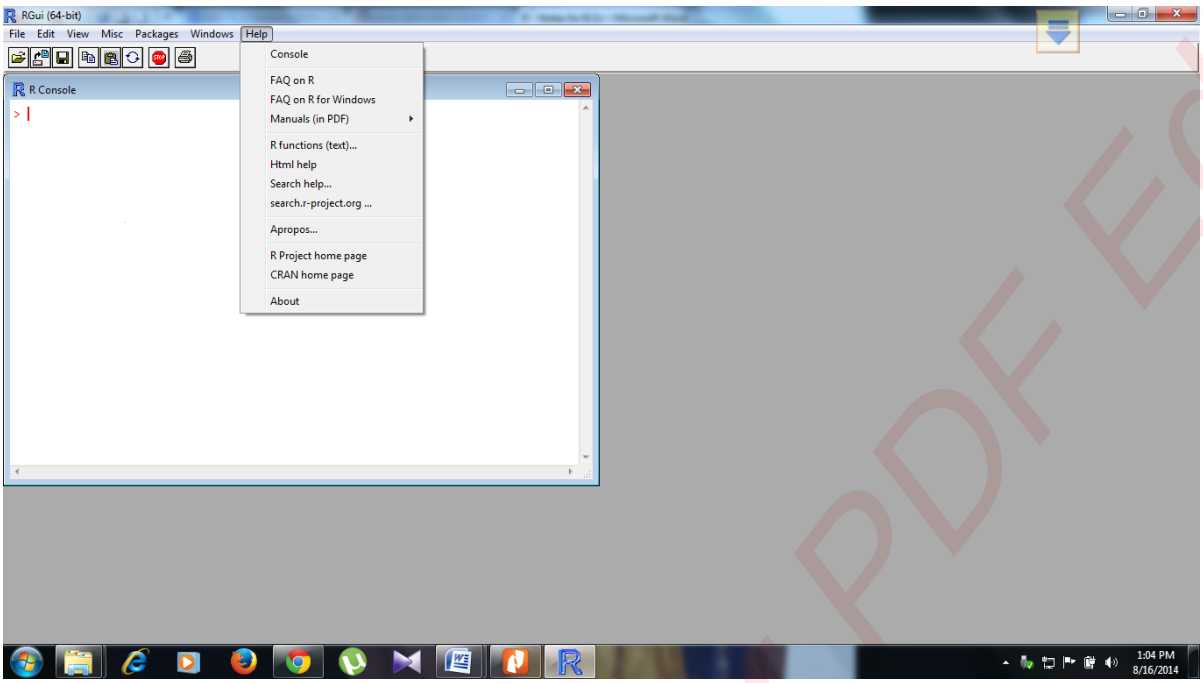


Fig. 1.19 Help menu in R

Another way is to use the help command. If one wants to get help regarding the named functions, it can be done so easily by calling help function. For example, named function is **mean**

```
> help(mean)
starting httpd help server ... done
>
```

On hitting the enter key after typing help function, html web page will be displayed and from which you can understand how the mean function works in R (fig.1.20).

Another way is to use question mark after the prompt ">" in the R-console window. For example, if users know the syntax of the command they are looking for (like mean), then they can type as follows

```
> ? mean
>
```

and the help information will show up on screen in Html format.

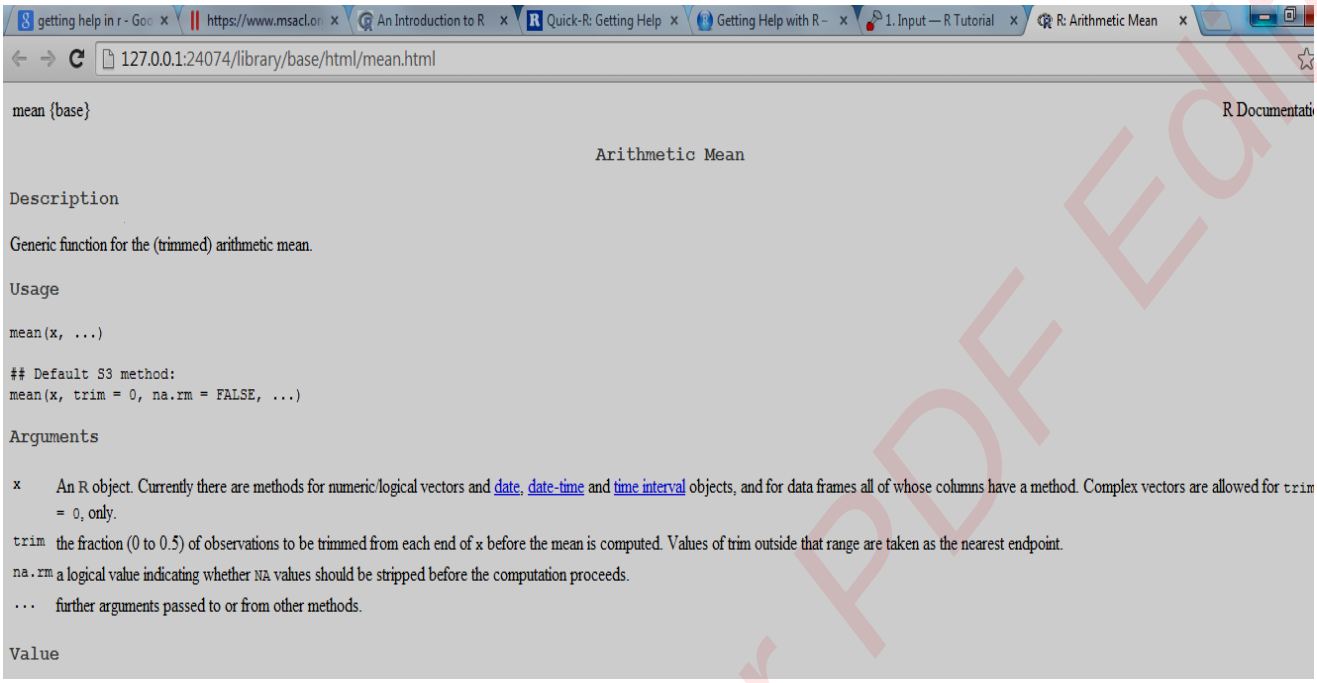


Fig. 1.20 Help html page

Even without knowing the official name of the command, one can get help from R. For example, if one wants to find out the command for standard deviation but does not know its official name (sd). Then user can type as below in R console and by pressing enter key, opens an html web page regarding standard deviation.

```
> help.search("standard deviation")
starting httpd help server ... done
>
```

Other useful functions are find ( ) and apropos ( ). The **find** function gives the information about the packages is in:

```
> find ("lowess")
[1] "package:stats"
>
```

while **apropos** returns a character vector giving the names of all objects in the search list that match your (potentially partial) enquiry:

```
> apropos("mean")
 [1] ".colMeans"      ".rowMeans"      "colMeans"       "kmeans"
 [5] "mean"           "mean.Date"      "mean.default"   "mean.difftime"
 [9] "mean.POSIXct"   "mean.POSIXlt"   "rowMeans"       "weighted.mean"
>
```

SEARCHING THE WEB FOR HELP

Tremendous amount of information about R is available in the web. One can directly use the function **RSiteSerach ("Key Phrase")** in the R console. The RSiteSearch function will open a browser window and direct it to the search engine on the R project web page http://search.r-project.org/. For example, the below call would start a search for Correlation

```
> RSiteSearch("Correlation")
```

Fig. 1.21 Search results from RSiteSearch function

This is quite handy for doing quick web search without leaving R. However, the search scope is limited to R documents and the mailing – list archives.

http://rseek.org : This is a Google custom search that is focused on R-Specific websites. It provides a wider search. Its virtue is that it harnesses the power of the Google search engine while focusing on sites relevant to R, which eliminates the extraneous results of a generic Google search. The beauty of RSeek.org is that it organizes the results in a useful way.

Fig. 1.22 shows the results of visiting RSeek.org and searching for "canonical correlation". The left side of the page shows general results for search R sites. The right side is a tabbed display that organizes the search results into several categories: Introduction, Task Views, Support lists, Functions, Books, Blogs and Related tools.

Fig. 1.22 Search results from RSeek.org

http://stackoverflow.com/ : It is a Question and Answer site, which means that anyone can submit a question and experienced users, will give answer, often there are multiple answers to each question. Readers vote on the answers, so good answers tend to appear on the top. This creates a rich database of searchable Question and Answer dialogues. Stack Overflow is strongly problem oriented, and the topics lean towards the programming side of R.

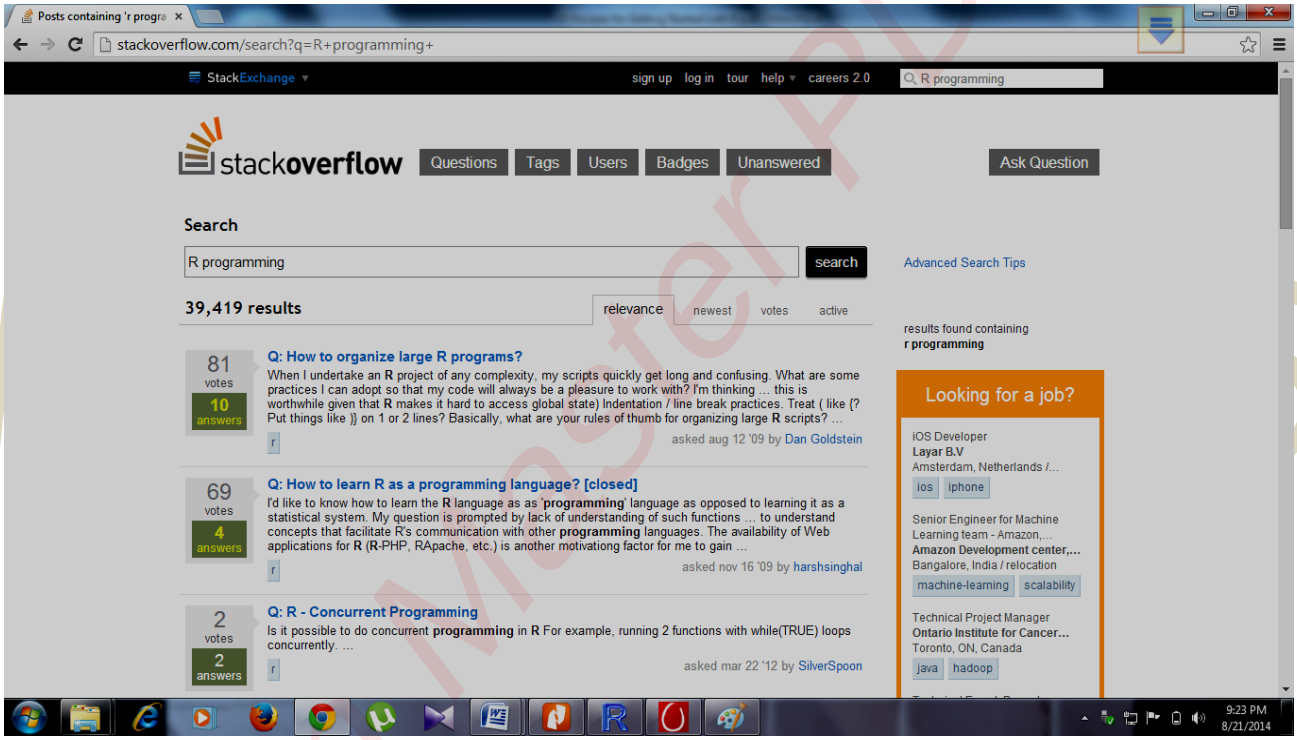


Fig. 1.23 Search results from stackoverflow.com

To see the worked example of mean ( ), just type the following function

```
> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))
[1] 8.75 5.50
>
```

To know the arguments of the function, type as below

```
> args(mean)
function (x, ...)
NULL
>
```

Some of the useful help functions in R:

```
> help.start()
If nothing happens, you should open
'http://127.0.0.1:19375/doc/html/index.html' yourself
>
```

The above command launches a web browser that allows the help pages to be browsed with the hyperlinks.

If you want to know the stored or installed packages

```
> .packages(all.available=T)
 [1] "base"         "boot"         "class"        "cluster"      "codetools"
 [6] "compiler"     "datasets"     "foreign"      "graphics"     "grDevices"
[11] "grid"         "KernSmooth"   "lattice"      "MASS"         "Matrix"
[16] "methods"      "mgcv"         "nlme"         "nnet"         "parallel"
[21] "rcom"         "rpart"        "rscproxy"     "spatial"      "splines"
[26] "stats"        "stats4"       "survival"     "tcltk"        "tools"
[31] "translations" "utils"
>
```

To get information regarding the available functions look in: base

```
> help(package= "base")
>
```

To see what global options are available:

```
> ?options
>
```

To see what graphics options are available:

```
> ?par
>
```

CASE SENSITIVE

R language is **case sensitive**, so **A** and **a** are not same. They refer to different variable. For example,

```
> A = c(12, 13, 14)
> A
[1] 12 13 14
> a
Error: object 'a' not found
>
```

In the above statement, **A** was assigned three values 12, 13 and 14. By typing **A** all the three values were displayed. In the next line **a** was typed. R shows an error message as object 'a' not found. This shows that R treats **A** and **a** differently. Therefore, user must be little careful while writing the functions or calling the objects. Suppose user typed Q ( ) instead of q ( ), R displays an error message as below

```
> Q ( )
Error: could not find function "Q"
>
```

R COMMANDS

In R, elementary commands consist of either **expressions** or **assignments.** If an expression is given a command, it is evaluated, printed and the value is lost. Expressions typically involve variable references, operators such as +, function calls and so on. An assignment also evaluates an expression and passes the value to a variable but the result is not automatically printed. Commands are separated either by a semi-colon (;), or by a newline. Elementary commands can be grouped together into one compound expression by using braces ('{' and '}'). If a command is not complete at the end of a line, R will give a different prompt, by default +. On second and subsequent lines and continue to read input until the command is syntactically complete. Command lines entered at the console are limited to about 4095 bytes (not characters).

R provides a mechanism for recalling and re-executing previous commands. The vertical arrow keys on the keyboard can be used to scroll forward and backward through a command history. Once a command is located in this way, the cursor can be moved within the command using the horizontal arrow keys, and characters can be removed with the DEL key or added with the other keys.

SCRIPTS

While writing functions and other multi-line sections of input, one will find it useful to use a text editor rather than execute everything directly at the command line. R has its own built-in editor. It is accessible from the RGui menu bar. Click **File** from menu bar and select **new script** or press **Crtl+N (Control key and N together)**. At this point R will open a new window titled **Untitled – R Editor.** One can type and edit in this, to execute a line which is typed, just highlight them and press **Ctrl + R**. The lines are automatically transferred to the command window and executed.

The typed text in the scripts dialog box can be saved by pressing **Ctrl + S** and it should be named. The saved file will have the **.R** file extension. Saved files can be opened in subsequent session by just clicking on **File - > Open Script**; all the saved files will be displayed, in it select the desired file.

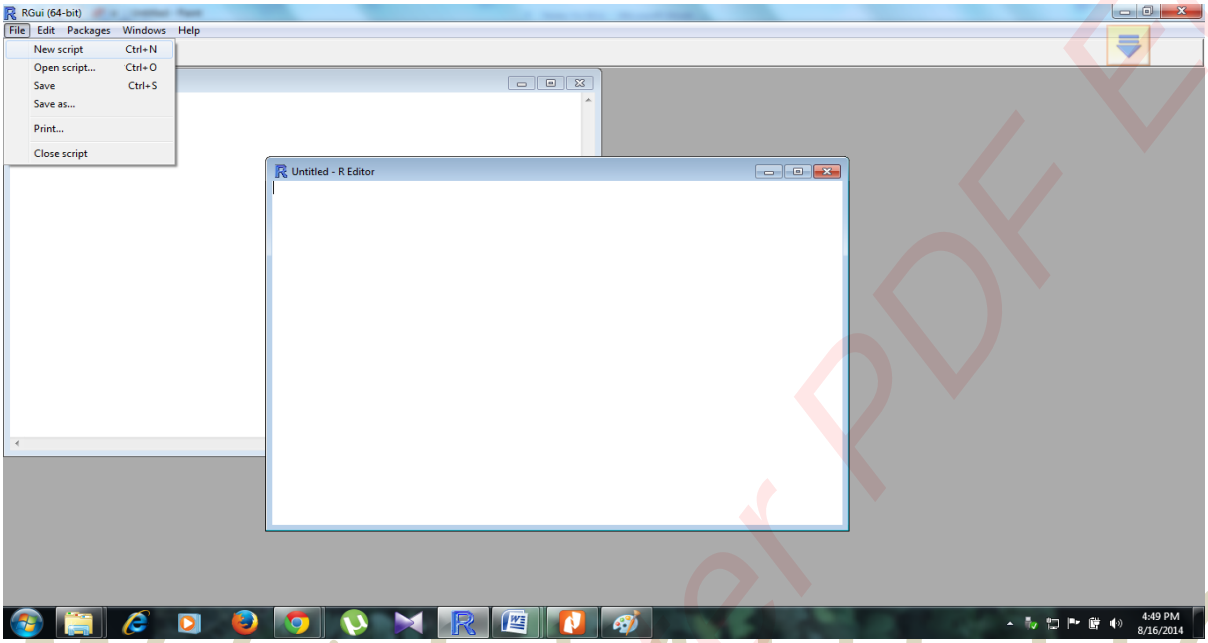


Fig. 1.24 Script Window

DATA EDITOR

R has data editor option. It can be selected from the menu bar **Edit - > Data editor**. Editing data set should be loaded first. For example, suppose you want to edit data set **iris** from package **datasets.** Load the dataset using following commands

```
> library(datasets)
> data(iris)
>
```

Now select data editor from Edit menu. Once clicked data editor a new dialog box will be displayed asking for the name of the data frame or matrix (**name of editing data set**). Enter **iris** and click **OK.** Data editor dialog box will be displayed and hence can edit the data (Fig.1.25).

Fig. 1.25 Data Editor Window

Alternatively, one can do this from the command line using the fix function (e.g. **fix(data.frame.name)).**

```
> fix(iris)
>
```

The window looks like an Excel spreadsheet, and one can change the contents of the cells, navigating with the cursor or with the arrow keys.

PACKAGES

Packages are collections of **R** functions, data and compiled code in a well-defined format. The directory where packages are stored is called the **library**. **R** comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used. To see which packages are installed, enter the following command in r console.

```
> library ( )
>
```

To load a particular package, use the following command. For example, MASS package is loaded as

```
> library(MASS)
>
```

INSTALLING PACKAGES IN R

Packages for R can be installed through command line and also RGui. To install the A3 package do the following

THROUGH COMMAND LINE

Type "install.packages("*package name*")" in the R-console and press enter. Once enter key is pressed CRAN mirror list box opens. Select the nearest mirror from it and click OK. Then package will be installed with their dependences.

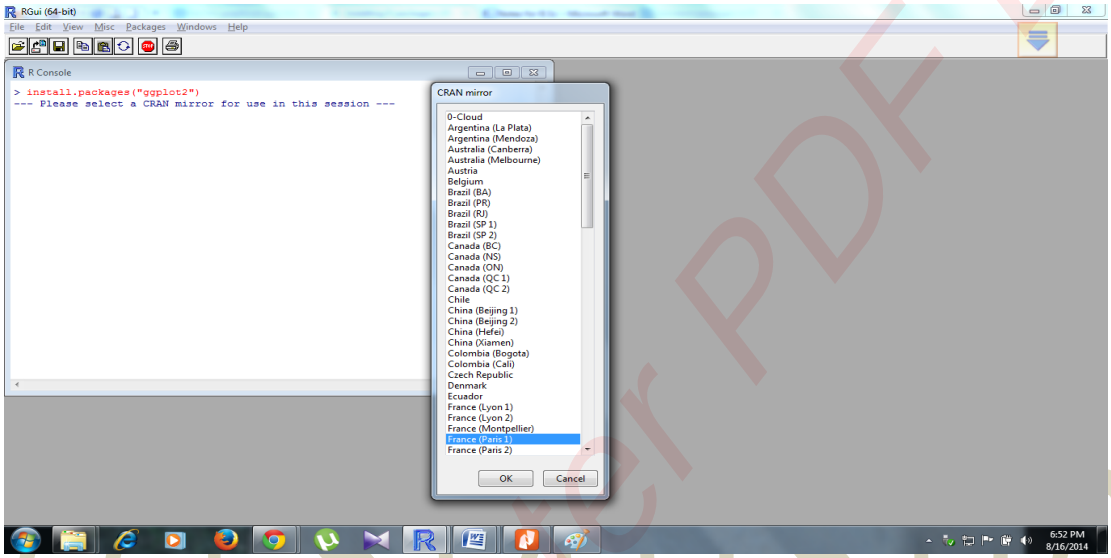


Fig. 1.26 CRAN Mirror dialog box

THROUGH RGUI

Packages can also be installed through RGui. From menu bar select **Packages** and from the submenu select **install package(s).** Then CRAN mirror list box opens, from it select the nearest mirror and click ok. Then packages dialog box will be opened. In that select the package which you wish to install. For example, A3 package to be installed, then select A3 and click Ok. A3 package will be downloaded within few minute and installed in R dictionary.

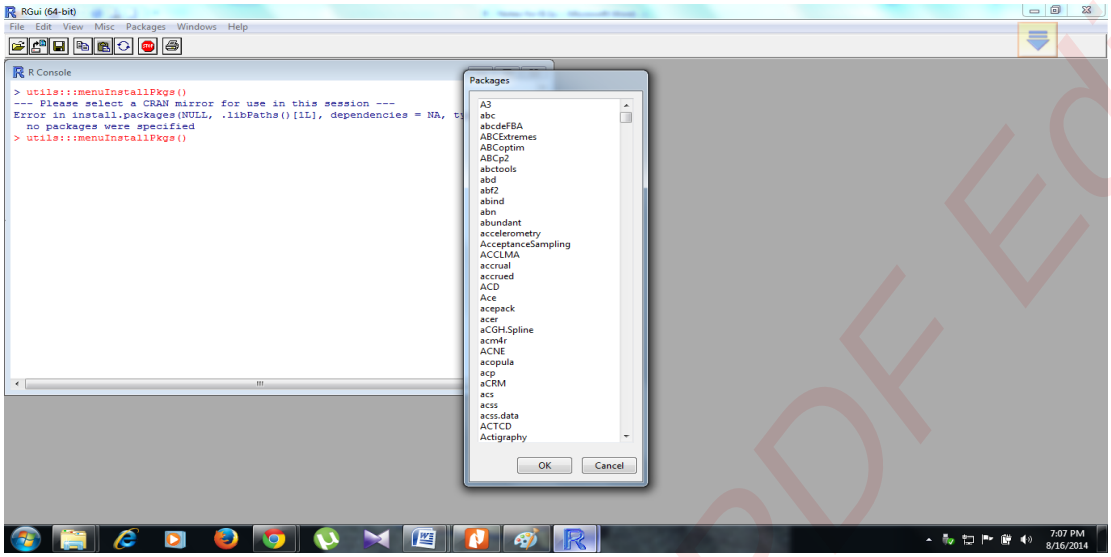


Fig. 1.27 Packages dialog box

In R there is another option to install packages from local zip file. To do this, first packages (in zip file) should be downloaded and stored in any local drive of the computer system. Now, from menu bar select **packages** and from dropdown list select **install package(s) from local zip files.** It opens the **select files** dialog box **as fig 1.28.** Choose the package which is to be installed from the local drive it stored and click open. The installation begins automatically.

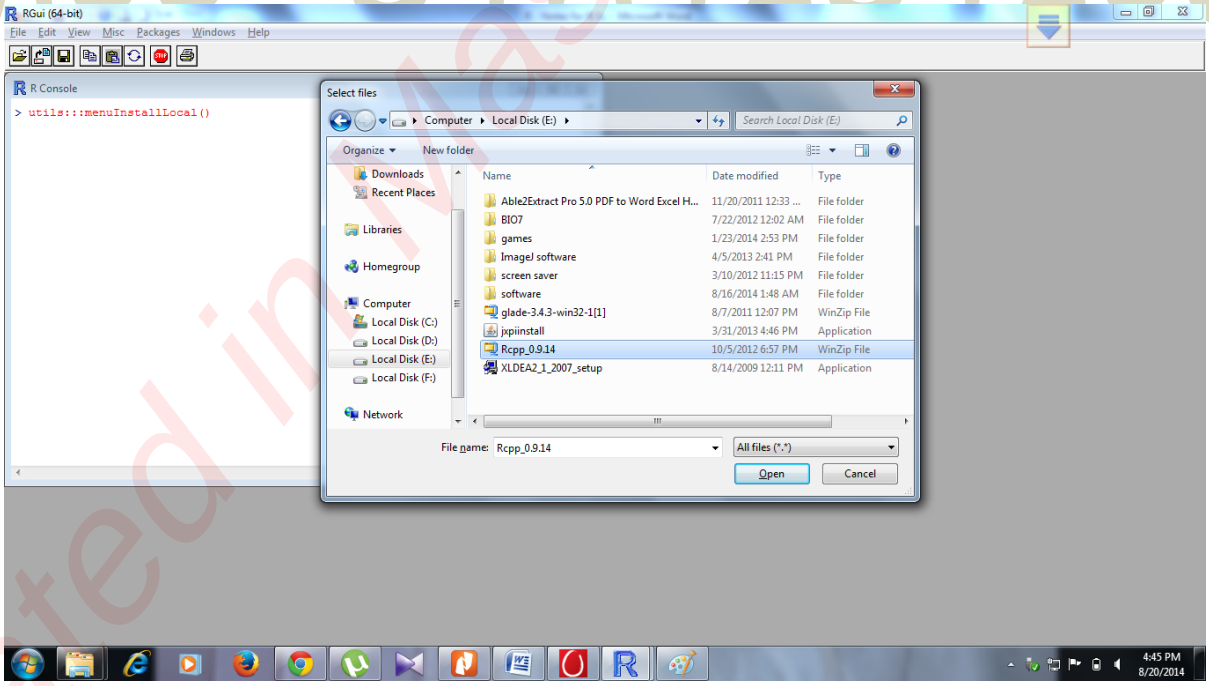


Fig. 1.2.8 Installing packages from local zip files

To see which packages is currently loaded, use the following function

```
> search ()
```

The standard (or base) packages are considered a part of the R source code. They contain the basic functions, the datasets, standard statistical and graphical functions that allow R to work.

There are thousands of contributed packages for R, written by various authors. Some of these packages implement specialized statistical methods, others give access to data or hardware. Most packages are available for download from CRAN (http://CRAN.R-project.org/ and its mirrors) and other repositories such as Bioconductor (http://www.bioconductor.org/) and Omegahat (http://www.omegahat.org/).

Removing objects and Clearing R console

The entities that R creates and manipulates are known as **objects**. These may be variables, array of numbers, character strings and functions. During R session, objects are created and stored by name. The R common **object ( )** or **ls ( )** (*ls* ( ) *stands for list of objects in my work space*) can be used to display the names of the objects which are currently stored within R.

```
> objects()
 [1] "Cluster_Data" "D"            "D1"           "Da"           "DC"
 [6] "DCF"          "DI"           "DS"           "hc"           "rf"
>
>
>
> ls()
 [1] "Cluster_Data" "D"            "D1"           "Da"           "DC"
 [6] "DCF"          "DI"           "DS"           "hc"           "rf"
>
```

To remove objects the function rm ( ) is used. For removing object DI, D from the workspace permanently.

```
> rm(D, DI)
>
```

re-run the command ls( ), D and DI objects will not be displayed.

```
> ls()
[1] "Cluster_Data" "D1"           "Da"           "DC"           "DCF"
[6] "DS"           "hc"           "rf"
>
```

To remove all the objects

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

Now to verify, re-type function ls ( ), this displays as character (0) instead of the objects stored. This implies that there is nothing in the workspace.

```
> ls()
character(0)
> 
```

To clear the screen have to press Ctrl and L together (ctrl + l). This will not detect the objects which are created in last sessions.

EXERCISE UNIT 1

1. What is the difference between help ( ) and help.search ()?
2. When you type the command help.search(correlation) you get an error message. Why?
3. What is the use of workspace? How you create a work space.
4. Execute X <- letters in R. What did you get?
5. Execute X <- LETTERS in R. What did you get?

UNIT – 2 OBJECTS AND DATA STRUCTURE

Data is the most valuable for statisticians, without which no analysis can be done. For most of the statistical softwares data set should be in very specific format such as data table or data frame. When comparing to all other softwares R is more flexible in data handling. Even though R has data frame format it does not stick strictly in it. It is flexible to keep the data in contingency table, or a list, or a matrix, or a single vector.

DATA SET

For example, information regarding AGE, HEIGHT, WEIGHT, RACE, YEAR, SAT score was collected from five college students A, B, C, D and E. In these students A, B, C, D and E are the **cases or subject** and AGE, HEIGHT, WEIGHT, RACE, YEAR, SAT are called **variables**.

Name	AGE	HEIGHT	WEIGHT	RACE	YEAR	SAT
A	19	67	155	Asian	Second	1174
B	18	78	180	African	First	867
C	22	57	142	Asian	Final	1057
D	21	66	160	African	Second	834
E	18	69	167	African American	First	1002

Cases or subject are going into ROWS in the table and each variable have its own column (having header row is optional). Two different types of variables are in this data set *viz.,* numeric and categorical variables. AGE, HEIGHT, WEIGHT and SAT comes under **numeric** and **RACE** and **YEAR** are under **categorical**. In R, often categorical variable are called as **FACTOR**. Categorical variables have natural ordering (Year: First, Second and Final).

Sometimes categorical variables of such a type are called as **ORDERED FACTOR** in R. Usually variables can be divided into 4 types viz., **Nominal**, **Ordinal**, **Interval** and **Ratio**. Factor is nominal and ordered factors are ordinal. Numeric variables are either **Interval** or **Ratio**. In factor or categorical variables mostly few possible values will be seen. These values are called **LEVELS**. For example, a level of RACE is Asian, African and African American. When a variable has different values for everyone, for example, **Name** of the students often called as **Character** variable.

DEFINING DATASET IN R SESSION

There are many ways to explore data in R. We can see it one by one in this section. Let us start with simple one

ASSIGNMENT

There are 3 ways to assign data to an object name in R. They are

i)

```
> X = 5
> |
```

It should be read as X **takes the value** or **is assigned the value** 5. In R, spacing does not a matter, following all formats is valid.

.

```
> X = 5
> X=5
> X         =5
> X         =     5
> X =
+ 5
>
```

Note : To bring 5 in the next line we have to press enter.

ii)

```
> X <- 5
>
```

This also assign value 5 to X. The **ARROW** assignment operator is actually of two symbols < (less than sign) and – (dash or minus). There should not be any space between < and -. The <- tells R to take the number 5 to the right of the symbol and store it in a variable whose name is given on the left. The following are valid format

```
> X <- 5
> X        <-5
> X        <-    5
> X <-
+ 5
> X<-5
>
```

iii)

This is also an arrow assignment, but it enters into opposite direction.

```
> 5 -> X
>
```

The important use of arrow than = is, arrow works either side, but = does not. When using = sign object must be given first followed by the value. When one makes an assignment, R does not print out any information. If we want to see what value a variable has, just type the name of the variable on a line and press the enter key. The output of all the above three assignment will be the same.

```
> X = 5
> X
[1] 5
> X <- 5
> X
[1] 5
> 5 -> X
> X
[1] 5
>
```

Now, we overwrite the value 3 to same object X.

```
> X = 3
>
```

Now, look at the object called 'X' in workspace. Variable 'X' is created and is displayed.

```
> ls ()
[1] "X"
> X
[1] 3
>
```

By entering 'X', the value 3 is displayed not 5. This shows R overwrite things in the workspace without any warning message. If new value is assigned to the same object name that already exists, the old object is gone!.

VARIABLE NAMES

R does not give importance for the variable name, whether it's a variable or complete data objects.

RULES FOR GIVING VARIABLE NAMES IN R

Variable names can be combination of Alphabets, numeric, dot and underscore symbols. Variable names should not start with numbers and underscores. There should not be any space or dashes in variables names.

Note: It is generally safer to confine yourself to letters, numbers, dots, and underline characters and to start your variable names with letters. Try to avoid using names that are also functions and objects in R, for example like "mean". Even if it works, avoid using T and F as variable names because R uses them to mean TRUE or FALSE. There is no limit for creating variable name, but avoid giving too long because it will be difficult while repeating it again and again. R has many built-in data type objects.

Following are some of the **valid** and (different) variable or object names in R

> A
> A23
> A.23
> A_23
> .A
> .A23
> a.23
> a
> a_23
> This.is.a.variable.name
> This_is_a_variable_name
> This.is_a.variable_name
> This is avariable name

Following are some of the **invalid** variable or object names in R

> 1.A
> _A
> This is.a.variable.name
> A-23

OPERATORS IN R

Symbols						Operator names
+	-	*	/	%%	^	Arithmetic
>	>=	<	<=	==	!=	Relational
!	&	|				Logical
~						Model formulae
<-	->					Assignment
$						List indexing (the element name operator)
:						Create a Sequence

Example 2.1: Use R to find all the numbers between 1 and 2000 which are multiples of 317.
Solution:

```
> A = 1:2000
> A[ A%%317 == 0]
[1]  317  634  951 1268 1585 1902
> |
```

Example 2.2: Find all the words with less than 6 or more than 8 characters in the vector c("Maine", "Maryland", " Missouri", "Montana", "Massachusetts", "Michigan", "Minnesota")
Hints: need OR operator | and function nchar( )

```
> A = "Maryland"
> nchar(A)
[1] 8
```

Solution:

```
> A = c("Maine", "Maryland",
+ "Massachusetts", "Michigan", "Minnesota",
+ "Mississippi", "Missouri", "Montana")
> A[nchar(A)>8 | nchar(A)<6]
[1] "Maine"         "Massachusetts" "Minnesota"     "Mississippi"
> |
```

DATA TYPES
DOUBLE: Doubles are used to represent continuous variables like height or weight of students. Double is used to represent the numbers.

```
> X = 8.4
> Y = 4.5 + 6.7
> Z = 3.1415*X
```

To check whether the object is of type double use the function **is.double ("object")** or to know the type of the object use **type of ("Object")**.

```
> typeof(X)
[1] "double"
> typeof(Z)
[1] "double"
> typeof(Y)
[1] "double"
> is.double(X)
[1] TRUE
> |
```

INTEGER
Integers are natural numbers. They can be used to represent counting variables, for example the number of pins in a box.

```
> pinbox = 100
> is.integer (pinbox)
[1] FALSE
> typeof(pinbox)
[1] "double"
> |
```

Note that even the value 100 is of integer but in R it considered as double. So, to write an integer type object in R

```
> pinbox = as.integer (100)
> is.integer(pinbox)
[1] TRUE
>
```

However, one can mix object of type double and integer in a calculation without any problems.

```
> X = as.integer(4)
> typeof(X)
[1] "integer"
> Y = 2.3
> typeof(Y)
[1] "double"
> Z = X/Y
> Z
[1] 1.73913
> typeof(Z)
[1] "double"
>
```

In contrast, in some other programming languages, the answer is of type double. The maximum integer in R is $2^{31} - 1$.

```
> as.integer(2^31 - 1)
[1] 2147483647
> as.integer(2^31)
[1] NA
Warning message:
NAs introduced by coercion
>
```

COMPLEX

To represent complex numbers object of type complex is used. For creating object of type complex use as.complex or complex functions.

```
> COM = as.complex(34+6i)
> COM
[1] 34+6i
> sqrt(COM)
[1] 5.853433+0.51252i
>
> COM2 = complex(3, real = 4, im = 5)
> COM2
[1] 4+5i 4+5i 4+5i
> typeof(COM)
[1] "complex"
>
```

LOGICAL

An object of data type logical can have the value TRUE or FALSE and is used to indicate if a condition is true or false. Such objects are usually the result of logical expressions.

```
> A.1 = 45
> .A2 = 2*10-1
> A.1 > .A2
[1] TRUE
>
```

The result of the function is.double is an object of type logical (TRUE or FALSE).

```
> is.double (3.21)
[1] TRUE
>
```

CHARACTER

A character object is represented by a collection of characters between double quotes (" "). For example: "x", "test character" and "iuiu8ygy-iuhu". One way to create character objects is as follows.

```
> Z = c("A","B","C")
> Z
[1] "A" "B" "C"
>
>
> Mychar1 = "This is a char"
> Mychar1
[1] "This is a char"
>
```

The double quotes indicate that we are dealing with an object of type `character'.

FACTOR

The factor data type is used to represent categorical data (i.e. data of which the value range is a collection of codes). For example: variable `Gender' with values male and female.
variable `blood type' with values: A, AB and O.

An individual code of the value range is also called a level of the factor variable. So the variable `Gender' is a factor variable with two levels, male and female. Sometimes user's confuse factor type with character type. Characters are often used or labels in graphs, column names or row names. Factors must be used when one wants to represent a discrete variable in a data frame and want to analyze it. Factor objects can be created from character objects or from numeric objects, using the function factor. For example, to create a vector of length five of type factor do the following:

```
> Gender = c("male", "male", "female", "male", "female")
>
```

The object 'Gender' is a character object. If we want to change it to factor then

```
> Gender = factor(Gender)
> Gender
[1] male   male   female male   female
```

Use **levels** funtion to see the different levels a factor variable has

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

Note that the result of the levels function is of type character. Another way to generate the Gender variable is as follows:

```
> Gender = c(1,1,2,1,2)
> |
```

The object `Gender' is an integer variable, you need to transform it to a factor.

```
> Gender = factor(Gender)
> Gender
[1] 1 1 2 1 2
Levels: 1 2
> |
```

The object `Gender' looks like an integer, but is not an integer variable. The 1 represents level "1" here. So arithmetic operations on the Gender variable are not possible:

```
> Gender + 3
[1] NA NA NA NA NA
Warning message:
In Ops.factor(Gender, 3) : + not meaningful for factors
> |
```

It is better to rename the levels, so level "1" becomes male and level "2" becomes female:

```
> levels(Gender) = c("male", "female")
> Gender
[1] male   male   female male   female
Levels: male female
> |
```

One can transform factor variables to double or integer variables using the **as.double** or **as.integer** function.

```
> Gender.N = as.double(Gender)
> Gender.N
[1] 1 1 2 1 2
> |
```

The 1 is assigned to the female level only, because alphabetically female comes first. If the order of the levels is of importance, one will need to use ordered factors. Use the function ordered and specifies the order with the levels argument. For example:

```
> Income <- c("High","Low","Average","Low","Average","High","Low")
> Income <- ordered(Income, levels=c("Low","Average","High"))
> Income
[1] High    Low     Average Low     Average High    Low
Levels: Low < Average < High
> |
```

The last line indicates the ordering of the levels within the factor variable.

MISSING DATA AND INFINITE VALUES

Symbol NA (Not Available) is used to represent missing data in R. It is not really a separate data type; it could be a missing double or a missing integer. To check if data is missing, use the function **is.na** or use a direct comparison with the symbol **NA**. There is also the symbol **NaN** (Not a Number), which can be detected with the function **is.nan**.

Example 2.3:

```
> S = as.double(c("12", "15", "ALL"))
> is.na(S)
[1] FALSE FALSE  TRUE
>
```

Example 2.4:

```
> R = sqrt(c(1,-1))
Warning message:
In sqrt(c(1, -1)) : NaNs produced
> is.nan(R)
[1] FALSE  TRUE
>
```

Infinite values are represented by Inf or -Inf. You can check if a value is infinite with the function **is.infinite**. Use is.finite to check if a value is finite.

Example 2.5

```
> X = c(1,0,4)
> Y = c(1,0,0)
> X/Y
[1]   1 NaN Inf
>
```

Example 2.6

```
> L = log(c(0))
> is.infinite(L)
[1] TRUE
>
```


DATA STRUCTURES

The following objects exists in R

VECTORS	MATRICES	ARRAYS
DATA FRAMES	TABLES	LISTS

VECTORS

The simplest data structure in R is the **vector**. A vector is an object that consists of a number of elements of the same type, all doubles or all logical. A vector with the name `x' consisting of four elements of type `double' (10, 5, 3, 6) can be constructed using the function **c**.

To construct a vector, type the following in the console:

```
> x = c(10,5,3,6)
> x
[1] 10  5  3  6
>
```

In programming language, a ***function*** is a piece of code that takes some inputs and does something specific with them. In constructing a vector, you tell the **c( )** function to construct a vector with (10, 5, 3, 6). The entries inside the parentheses are referred to as ***arguments***. One can also construct a vector by using operators. For example,

```
> x = 2+4
> x
[1] 6
> |
```

In the above example mathematical operator + is used to construct the vector. One such very handy operator is called ***sequence,*** and it looks like a colon (:). Suppose one wishes to find the average of first 100 integer values, entering first 100 data values through c ( ) is little difficult. So one can use sequence as below

```
> Z = 1:100
> Z
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
> |
```

Same could be used as arguments inside the c ( ).

```
> Z = c(1:100)
> Z
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
 [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
 [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
 [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
 [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
 [91]  91  92  93  94  95  96  97  98  99 100
> |
```

To find the average of first 100 integer values, function mean ( ) is used

```
> mean(Z)
[1] 50.5
> |
```

The function **c** merges an arbitrary number of vectors to one vector. A single number is regarded as a vector of length one.

Example: 2.7

```
> X = c(23,45,60)
> Y = c("A","B")
> Z = c(X,Y)
> Z
[1] "23" "45" "60" "A"  "B"
> |
```

Note: The merged Z object is a character type.

Example: 2.8

```
> S = c(1:10)
> S
 [1]  1  2  3  4  5  6  7  8  9 10
> D = c(400,320,S)
> D
 [1] 400 320   1   2   3   4   5   6   7   8   9  10
> |
```

Example: 2.9

```
> S = c(1:10)
> D = c(400.45,320.61,S)
> D
 [1] 400.45 320.61   1.00   2.00   3.00   4.00   5.00   6.00   7.00   8.00
[11]   9.00  10.00
> |
```

To make bigger or smaller steps in a sequence, use the **seq( )** function. This function's by argument allows one to specify the amount by which the numbers should increase or decrease. For example,

```
> y = seq(from = 4, to = 20, by = 2)
> y
[1]  4  6  8 10 12 14 16 18 20
> |
```

The following command will yield the same values

```
> y = seq(4,20,length = 9)
> y
[1]  4  6  8 10 12 14 16 18 20
> |
```

REPEAT FUNCTION rep ( )

The function rep repeats a given vector. The first argument is the vector and the second argument can be a number that indicates how often the vector needs to be repeated.

```
> R = rep(1:5,3)
> R
 [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
> |
```

The second argument can also be a vector of the same length as the vector used for the first argument. In this case each element in the second vector indicates how often the corresponding element in the first vector is repeated.

```
> R = rep(1:5, c(3,3,3,3,3))
> R
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
> |
```

```
> R1 = rep(1:5, 1:5)
> R1
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
> |
```

To generate vectors with random elements one can use the functions **rnorm** or **runif**. There are more of these functions.

```
> I = rnorm(6)  # six random standard normal numbers
> I
[1] -2.6843936 -0.3013765 -1.6821599 -2.1804742 -1.6059309 -0.1707436
>
> U = runif(6,3,6)  # Six random numbers between 3 and 6
> U
[1] 5.975984 3.320140 5.323499 3.605357 4.732542 3.833418
> |
```

MATHEMATICAL CALCULATIONS ON VECTORS

Calculations on (numerical) vectors are usually performed on each element. For example, X*X results in a vector which contains the squared elements of X.

```
> X = c(2,4,6)
> V = X*X
> V
[1]  4 16 36
> |
```

Most of the standard mathematical functions are available in R. These functions also work on each element of a vector. For example the logarithm of X:

```
> log(X)
[1] 0.6931472 1.3862944 1.7917595
> |
```

Some of the basic mathematical functions applied on vector

Function Name			Operation
abs			Absolute value
sin	cos	tan	Geometric functions
asin	acos	atan	Inverse geometric functions
sinh	cosh	tanh	Hyperbolic functions
asinh	acosh	atanh	Inverse hyperbolic functions
exp	log	log10	Exponent , natural logarithm and log with base 10
gamma	lgamma		Gamma and log gamma function
round			Rounding
floor	ceiling	trunc	Creates integers from floating point number
sqrt			Square root

THE RECYCLING RULE

It is not necessary to have vectors of the same length in an expression. If two vectors in an expression are not of the same length then the shorter one will be repeated until it has the same length as the longer one. A simple example is a vector and a number (which is a vector of length one).

```
> X = c(2,4,6)
> X
[1] 2 4 6
> S = sqrt(X)
> S
[1] 1.414214 2.000000 2.449490
> A = 2
> A
[1] 2
> R = S + A
> R
[1] 3.414214 4.000000 4.449490
> |
```

In the above example 2 is repeated 3 times until it has the same length of S and then addition of two vectors is carried out.

MATRICES

GENERATING MATRICES

A matrix can be regarded as a generalization of a vector. As with vectors, all the elements of a matrix must be of the same data type. A matrix can be generated in several ways. For example:

Using dim ( ) function

```
> M = 1:21
> dim (M) = c(3,7)
> M
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    7   10   13   16   19
[2,]    2    5    8   11   14   17   20
[3,]    3    6    9   12   15   18   21
> |
```

Using matrix ( ) function

```
> M = matrix (1:21, 3, 7, byrow = F)
> M
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    4    7   10   13   16   19
[2,]    2    5    8   11   14   17   20
[3,]    3    6    9   12   15   18   21
> |
```

By default the matrix is filled by column. To fill the matrix by row specify **byrow = T** as argument in the matrix function.

Syntax:

matrix(data, nrow, ncol) # data is a vector of nrow*ncol values

```
> M = matrix (1:21, 3, 7, byrow = T)
> M
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    1    2    3    4    5    6    7
[2,]    8    9   10   11   12   13   14
[3,]   15   16   17   18   19   20   21
> |
```

One can use the function **cbind ( )** to create a matrix by binding two or more vectors as column vectors.

Syntax:

cbind(d1, d2, ..., dm) # d1, ..., dm are vectors (columns)

```
> cbind (c(1,2,3), c(4,5,6))
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> |
```

The function **rbind ( )** is used to create a matrix by binding two or more vectors as row vectors.

Syntax:

rbind(d1, d2, ..., dm) # d1, ..., dm are vectors (rows)

```
> rbind (c(1,2,3), c(4,5,6))
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
>
```

CALCULATIONS ON MATRIX

A matrix can be regarded as a number of equal length vectors pasted together. All the mathematical functions that apply to vectors also apply to matrices and are applied on each matrix element.

Let us consider the following matrix

```
> MAT = matrix(1:9, nrow = 3, ncol = 3, byrow = T)
> MAT
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
>
```

To find the maximum value in MAT

```
> max(MAT)
[1] 9
>
```

To find the range of MAT

```
> range (MAT)
[1] 1 9
>
```

To find the average of MAT

```
> mean(MAT)
[1] 5
>
```

To find square of MAT

```
> MAT^2
     [,1] [,2] [,3]
[1,]    1    4    9
[2,]   16   25   36
[3,]   49   64   81
>
```

One can multiply a matrix with a vector. For example, let us create a 4 by 4 matrix with first 16 integer values under the object S. Then create a vector of size 4 from 7 to 10. Multiply S and Y.

```
> S = matrix(1:16, ncol = 4)
> S
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
> Y = 7:10
> S*Y
     [,1] [,2] [,3] [,4]
[1,]    7   35   63   91
[2,]   16   48   80  112
[3,]   27   63   99  135
[4,]   40   80  120  160
> |
```

To perform a matrix multiplication in the mathematical sense, use the operator: **%*%**.
The dimensions of the two matrices must agree.

```
> A = matrix(seq(from = 5, to = 20, by = 2),nrow = 2, ncol = 4)
> A
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    7   11   15   19
> B = matrix(seq(from = 3, to = 17, by = 2), nrow = 4, ncol = 2)
> B
     [,1] [,2]
[1,]    3   11
[2,]    5   13
[3,]    7   15
[4,]    9   17
> A %*% B
     [,1] [,2]
[1,]  304  656
[2,]  352  768
> |
```

ACCESSING MATRIX ELEMENTS (INDEXING)

Indexing of matrix elements is the same as for data frames; the (*i*, *j*) element is located in the *i*th row and *j*th column. For example, the (2, 3) element of A is 15. We can access this element using

```
> A = matrix(seq(from = 5, to = 20, by = 2),nrow = 2, ncol = 4)
> A
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    7   11   15   19
> A[2,3]
[1] 15
> |
```

We can access the *i*th row using **A[i,]**, and the *j*th column using **A[, j]**. For example, for accessing 2nd row and 3rd column.

```
> A[2,]
[1]  7 11 15 19
>
> A[,3]
[1] 13 15
> |
```

When we do this, the result is usually a vector, with no dimension information kept. If one want to maintain the result as a row or column vector, we use the optional **drop = FALSE** argument when we index:

```
> A[2,,drop = FALSE]
     [,1] [,2] [,3] [,4]
[1,]    7   11   15   19
> A[,3, drop = FALSE]
     [,1]
[1,]   13
[2,]   15
> |
```

ROW AND COLUMN NAMES

One can give names for column and row. In the above example column and row does not have any names.

```
> A = matrix(seq(from = 5, to = 20, by = 2),nrow = 2, ncol = 4)
> A
     [,1] [,2] [,3] [,4]
[1,]    5    9   13   17
[2,]    7   11   15   19
> rownames(A)
NULL
> colnames(A)
NULL
> |
```

One can assign names using following functions

```
> rownames (A) = c("R1", "R2")
> A
   [,1] [,2] [,3] [,4]
R1    5    9   13   17
R2    7   11   15   19
> colnames (A) = c("C1", "C2", "C3", "C4")
> A
   C1 C2 C3 C4
R1  5  9 13 17
R2  7 11 15 19
> |
```

ARRAY

Arrays are generalizations of vectors and matrices. A vector is a one-dimensional array and a matrix is a two dimensional array. As with vectors and matrices, all the elements of an array must be of the same data type. One can create an array easily with the **array()** function, where one give the data as the first argument and a vector

with the sizes of the dimensions as the second argument. The number of dimension sizes in that argument gives you the number of dimensions. For example, an array with **four columns**, **two rows, and two "tables"** like this:

Syntax:

Z = array (data_vector, dim_vector)

```
> S = array(1:16, dim = c(4,2,2))
> S
, , 1

     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

, , 2

     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16

> |
```

By giving function **dim = c(4, 2, 2)**, it divides the single two dimensional 4 by 2 matrix into two tables. The first two arguments (4, 2) talks about the matrix and third arguments (2) divide into 2 tables as above.

All basic arithmetic operations which apply to matrices are also applicable to arrays and are performed on each element.

```
> Test = S + S^2
> Test
, , 1

     [,1] [,2]
[1,]    2   30
[2,]    6   42
[3,]   12   56
[4,]   20   72

, , 2

     [,1] [,2]
[1,]   90  182
[2,]  110  210
[3,]  132  240
[4,]  156  272

> |
```

DATA FRAMES

Data frames can also be regarded as an extension to matrices. Data frames can have columns of different data types and are the most convenient data structure for data analysis in R. In fact, most statistical modeling routines in R require a data frame as input.

One of the built-in data frames in R is `*mtcars*'.

```
> mtcars
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood 10.4  8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4 8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial 14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

The data frame contains information on different cars. Usually each row corresponds with a case and each column represents a variable. In this example the `carb' column is of data type `double' and represents the number of carburetors.

DATA FRAME ATTRIBUTES

A data frame can have the attributes names and row.names. The attribute names contain the column names of the data frame and the attribute row.names contains the row names of the data frame. The attributes of a data frame can be retrieved separately from the data frame with the functions names and row.names. The result is a character vector containing the names.

```
> rownames ( mtcars )[1:5]
[1] "Mazda RX4"          "Mazda RX4 Wag"      "Datsun 710"
[4] "Hornet 4 Drive"     "Hornet Sportabout"
>
> names (mtcars)
 [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
[11] "carb"
> |
```

CREATING DATA FRAMES

One can create data frames in several ways, by using the function **data.frame**. This function can be used to create new data frames or convert other objects into data frames. Few examples of the data Frame function are:

Example: 2.10

```
> my.data1 = sample(c(T,F), size = 5, replace = T)
> my.data1
[1]  TRUE  TRUE  TRUE FALSE  TRUE
> my.data2 = rnorm(5)
> my.data2
[1]  0.75157353 -0.10814813  0.02864433 -1.61957135 -1.24419896
> my.dF = data.frame(my.data1, my.data2)
> my.dF
  my.data1    my.data2
1     TRUE  0.75157353
2     TRUE -0.10814813
3     TRUE  0.02864433
4    FALSE -1.61957135
5     TRUE -1.24419896
> |
```

Example: 2.11

```
> test = matrix(runif(18),6,3) # creat a matrix with random elements
> test.df = data.frame(test)
> test.df
         X1         X2         X3
1 0.4814991 0.08952291 0.42438919
2 0.6389205 0.28422181 0.39558440
3 0.9005516 0.80632836 0.08159808
4 0.6085659 0.09627091 0.05374390
5 0.8808786 0.64853625 0.28995539
6 0.0854156 0.51274436 0.53556009
> names(test.df)
[1] "X1" "X2" "X3"
> |
```

R automatically creates column names: `X1', `X2' and `X3'. One can use the names function to change these column names.

```
> names(test.df) = c("Price", "Length", "Income") # Column names
> row.names(test.df) = c("RAJ", "MOHAN", "PAUL", "SAM", "RAVI", "SURESH")
> test.df
            Price     Length     Income
RAJ    0.4814991 0.08952291 0.42438919
MOHAN  0.6389205 0.28422181 0.39558440
PAUL   0.9005516 0.80632836 0.08159808
SAM    0.6085659 0.09627091 0.05374390
RAVI   0.8808786 0.64853625 0.28995539
SURESH 0.0854156 0.51274436 0.53556009
> |
```

Example: 2.12

```
> Age = c(25, 27, 28, 24, 33)
> Name = c("A", "B", "C", "D", "E")
> Income = c(25000, 20000, 24500, 17543, 30000)
> data.frame(Name, Age, Income)
  Name Age Income
1    A  25  25000
2    B  27  20000
3    C  28  24500
4    D  24  17543
5    E  33  30000
> |
```

Entering the data frame in spread sheet style

For entering the data frame in a spread sheet style data editor, the following command invokes the build-in editor with an empty spread sheet.

```
> Data_1 = edit(data.frame())
```

After entering the few points it looks likes below in fig 2.1.

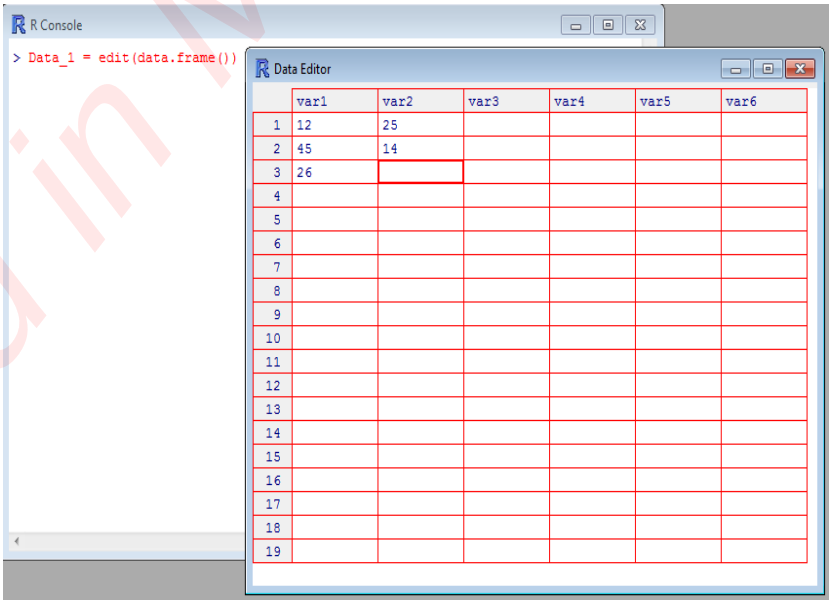


Fig. 2.1 Entering data frame in spread sheet style

One can change the variable names by clicking once on the cell containing it. This will display a dialog box where one can enter the variable name. In the below fig. 2.2 named the first column “marks”.

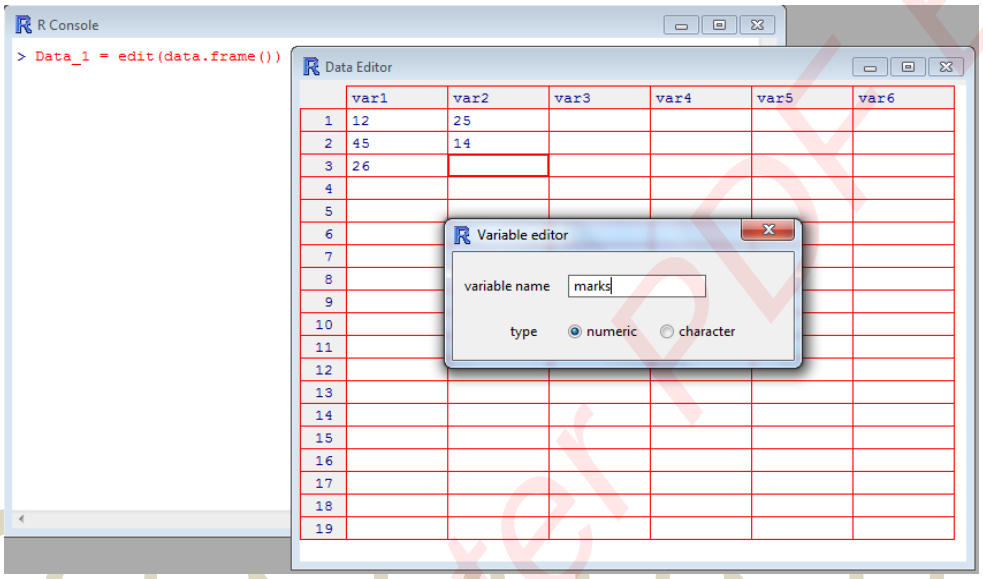


Fig. 2.2 Entering variable names

When finished, click the “X” in the upper right corner of the dialog box to return to the Data Editor window. The resulting data is now saved in a data frame called Data_1. Check the results by typing:

```
> Data_1
  marks_A marks_S Grade
1      12      25     A
2      45      14     B
3      26      25     A
4      56      55     B
5      58      58     A
6      25      32     B
>
```

We can also use the edit function to make changes in an already existing data frame. Suppose we want to edit a data frame entitled Dat. To do so, type:

```
> Data_2 = edit(Data_1)
```

An editable spread sheet appears and all changes will be saved in a new data frame entitled Data_2.

COMBINING DATA FRAMES

Use the function rbind to combine (or stack) two or more data frames. Consider the following two data frames `rand.df1' and `rand.df2'.

```
> Data1 = data.frame (
+ norm = rnorm(3), binom = rbinom(3,10,.01), chisq = rchisq (3,3)
+   )
> Data1
       norm binom    chisq
1  1.6219766     1 4.126759
2  0.7666341     0 4.973759
3 -0.8502004     0 1.728015
>
> Data2 = data.frame (
+ binom = rbinom (3,3,.02), chisq = rchisq(3,2)
+  )
> Data2
  binom     chisq
1     0 4.1759608
2     0 1.3336934
3     0 0.6160207
>
> data.com = rbind (
+ Data1[,c("binom","chisq")],
+             Data2[,c("binom","chisq")]
+ )
>
> data.com
  binom     chisq
1     1 4.1267594
2     0 4.9737590
3     0 1.7280151
4     0 4.1759608
5     0 1.3336934
6     0 0.6160207
>
```

The functions rbind expects that the two data frames have the same columns. The function **rbind.fill** in the `reshape' package can stack two or more data frames with any columns. It will fill a missing column with NA.

```
> library(plyr)
> library(reshape)
>
> Data1 = data.frame (
+ norm = rnorm(3), binom = rbinom(3,10,.01), chisq = rchisq (3,3))
> Data1
       norm binom    chisq
1 -1.4916699     0 3.124843
2  1.0702944     0 2.583311
3 -0.8684588     1 1.868600
>
> Data2 = data.frame (
+ binom = rbinom (3,3,.02), chisq = rchisq(3,2) )
> Data2
  binom    chisq
1     0 1.796967
2     0 1.639192
3     0 1.600860
>
> data.com = rbind.fill ( Data1,Data2 )
>
> data.com
       norm binom    chisq
1 -1.4916699     0 3.124843
2  1.0702944     0 2.583311
3 -0.8684588     1 1.868600
4         NA     0 1.796967
5         NA     0 1.639192
6         NA     0 1.600860
>
```

MERGING DATA FRAME

Two data frames can be merged into one data frame using the function merge. If the original data frames contain identical columns, these columns only appear once in the merged data frame. Consider the following two data frames:

```
> Name = c("RAJ", "RAMA", "SAM", "SASI")
> Year = c(1992,1979,1980,1974)
> Interest = c(0.12, 0.15, 0.89, 0.79)
>
> D1 = data.frame(Name,Year,Interest)
> D1
  Name Year Interest
1  RAJ 1992     0.12
2 RAMA 1979     0.15
3  SAM 1980     0.89
4 SASI 1974     0.79
> Name = c("RAJ", "RAMA", "SAM", "SASI")
> Year = c(1992,1979,1980,1974)
> Age = c(45, 58, 57, 96)
> Income = c(2580, 2457, 5689, 2257)
>
> D2 = data.frame(Name,Year, Age, Income)
> D2
  Name Year Age Income
1  RAJ 1992  45   2580
2 RAMA 1979  58   2457
3  SAM 1980  57   5689
4 SASI 1974  96   2257
> mergeD12 = merge(D1,D2)
> mergeD12
  Name Year Interest Age Income
1  RAJ 1992     0.12  45   2580
2 RAMA 1979     0.15  58   2457
3  SAM 1980     0.89  57   5689
4 SASI 1974     0.79  96   2257
>
```

STACKING COLUMNS OF DATA FRAMES

The function **stack** can be used to stack columns of a data frame into one column and one grouping column. Consider the following example:

```
> G1 = rnorm(3)
> G2 = runif(3)
> G3 = rbinom (3,2,0.8)
>
> DF = data.frame(G1, G2, G3)
> stack(DF)
       values ind
1 -0.08478802  G1
2  1.23676684  G1
3 -0.19602415  G1
4  0.16853181  G2
5  0.88269341  G2
6  0.63583203  G2
7  2.00000000  G3
8  1.00000000  G3
9  1.00000000  G3
>
```

So by default all the columns of a data frame are stacked. Use the **select** argument to stack only certain columns.

```
> stack(DF,select=c(G1,G3))
      values ind
1 -0.08478802  G1
2  1.23676684  G1
3 -0.19602415  G1
4  2.00000000  G3
5  1.00000000  G3
6  1.00000000  G3
>
```

TABLES

Another common way to store information is in a table. First let us see how to create one way table. One way to create a table is using the **table** command. The argument it takes is a vector of factors, and it calculates the frequency that each factor occurs. Here is an example of how to create a one way table:

Example: 2.13

```
> a = factor(c("A","A","B","A","B","B","C","C","A"))
> T = table(a)
> T
a
A B C
4 3 2
>
```

TWO WAY TABLES

Example: 2.14

```
> smoke <- matrix(c(51,43,22,92,28,21,68,22,9),ncol=3,byrow=TRUE)
> colnames(smoke) <- c("High","Low","Middle")
> rownames(smoke) <- c("current","former","never")
> smoke <- as.table(smoke)
> smoke
        High Low Middle
current   51  43     22
former    92  28     21
never     68  22      9
>
```

Example: 2.15 In the example below we have two questions. In the first question the responses are labeled “Never,” “Sometimes,” or “Always.” In the second question the responses are labeled “Yes,” “No,” or “Maybe.” The set of vectors “a” and “b” contain the response for each measurement. The third item in “a” is how the third person responded to the first question and the third item in “b” is how the third person responded to the second question.

```
> a <- c("Sometimes","Sometimes","Never","Always","Always","Sometimes")
> b <- c("Maybe","Maybe","Yes","Maybe","Maybe","No","Yes","No")
> results <- table(a,b)
Error in table(a, b) : all arguments must have the same length
> results
           b
a           Maybe No Yes
  Always        2  0   0
  Never         0  1   1
  Sometimes     2  1   1
> |
```

Example: 2.16 Just as in the case with one-way tables it is possible to manually enter two way tables. The procedure is exactly the same as above except that we now have more than one row. We give a brief example below to demonstrate how to enter a two-way table that includes breakdown of a group of people by both their gender and whether or not they smoke. You enter all of the data as one long list but tell R to break it up into some number of columns:

```
> sexsmoke<-matrix(c(70,120,65,140),ncol=2,byrow=TRUE)
> rownames(sexsmoke)<-c("male","female")
> colnames(sexsmoke)<-c("smoke","nosmoke")
> sexsmoke <- as.table(sexsmoke)
> sexsmoke
       smoke nosmoke
male      70     120
female    65     140
> |
```

LISTS

A list is like a vector. However, an element of a list can be an object of any type and structure. Consequently, a list can contain another list and therefore it can be used to construct arbitrary data structures. Lists are often used for output of statistical routines in R. The output object is often a collection of parameter estimates, residuals, predicted values etc.

CREATING LISTS

A list can also be constructed by using the function list. The names of the list components and the contents of list components can be specified as arguments of the list function by using the = character.

```
> x1 <- 1:5
> x2 <- c(T,T,F,F,T)
> y <- list(numbers=x1, wrong=x2)
> y
$numbers
[1] 1 2 3 4 5

$wrong
[1]  TRUE  TRUE FALSE FALSE  TRUE

> |
```

So the left-hand side of the = operator in the list function is the name of the component and the right-hand side is an R object. The order of the arguments in the list function determines the order in the list that is created. In the above example the logical object `wrong' is the second component of y.

```
> y[[2]]
[1]  TRUE  TRUE FALSE FALSE  TRUE
> |
```

The function names can be used to extract the names of the list components. It is also used to change the names of list components.

```
> names(y)
[1] "numbers" "wrong"
> names(y) <- c("lots", "valid")
> names(y)
[1] "lots"  "valid"
> |
```

READING AND WRITING DATA FROM EXTERNAL FILES

Often it is need to access data sets that are saved in a computer file. The easiest way to import and export data into R is using text files. Begin by saving the data in plain text format. The command **read.table( )** reads in an external text file and creates a data frame.

For example, suppose we have saved the file called "**ACC.txt**" on drive F which reads:

Bank	Acc_number	Amount
AAA	33568	58000
AAB	35698	24850
BXA	25871	35698

To read this data **into R we use the following command**

```
> data1<-read.table("F:/ACC.txt", header=TRUE)
> |
```

The option **header=TRUE** tells R that the first row consists of variable names rather than actual data.
The command **write.table( )** outputs the specified data frame to a file. A blank space is used to separate columns when **sep=" "** is specified within its argument. Other popular
choices include comma **(sep=",")**, and **tab (sep="\t")**.

The command

```
> write.table(data1,"ACC_1.txt",sep = "")
```

writes the data contained in the data frame "ACC" into a file called "ACC_1.txt". It uses spaces to separate the variables from one another.

EXERCISE –UNIT 2

1. Classify the following data as categorical or numerical. If numerical, classify into ordinal, discrete or continuous.

a. Number of trees in an area. b. Sex of trapped animal

c. Carbon monoxide emission per day by car d. order of a child in a family.

2. Create a vector of factors with the following levels: Low, Medium, and High.
3. Create the same vector, but with the levels ordered.
4. In the following, show how you verify that your answer is correct.
   (a) Create a vector of double values 1, 2, 3. (b) Create the same vector, but now with integer values.
5. Prove that x produced with x <- 1:10 is not an array.
6. Turn x into an array.
7. Is x produced with matrix(0, ncol=3, nrow=2) an array? Is it a matrix? Prove it!
8. You are given a list of 10 "names" and 10 test scores:
   Names = c(A, B, C, D, E, F, G, H, I, J)
   Scores = c(59, 34, 56, 47, 65, 72, 79, 90, 83, 51)
   Show the code and the result for the following:
   (a) Make a data frame with the first column named "score" and the second named "names."
   (b) Make a list with the first element named "names" and the second named "score."
9. Use scan( ) to create a vector of 10 integers
10. Import data directly to R from an Excel file of your choice.
11. Pick 10 people at random (5 males and 5 females) and create a data frame with the following columns: Gender—a factor with two levels, M and F, Height— a numeric variable holding the height of each person (in cm), First Name—a character variable and Last Name—a character variable.
12. Give the command that creates the sequence 0, 2, 4, . . . , 20.
13. Create a sequence that includes the first 5 and last 5 of the English lower case letters. Use the symbol :, c() and length() in a single statement to create this sequence.
14. Answer the following briefly:
    (a) What prompt do you get following the statement seq(1 : 10, by? Why?
    (b) How would you restart typing the statement above by getting out of the continuation prompt?
15. Will the following addition work? Why?
    X = c(1, '2', 3); Y = 5
    X+Y
16. Let x <- 1 : 7 and y <- 1 : 2. What is the length of the vector x + y? What are the values of its elements? Why?
17. Discuss the difference between the functions is.na( ) and is.nan( ).
18. Let x <- 1 : 6 and y <- 1 : 3. What is the length of the vector x * y? What are the values of its elements? Why?
19. Will the assignment x <- c(4 < 5, 'a' < 'b') work? What do you get?
20. What will be the modes of x under the following assignments? Explain.
    (a) X = c(TRUE, 'a') (b) X = c(TRUE, 1) (c X = c('a', 1)

UNIT 3: GRAPHICS IN R

Graphics is an important and extremely versatile component of the R environment. It is possible to use the facilities to display a wide variety of statistical graphs and also to build entirely new types of graph. R plotting commands can be used to produce a variety of graphical displays and to create entirely new kinds of display.

Plotting commands are divided into three basic groups:
(a) High-level plotting functions create a new plot on the graphics device, possibly with axes, labels, titles and so on. (b)Low-level plotting functions adds more information to an existing plot, such as extra points, lines and labels. (c)Interactive graphics functions allow us interactively add information to, or extract information from, an existing plot, using a pointing device such as a mouse.

In addition, R maintains a list of graphical parameters which can be manipulated to customize your plots. This manual describes what are known as 'base' graphics. A separate graphics sub-system in package grid co-exists with the base–it is more powerful but harder to use.

SAVING GRAPHS

You can save the graph using a variety of formats from the given menu
File -> Save As.
You can also save the graph via code using one of the following functions.

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

GRAPHICAL PARAMETERS

You can specify fonts, colors, line styles, axes, reference lines, etc. by specifying graphical parameters. This allows a wide degree of customization. Graphical parameters, are covered in the Advanced Graphs section. The Advanced Graphs section also includes a more detailed coverage of axis and text customization.

ARGUMENTS TO HIGH-LEVEL PLOTTING FUNCTIONS

There are a number of arguments which can be passed to high-level graphics functions, as follows:

add=TRUE Forces the function to act as a low-level graphics function, superimposing the plot on the current plot (some functions only).

axes=FALSE Suppresses generation of axes—useful for adding your own custom axes with the axis() function. The default, axes=TRUE, means include axes.

log="x"
log="y"
log="xy" Causes the x, y or both axes to be logarithmic. This will work for many, but not for all, types of plot.

type = The type= argument controls the type of plot produced, as follows:

type="p" Plot individual points (the default)

type="l" Plot lines

type="b" Plot points connected by lines (both)

type="o" Plot points overlaid by lines

type="h" Plot vertical lines from points to the zero axis (high-density)

type="s"
type="S" Step-function plots. In the first form, the top of the vertical defines the point; in the second, the bottom.

type="n" No plotting at all. However axes are still drawn (by default) and the coordinate system is set up according to the data. Ideal for creating plots with subsequent low-level graphics functions.

xlab=string ylab=string

Axis labels for the x and y axes. Use these arguments to change the default labels, usually the names of the objects used in the call to the high-level plotting function.

main=string
Figure title, placed at the top of the plot in a large font.

sub=string
Sub-title, placed just below the x-axis in a smaller font.

LOW-LEVEL PLOTTING COMMANDS

Sometimes the high-level plotting functions don't produce exactly the kind of plot you desire. In this case, low-level plotting commands can be used to add extra information (such as points, lines or text) to the current plot. Some of the more useful low-level plotting functions are:

points(x, y) lines(x, y)
Adds points or connected lines to the current plot. plot( ) type= argument can also be passed to these functions (and defaults to "p" for points() and "l" for lines().)

text(x, y, labels, ...)
Add text to a plot at points given by x, y. Normally labels is an integer or character vector in which case labels[i] is plotted at point (x[i], y[i]). The default is 1:length(x).

Note: This function is often used in the sequence

> plot(x, y, type="n"); text(x, y, names)

The graphics parameter type="n" suppresses the points but sets up the axes, and the text() function supplies special characters, as specified by the character vector names for the points.

abline(a, b) abline(h=y) abline(v=x) abline(lm.obj)
Adds a line of slope **b** and intercept **a** to the current plot. h=y may be used to specify y-coordinates for the heights of horizontal lines to go across a plot, and v=x similarly for the x-coordinates for vertical lines. Also lm.obj may be list with a coefficients component of length 2 (such as the result of model-fitting functions) which are taken as an intercept and slope, in that order.

polygon(x, y, ...)
Draws a polygon defined by the ordered vertices in (x, y) and (optionally) shade it with hatch lines, or fill it if the graphics device allows the filling of figures.

legend(x, y, legend, ...)
Adds a legend to the current plot at the specified position. Plotting characters, line styles, colors etc., are identified with the labels in the character vector legend. At least one other argument v (a vector the same length as legend) with the corresponding values of the plotting unit must also be given, as follows:

legend(, fill=v)
Colors for filled boxes

legend(, col=v)
Colors in which points or lines will be drawn

legend(, lty=v)
Line styles

legend(, lwd=v)
Line widths

legend(, pch=v)
Plotting characters (character vector)

title(main, sub)
Adds a title main to the top of the current plot in a large font and (optionally) a sub-title sub at the bottom in a smaller font.

axis(side, ...)

Adds an axis to the current plot on the side given by the first argument (1 to 4, counting clockwise from the bottom). Other arguments control the positioning of the axis within or beside the plot, and tick positions and labels. Useful for adding custom axes after calling plot() with the axes=FALSE argument.

Low-level plotting functions usually require some positioning information (e.g., x and y co-ordinates) to determine where to place the new plot elements. Coordinates are given in terms of user coordinates which are defined by the previous high-level graphics command and are chosen based on the supplied data.

Where x and y arguments are required, it is also sufficient to supply a single argument being a list with elements named x and y. Similarly a matrix with two columns is also valid input. In this way functions such as locator() (see below) may be used to specify positions on a plot interactively.

BAR GRAPHS

Bar chart is constructed to show frequencies of different categories of a categorical variable in the given data. The horizontal axis represents the different categories and the vertical axis is used to show the frequency. Bar Graphs can be used to depict specific information like mean, median and cumulative frequency. Bar charts allow you to make selections that determine the type of chart you obtain.

The basic function is: barplot(data)

Example 3.1:

Particulars	Expressed in % for the year		
	'1979	'1980	'1981
Raw Material	60	65	60
Labor	15	17.5	18
Direct Expenses	10	5	8
Factory Expenses	10	7.5	8
Office Expenses	5	5	6
Total Cost	100	100	100

Represent the above data for 1979 by a simple bar diagram
Represent the above data by means of staked (a.k.a. sub-divided) bar diagrams:

Solution:

```
>year_1979 = c(60,15,10,10,5)

>barplot(year_1979,main="Expenses",xlab="year1979",ylab="Expenses +%",beside=TRUE,col=rainbow(5))
```

Explanation:
Line 1
year_1979 <- c(60,15,10,10,5)
The above command constructs a vector year_1979 for the expenses for the year 1979 expressed as % for the year 1979.

Line 2
Graph with adjacent bars using rainbow colors
The above is a comment line.

Line 3 - 4
barplot(year_1979,main="Expenses",xlab="year 1979",ylab="Expenses %",beside=TRUE, col=rainbow(5))

The command **barplot** creates a plot with vertical or horizontal bars.
The **title, x-axis label** and **y-axis** labels are set. The option **beside** is a logical value .If FALSE, the columns of height are portrayed as stacked bars, and if TRUE the columns are portrayed as juxtaposed bars.

The option **horiz** has default value FALSE. If TRUE, the columns are drawn horizontally with first at the bottom; Otherwise, the bars are drawn vertically with the first bar to the left.

The option **col** is a vector of colors for the bars or bar components. By default, grey is used if height is a vector, and a gamma-corrected grey palette if height is a matrix.The rainbow(5) creates a vector of 5 contiguous colors.

Line 5
Place the legend at the top-left corner with no frame using the rainbow colors
The above line is a comment line.

Line 6 – 8
legend("topright",c("Raw Material","Labor","Direct Expenses","Factory Expenses","Office Expenses"),cex=0.8,bty="n",fill=rainbow(5))

The command **legend** adds legend at the specified location ("topright"). The height of the legend are magnified to the specified limit of 0.8. The option bty specifies the type of box to be drawn around the legend. The allowed values are "o" (the default) and "n". For value "o", the resulting box resembles the corresponding upper case letter. A value of "n" suppresses the box. The option fill causes the boxes filled with the specified colors to appear beside the legend text.

HISTOGRAMS

The barplot function can be used to create a frequency plot of various sorts but it does not produce a continuous distribution along the x-axis. A true frequency distribution should have the bar categories (i.e. the x-axis) as continuous items. The frequency plot produced previously has discontinuous categories. To create a frequency distribution chart we need a histogram, which has a continuous range along the x-axis.

The command in R is:
hist(variable)

To plot the probabilities (i.e. proportions) rather than the actual frequency we need to add the command
prob= TRUE
The command is
hist(variable, prob= TRUE)

We can change axis labels and the main title using the same commands as for the barplot function. By default R works out where to insert the breaks between the bars. You can change the number of breaks by adding a simple command e.g.

hist(variable,breaks=10)

You can manipulate the axes by changing the limits e.g. make the x-axis start at zero and run to 6 and y-axis start at 0 and run to 10 by another simple command

hist(test.data, 10, xlim=c(0,6), ylim=c(0,10))

Example 3.2 For the given 16 observations use the scan function to input the data and draw a histogram.

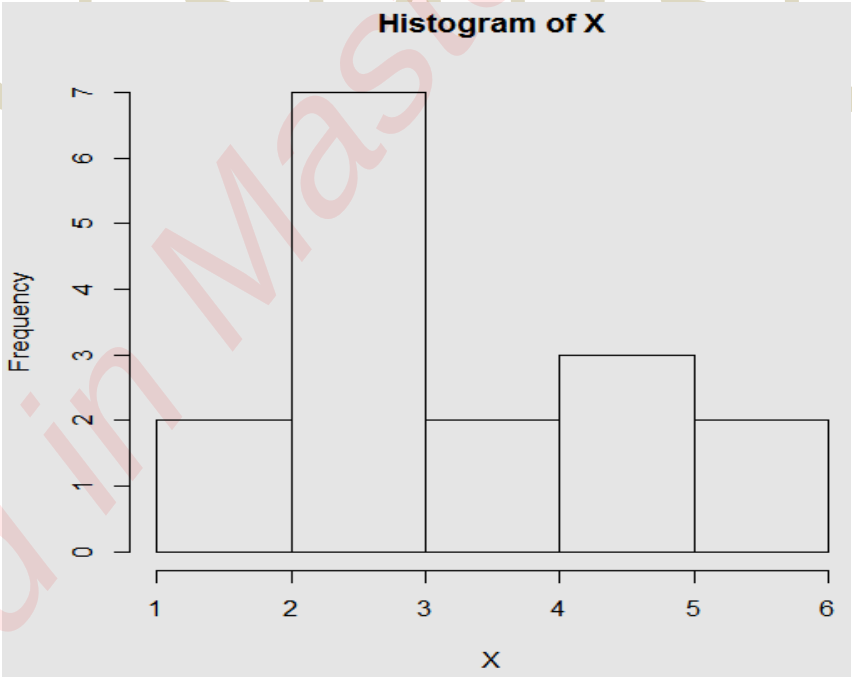
2.1 2.6 2.7 3.2 4.1 4.3 5.2 5.1 4.8 1.8 1.4 2.5 2.7 3.1 2.6 2.8

Solution:
x = scan()
1: 2.1
2: 2.6
3: 2.7
4: 3.2
5: 4.1
6: 4.3

7: 5.2
8: 5.1
9: 4.8
10: 1.8
11: 1.4
12: 2.5
13: 2.7
14: 3.1
15: 2.6
16: 2.8
17:
> hist(x)

Explanation:
Line1
The scan() reads the 16 values and stores it in a variable x

Line2
hist(x)
Displays a histogram.



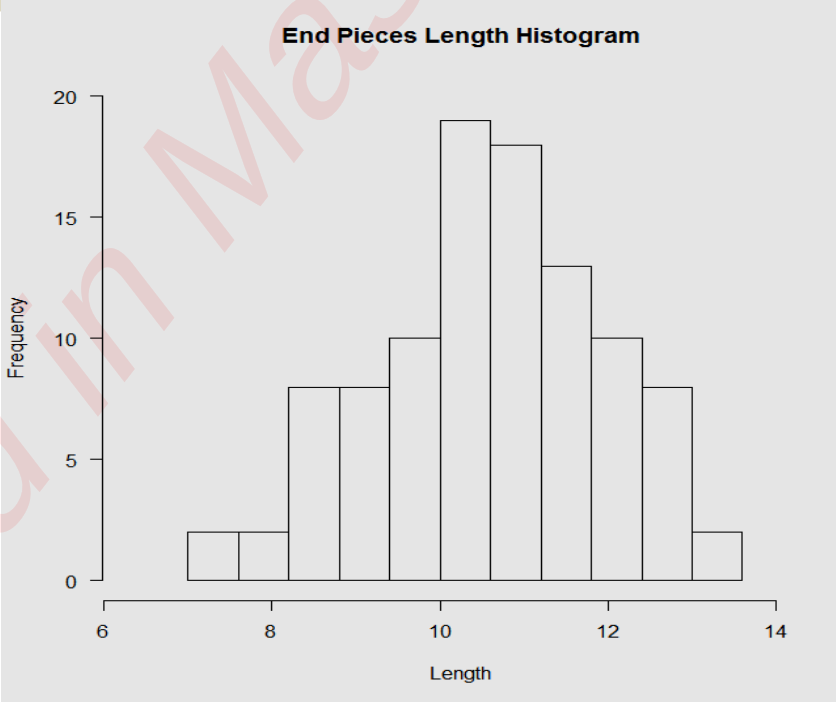
Example 3.3 Data on lengths in centimeters (cm) of 100 end pieces in an engineering plant are specified in the file **end_pieces.dat**. (Use the enclosed file).Create a frequency table and also draw a histogram for the above data.

Solution:
>length_data<read.table("C:/Users/PVS/Documents/R/end_pieces.dat",header=T, sep="\t")

```
>eld<-c(length_data$Batch.1,length_data$Batch.2,length_data$Batch3,
+length_data$batch.4,length_data$Batch.5,length_data$Batch.6,
+length_data$Batch7,length_data$batch.8,length_data$Batch.9,
+length_data$Batch.10)
>boundaries=seq(7.0,13.6,by=0.6)
>x<-sort(eld)
>#factorx=factor(cut[x,breaks=nclass.sturges(x),include.lowest=T,right=F])
>factorx=factor cut[x,breaks=nclass.sturges(x),include.lowest=T,right=F])
>as.matrix(table(factorx))
```

```
             [,1]
[7,7.6)         2
[7.6,8.2)       2
[8.2,8.8)       8
[8.8,9.4)       8
[9.4,10)       10
[10,10.6)      19
[10.6,11.2)    18
[11.2,11.8)    13
[11.8,12.4)    10
[12.4,13)       8
[13,13.6]       2
```

```
>hist(x,braeks=boundaries,include.lowest=T,right=F,main="End Pieces Length
                Histogram",xlab="length",xlim=c(min(x)-1,max(x)+1),ylim=c(0,20),las=1)
```



Explanation
Set the directory to the location where the data file end_pieces.dat is stored.
You can view the data by issuing the command file.show("end_pieces.dat")

```
>setwd("c:/users/PVS/Documents/R")
>file.show("end_pieces.dat")
```

Batch 1	Batch 2	Batch 3	Batch 4	Batch 5	Batch 6	Batch 7	Batch 8	Batch 9	Batch 10
9.7	10.0	8.7	9.1	10.3	10.1	10.1	7.3	10.7	8.4
9.9	11.0	11.3	11.4	11.7	10.5	10.2	11.3	10.3	10.6
10.2	12.3	12.0	12.6	12.4	9.8	8.7	8.3	9.0	11.8
10.0	7.9	10.7	11.2	11.2	10.8	10.9	12.8	9.2	8.7
9.0	9.8	11.0	12.3	10.7	10.9	7.3	12.1	8.7	9.2
10.7	10.7	9.9	11.7	12.3	12.7	11.0	9.8	10.3	10.0
10.5	9.6	10.0	12.8	9.3	8.7	10.4	10.3	11.3	9.2
12.8	12.7	11.3	11.3	11.0	12.3	10.8	12.9	13.0	7.8
9.9	11.3	12.3	10.7	12.1	13.2	10.3	10.4	9.8	10.7
10.0	11.0	11.7	9.3	8.7	12.2	11.3	10.7	10.5	9.8

Line 1
This command **read.table** reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file. The data contains the heading and each piece of data is separated by a tab delimiter. Output of this command is assigned to the variable **length_data.**

Lines 2 – 4
The command **c** concatenates various columns in the variable **length_data** into a vector eld.

Line 5

Create a vector, **boundaries** giving the breakpoints between histogram cells
i.e.(7.0,7.6,8.2,8.8,9.4,10.0,10.6,11.2,11.8,12.6,13.2,13.8).

Line 6
Sort **eld** into ascending order and store the result in x.

Line 7
Now create a factor, **factorx** for the frequency table by giving the breakpoints between histogram cells creating a frequency table by using the option cut(x, breaks=breakpoints) where x is a numeric vector and breaks is the break points to divide x into different ranges based on the break points.including the x[i] in the first bar, if x[i] = breaks value if include.lowest= TRUE, an x[i] equal to the breaks value will be included in the first (or last, for right = FALSE) bar.creating right open intervals If right = TRUE, the histograms cells are right-closed (left open) intervals.

Line 8
Convert the result into a matrix table which displays the frequency table.

```
             [,1]
[7,7.6)         2
[7.6,8.2)       2
[8.2,8.8)       8
[8.8,9.4)       8
[9.4,10)       10
[10,10.6)      19
[10.6,11.2)    18
[11.2,11.8)    13
[11.8,12.4)    10
[12.4,13)       8
[13,13.6]       2
```

Line 9

Create a histogram for x with break calculated as in line 5; including the x[i] in the first bar, if x[i] = breaks valuedisable right-closing of cell interval;set heading and x-axis label; make x-axis range from minimum of x to maximum of x; make y-axis range from 0 to 20;

Example 3.4

Construct a histogram for the following frequency distribution.

Weights in Kg.	40.5 – 45.5	45.5 – 50.5	50.5 – 55.5	55.5 – 60.5	60.5 – 65.5	65.5 – 70.5	70.5 – 75.5	75.5 – 80.5
Number of men	4	5	9	6	11	5	7	3

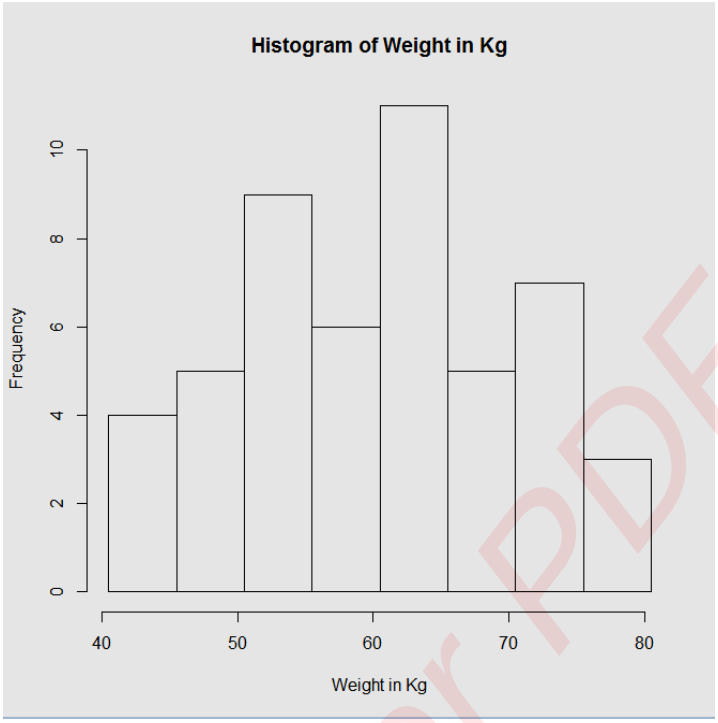
Solution

>myhist = list(braeks = seq(40.5,80.5,by=5),counts = c(4,,9,6,11,5,7,3),

>class(myhist) = “histogram”

>plot(myhist)

+density=c(4,5,9,6,11,5,7,3)/50,xname=”weight in Kg”)



Explanation
Line 1 -2
> myhist<-list(braeks=seq(40.5,80.5,by=5),counts=c(4,,9,6,11,5,7,3),
+density=c(4,5,9,6,11,5,7,3)/50,xname=”weight in Kg”)

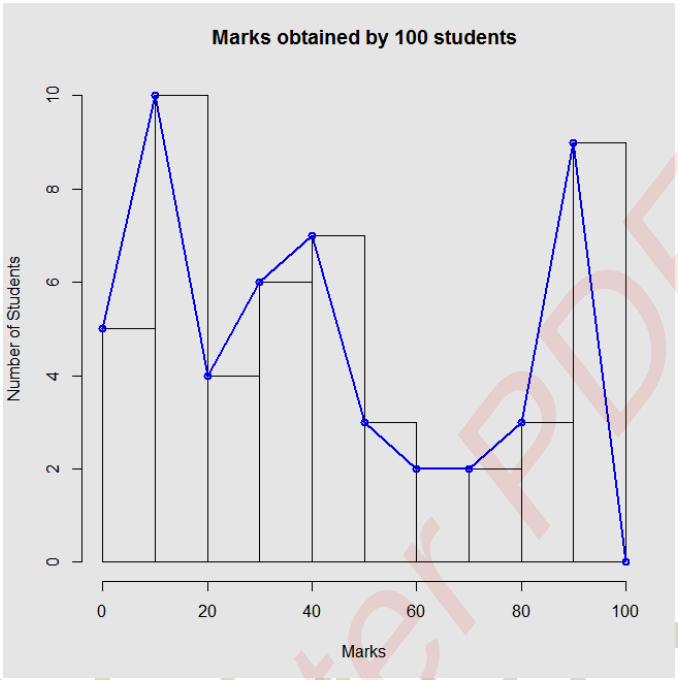
The function seq(40.5,80.5,by=5) creates the sequence 40.5,45.5,..80.5.Construct a list myhist by coercing the boundaries, counts = number of men in each class interval, density = relative frequency being count divided by 50, x-axis name “Weight in Kg.”

Line 3 – 4
>class(myhist)<-“histogram”
>plot(myhist)
Make the class of the object my hist as histogram. Plots the histogram.
Example 3.5
Consider the marks out of 100, obtained by the students in a class in a test. Construct a histogram and frequency polygon for the following data.

8, 8, 6, 6, 11, 11, 12, 13, 14, 15, 16, 17, 19, 21, 22, 23, 24, 31, 32, 35, 38, 39, 39, 41,42,
42, 43, 44, 45, 46, 55, 56, 67, 69, 71, 75, 81, 84, 86, 91, 92, 93, 94, 94, 94, 95, 96, 98

Solution:
> marks = c(8,8,6,6,11,11,12,13,14,15,16,17,19,21,22,23,24,31,32,35,
+38,39,39,41,42,42,43,44,45,46,55,56,67,69,71,75,81,84,86,91,92,93,
+94,94,94,95,96,98)
>boundaries <- seq(10,100,by=10)

>hist(marks, breaks = boundaries, main = "Marks obtained by 100 +students", xlab = "Marks", ylab="Number of Students")



Explanation
Line 1 – 3
marks = c(8,8,6,6,11,11,12,13,14,15,16,17,19,21,22,23,24,31,32,35,38,39,39,41,42,42,43,44,45,46,55,56, 67,69,71, 75,81, 84,86,91,92,93,94,94,94,95,96,98)
Construct a vector **marks** for the given data

Line 4
boundaries <- seq(10,100,by=10)
The option **seq**(10,100,by=10) creates the sequence 10,20,30,..100

Line 5
hist(marks, breaks = boundaries, main = "Marks obtained by 100 students", xlab = "Marks", ylab="Number of Students")

The command **hist** is the basic command to construct a histogram.You can use the **breaks()** option to create the number of bins from the numeric vector, marks and give it a vector of breakpoints, boundaries.Use the option **main**, **xlab** and **ylab** to set the title for the graph, x-axis and y-axis labels.

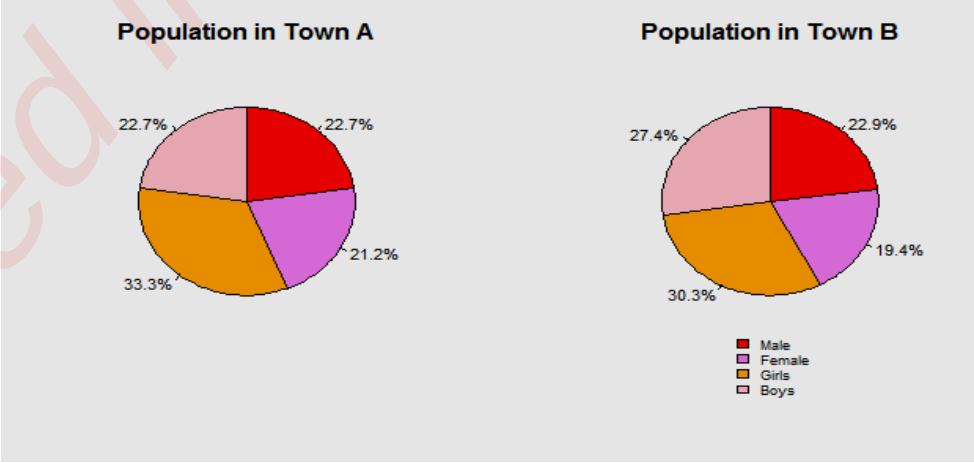
PIE CHART
A pie chart displays the contribution of parts to a whole.Each slice of a pie chart corresponds to a group defined by a single grouping variable.This provides a lucid visual summary of a dummy variable or a categorical variable with several categories. Pie charts are only used when the values of a variable taken are limited.
To produce a simple pie chart we type the following:
pie(pie.data)

Example 3.6 Draw a Pie diagram to represent the following population in a town:

Category	Town A	Town B
Male	3000	4000
Female	2800	3400
Girls	4400	5300
Boys	3000	4800
Total	13200	17500

Solution:

```
#  Define population vector with 4 values
#
townA <- c(3000,2800,4400,3000)
townB<-c(4000,3400,5300,4800)
# Calculate the percentage for each category, rounded to one decimal place
townA_labels <- round(townA/sum(townA) *100,1)
townB_labels <- round(townB/sum(townB) *100,1)
#
# Concatenate a '%' char after each value
townA_labels <- paste(townA_labels, "%", sep="")
townB_labels <- paste(townB_labels, "%", sep="")
#
# Create a pie chart with defined heading and custom colors and labels
par(mfrow = c(2,2),xpd=TRUE)
pie(townA, main = "Population in Town A", col = colors, labels = townA_labels,cex = 0.8,clockwise=TRUE)
pie(townB, main = "Population in Town B", col = colors, labels = townB_labels,cex = 0.8,clockwise=TRUE)
# Create a legend at the bottom
legend("bottom",inset=c(0,-0.3),c("Male","Female","Girls","Boys"),cex=0.7, fill=colors,bty="n")
par(mfrow = c(1,1))
```



Explanation
Line 1 – 4

```
#  Define population vector with 4 values
#
townA <- c(3000,2800,4400,3000)
townB<-c(4000,3400,5300,4800)
```

First two lines are comment lines.Third line constructs vector townA by concatenating the population values for various categories in town A.Fourth line constructs vector townB.

Line 5 –12

```
# Calculate the percentage for each category, rounded to one decimal place
townA_labels <- round(townA/sum(townA) *100,1)
townB_labels <- round(townB/sum(townB) *100,1)
#
# Concatenate a '%' char after each value
townA_labels <- paste(townA_labels, "%", sep="")
townB_labels <- paste(townB_labels, "%", sep="")
```

Line 5 is a comment line.

Line 6 creates the vector townA_labels, the percentage for each category of population in townA

Line 7 creates the vector townB_lables, the percentage for each category of population in townB

Line 8 and 9 are comment lines.

Line 10 concatenates vector townA_lables with “%” symbol and this will be used in the diagram

Similarly Line 11 concatenates vector townB_labels with “%” symbol.
Line 12 is a comment line

Line 13 –16

```
# Create a pie chart with defined heading and custom colors and labels
par(mfrow = c(2,2),xpd=TRUE)
pie(townA, main = "Population in Town A", col = colors, labels = townA_labels,cex = 0.8,clockwise=TRUE)
pie(townB, main = "Population in Town B", col = colors, labels = townB_labels,cex = 0.8,clockwise=TRUE)
```

Line 13 is a comment line.

Line 14: The option **par** lets you set graphical parameters. The sub-option **mfrow** sets the number of rows and columns, the plotting device will be divided in.Here, a vector of c(2,2) is given as the value to the **mfrow** subcommand. Subsequently, four plots are made as usual. Finally, you must set the graphical parameter to normal by entering vector c(1,1) to the **mfrow** subcommand.

Set **xpd** as TRUE. If TRUE, all plotting is clipped to the figure region, if FALSE, all plotting is clipped to the plot region; if NA all plotting is clipped to the device region.

Line 15 draws a pie diagram with the population values for townA with the defined heading and labels created in the above code

Option **col** specifies a vector of colors to be used in filling or shading the slices. If missing, a set of 6 pastel colors is used.

Option **cex** specifies the numerical value giving the amount by which plotting text and symbols should be magnified relative to the default.

Option **clockwise:** If TRUE, the slices are drawn clockwise, otherwise it is drawn anti-clockwise (default)

Line 16 draws a pie diagram with population values for townB.

Line 17 – 21

```
# Create a legend at the bottom
legend("bottom",inset=c(0,-0.3),c("Male","Female","Girls","Boys"),cex=0.7, fill=colors,bty="n")
par(mfrow = c(1,1))
```

Line 17 and 18 are comment lines.

Line 19 - 20: The command **legend** is used to add legends to the plot.
Location of the legend can be specified by x and y coordinates. Also, we can specify a single keyword from the list ("bottomright" ,"bottom", "left", "topleft", "topright", "right", "center")

inset distance(s) from the margins as a fraction of the plot region when legend is placed by a key word.

labels specifies a character or expression vector of length > 1 to appear in the legend.

fill will cause the boxes filled with the specified colors to appear beside the legend text

bty specifies the type of box to be drawn around the legend. The allowed values are "o" and "n". Default is "o"
Line 21 sets the graphical parameter to normal by specifying a vector c(1,1) to the **mfrow** subcommand.

FREQUENCY POLYGON

Frequency polygons are a graphical device for understanding the shapes of the distributions. They serve the same purpose as histograms, but are especially helpful for comparing sets of data. Frequency polygons are also a good choice for displaying cumulative frequency distributions. In a Frequency Polygon, a line graph is drawn by joining all the midpoints of the top of the bars of a histogram.

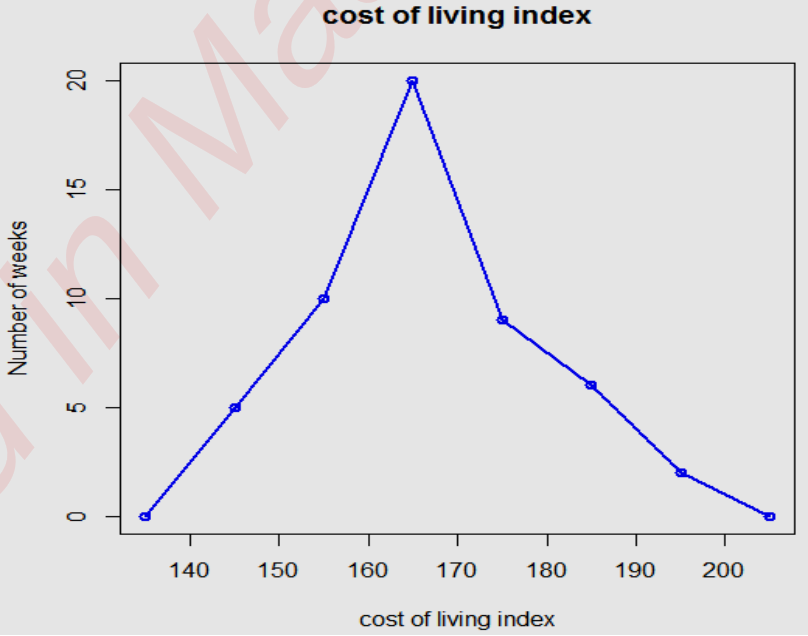
Example 3.7 Construct a frequency polygon for the following data:

In a city, the weekly observations made in a study on the cost of living index are given in the following table.

Cost of living index	Number of weeks
140-150	5
150-160	10
160-170	20
170-180	9
180-190	6
190-200	2
Total	52

Solution

```
myfrq<-list(breaks=seq(135,205,by=10),counts=c(0,5,10,20,9,6,2,0))

plot(as.vector(myfrq$breaks),as.vector(myfrq$counts),main="cost of living index" , xlab="cost of living index",ylab="Number of weeks")

text(x=seq(15,205,by=10),y=c(0,5,10,20,9,6,2,0),labels=c("A","B","C","D","E", "F","G","H"),pos=4,col="red")

lines(as.vector(myfrq$breaks),as.vector(myfrq$counts),lwd=2,col="blue",type="o")
```



Explanation

We now draw a frequency polygon by plotting the cost-of-living index along with the horizontal axis, the frequencies along the vertical axis, and then plotting and joining the points B(145,5), C(155,10),D(165,20),E(175,9),F(185,6) and G(195,2) by line segments.

We should not forget to plot the point corresponding to the cost-of-living index in the class 130-140 (just before the lowest class 140-150) with zero frequency, that is A(135,0), and the point H(205,0) occurs immediately after G(195,2). So the resultant frequency polygon will be ABCDEFGH.

Line 1
>myfrq<-list(breaks=se(135,205,by=10),counts=c(0,5,10,20,9,6,2,0))

The option **seq**(145,195,by=10) creates the sequence 135,155,165,175,..195,205
Construct a list myfrq by coercing the breaks and counts(containing the cost of living in each class interval - each week)

Line 2- 3
>plot(As.vector(myfrq$breaks),as.vector(myfrq$counts),main="cost of living index ",xlab="cost of living index",y lab="Number of weeks")

The command **plot** the points in vector, as.vector(myfrq$counts) versus the points in vector as.vector(myfrq$breaks);
Set the **main** title for the plot as "Cost of living index";
Set the X-axis title by specifying **xlab** as "Cost of Living Index";
Set the Y-axis title by specifying **ylab** as "Number of weeks".

Line 4
>text(x=seq(15,205,by=10),y=c(0,5,10,20,9,6,2,0),labels=c("A","B","C","D",
+"E""F","G","H"),pos=4,col="red")

The command **text** draws the strings given in the vector labels at the coordinates given by x and y

Option **pos** specifies a position parameter for the text
1 indicates position below the specified coordinates
2 indicates position to the left of the specified coordinates
3 indicates position above the specified coordinates
4 indicates to the right of the specified coordinates

Option **col** is used to specify the color

Line 5
>lines(as.vector(myfrq$breaks),as.vecxtor(myfreq$counts),lwd=2,col="blue",
+type="o")

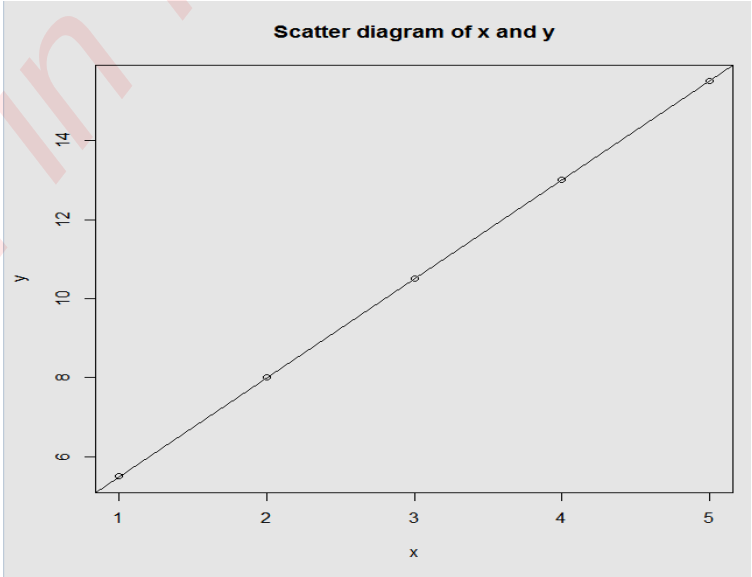
Line charts are created with the command **lines**(x,y,type=) where x and y are numeric vectors of (x,y) points to connect.
The option **type** contains the following values:
p – point
l- lines
o – overplotted points and lines
b.c points (empty if "c") joined by lines
s,S – Stair steps
h – histogram like vertical line
n – does not produce any points or line

The option **lwd** specifies the thickness of the line; 2 specify the line width in multiples of 1/96 inch

Example 3.8
Find the scatter diagram of the variables x and y. Does it reveal any relationship between the variables?

x	y
1	5.5
2	8
3	10.5
4	13
5	15.5

Solution
```
>x<-c(1,2,3,4,5)
>y<-c(5.5,8,10.5,13,15.5)
>plot(x,y,main="Scatter diagram of x and y",xlab="x",ylab="y")
>#Add a straight line through the current plot
>abline(lm(y~x))
```



Explanation:
Line 1 - 2
x <- c(1,2,3,4,5)
y <- c(5.5,8,10.5,13,15.5)
Construct vectors x and y with the given values.

Line 3
plot(x,y,main = "Scatter diagram of x and y", xlab="x",ylab="y")
A scatter plot is used when you have two variables to plot against one another. The command plot performs this task.
Set the main heading and x-axis and y-axis label

Line 4
Add a straight line through the current plot
This is a comment line

Line 5
abline(lm(y~x))
Add a best line of fit to the plot.

Observation: There is a positive correlation among x and y variables. The shape of the line drawn through the data points, gives the nature of relationship between the two variables. A straight line is interpreted as a linear relationship; a curved shape suggests a quadratic relationship. A line that lies relatively flat before suddenly shooting up or down is interpreted as an exponential relationship. When we examine the scatter plot for outliers, values that lie abnormally far from the cluster of data points. Outliers distort the relationship between the variables. A positive association is indicated by a upward trend (positive slope) - For example, higher income correspond to higher education levels. A negative association is indicated by negative slope. Absence of significant association is indicated by the scatter plot indicates there exists no trend at all.

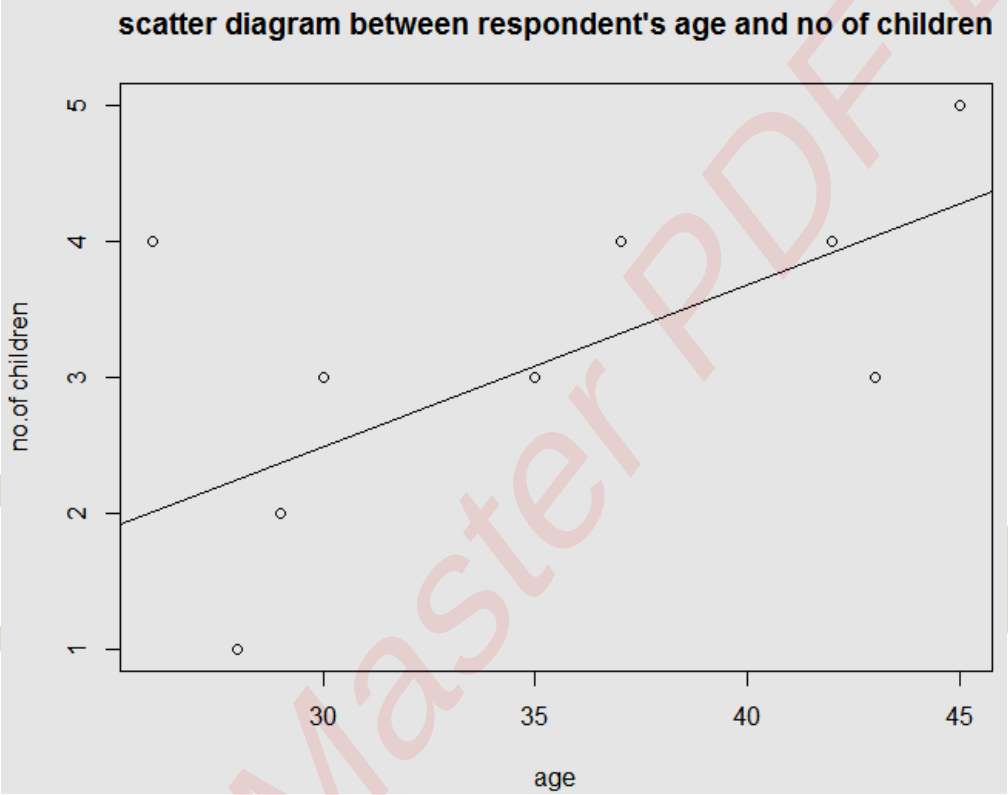
Example 3.9: The following data pertain to the respondent's age when first married and the number of children of the respondent. Plot Scatter diagram and compute the correlation coefficient.

Age	26	30	38	42	45	29	35	28	43	37
No	4	3	1	4	5	2	3	1	3	4

Solution:

```
> x<-c(26,30,28,42,45,29,35,28,43,37)
> y<-c(4,3,1,4,5,2,3,1,3,4)
> cor(x,y,method="spearman")
[1] 0.5626759
> plot(x,y,main="scatter diagram between respondent's age and no of children", xlab="age", ylab="no. of
  children")
> abline(lm(y~x))
```

Explanation:
There is a positive correlation between respondent's age when married and the number of children. From the scatter plot we can interpret that the age of the respondent may be a function of number of children. But the presence of outliers disturbs the relationship between the two variables.



MATRIX CONSTRUCTION

There are various ways to construct a matrix. When we construct a matrix directly with data elements, the matrix content is filled along the column orientation by default. The function matrix creates matrices.

matrix(data, nrow, ncol, byrow)

The data argument is usually a list of the elements that will fill the matrix.
The nrow and ncol arguments specify the dimension of the matrix. Often only one dimension argument is needed if, for example, there are 20 elements in the data list and ncol is specified to be 4 then R will automatically calculate that there should be 5 rows and 4 columns since 4*5=20.

The by row argument specifies how the matrix is to be filled. The default value for byrow is FALSE which means that by default the matrix will be filled column by column.

For example, in the following code snippet, the content of B is filled along the columns consecutively.

```
> B = matrix( c(2, 4, 3, 1, 5, 7),  nrow=3, ncol=2)
> B           # B has 3 rows and 2 columns
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
```

DETERMINANT OF A MATRIX

det ( ) function calculates the determinant of a matrix. Determinant is a generic function that returns separately the modulus of the determinant, optionally on the logarithm scale, and the sign of the determinant.

>det(A) is the code to find the determinant of the given matrix.

Example 3.10 Find the determinant of the give matrix

```
X= -2   -1    2
       2    1    0
     -3    3   -1
```

Solution:

```
> x <- matrix(c(-2,2,-3,-1,1,3,2,0,-1),3,3)
> x
     [,1] [,2] [,3]
[1,]   -2   -1    2
[2,]    2    1    0
[3,]   -3    3   -1

> det(x)
[1] 18

> determinant(x)
$modulus
[1] 2.890372
attr(,"logarithm")
[1] TRUE

$sign
[1] 1
```

TRANSPOSE OF A MATRIX

We construct the transpose of a matrix by interchanging its columns and rows with the function t .

```
> t(A)        # transpose of A
```

Example 3.11 Find the transpose of the give matrix

```
X= -2  -1   2
    2   1   0
   -3   3  -1
```

Solution:

```
> x <- t(x)
> x
     [,1] [,2] [,3]
[1,]   -2    2   -3
[2,]   -1    1    3
[3,]    2    0   -1
```

INVERSE OF A MATRIX

the command:

solve(A)

Solve(A,b)

returns a vector x in the equation b=Ax(i,e.,A^{-1}b)

DIAGONAL MATRIX

diag(x)	Creates diagonal matrix with elements of x in the principal diagonal
diag(A)	Returns a vector containing the elements of the principal diagonal
diag(k)	If k is a scalar, this creates a k x k identity matrix.

COMBINING MATRICES

The columns of two matrices having the same number of rows can be combined into a larger matrix. For example, suppose we have another matrix C also with 3 rows.

cbind(A,B,...)	Combine matrices(vectors) horizontally. Returns a matrix.
rbind(A,B,...)	Combine matrices(vectors) vertically. Returns a matrix.

Example 3.12

Use cbind() to combine the two given matrices.

```
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7   and
```

```
     [,1]
[1,]    7
[2,]    4
[3,]    2
```

Solution:

```
> C = matrix( c(7, 4, 2),  nrow=3,ncol=1)
 > C            # C has 3 rows

      [,1]
[1,]    7
[2,]    4
[3,]    2

> cbind(B, C)
     [,1] [,2] [,3]
[1,]    2    1    7
[2,]    4    5    4
[3,]    3    7    2
```

Example 3.13 : Use rbind() to combine the two given matrices.

```
> D = matrix(  c(6, 2),nrow=1,ncol=2)
> D            # D has 2 columns
```

Solution :

```
[,1] [,2]
[1,]    6    2

> rbind(B, D)
     [,1] [,2]
[1,]    2    1
[2,]    4    5
[3,]    3    7
[4,]    6    2
```

DECONSTRUCTION

We can deconstruct a matrix by applying the c function, which combines all column vectors into one.

```
> c(B)
[1] 2 4 3 1 5 7
```

MATRIX MULTIPLICATION

The operator %*% is used for matrix multiplication.

For example, A and B are square matrices of the same size, then

```
> A * B
```

is the matrix of element by element products and

```
> A %*% B
```

is the matrix product. If x is a vector, then

```
> x %*% A %*% x
```

EIGEN VALUES AND EIGENVECTORS

The function eigen(Sm) calculates the eigenvalues and eigenvectors of a symmetric matrix Sm. The result of this function is a list of two components named values and vectors. The assignment

```
> ev <- eigen(Sm)
```

will assign this list to ev. Then ev$val is the vector of eigenvalues of Sm and ev$vec is the matrix of corresponding eigenvectors. Had we only needed the eigenvalues we could have used the assignment:

```
> evals <- eigen(Sm)$values
```

evals now holds the vector of eigenvalues and the second component is discarded. If the expression

```
> eigen(Sm)
```

is used by itself as a command the two components are printed, with their names. For large matrices it is better to avoid computing the eigenvectors if they are not needed by using the expression

```
>evals <- eigen(Sm, only.values = TRUE)$values
```

EXERCISE UNIT 3

1. Draw a histogram to present the following data:

Income (in '000s – Rs)	Number of Individuals
100 – 149	21
150 – 199	32
200 – 249	52
250 – 299	105
300 – 349	62
350 – 399	43
400 – 449	18
450 – 499	9
Total	342

2. The heights of 50 students, measured to the nearest centimeter (cm) have been found to be as follows:

161	150	154	165	168	161	154	162	150	151	162	164	171
165	158	154	156	172	160	170	153	159	161	170	162	165
166	168	165	164	154	152	153	156	158	162	160	161	163
166	161	159	162	167	168	159	158	153	154	159		

Represent the data given above by a grouped frequency table taking the class intervals as 160-165, 165-170, etc., Draw a histogram to present the above data

3. Consider the marks, out of 50. Obtained by 25 students of a class to a test given in the table below:

Marks	Number of students
0 – 10	1
10 – 20	4
20 – 30	14
30 – 40	4
40 – 50	2
Total	25

Draw a frequency polygon corresponding to this frequency distribution table.

4. Represent the following data by a simple bar diagram:

Year	**Production (in Tonnes)**
1974	45
1975	40
1976	44
1977	41
1978	49
1979	55
1980	50

5. Present the profit before tax and profit after tax for the year ended 30th September 2009, 2010, 2011,2012 and 2013 respectively of the public limited company mentioned below by a bar diagram.

Financial highlights of the Public Limited Co.

Year ended 30th September	**Profit before Tax (In Lakhs of Rupees)**	**Profit after Tax (In Lakhs of Rupees)**
2009	190	79
2010	191	71
2011	200	90
2012	109	36
2013	127	89

6. Draw a Pie diagram to represent the following population in a town

Male	**2000**
Female	1800
Girls	4200
Boys	2000
Total	10,000

7. Draw a Pie diagram to represent the following data:

Type of commodity	Expenditure in '000 Rs.	
	Family A	Family B
Food	5	6
Rent	20	30
Clothes	2	3
Education	10	14
Miscellaneous	3	5
Savings	8	12
Total	48	70

8. The frequency distribution of wages in a certain factory is as follows:

Wages (in Rs.)	No. of Workers
7000 – 7499	10
7500 – 7999	18
8000 – 8499	27
8500 – 8999	20
9000 – 9499	15
9500 – 9999	8
10000 – 10499	2

Draw an Ogive curve for this distribution.

9. Given

$A = \begin{bmatrix} 2 & -4 \\ 1 & 3 \end{bmatrix}$ $B = \begin{bmatrix} 4 & -1 \\ 2 & 0 \end{bmatrix}$ $C = \begin{bmatrix} 4 \\ 3 \end{bmatrix}$ $D = \begin{bmatrix} 3 & 1 \end{bmatrix}$ $E = \begin{bmatrix} -3 & 2 & 0 \\ 1 & -1 & -2 \end{bmatrix}$

Calculate. If not possible, put undefined:

a) A + B b) 3B c) AC d) AE e) B + D f) B – 2A

10. Given the matrix $\begin{bmatrix} 1 & 6 & 5 \\ 2 & 3 & 1 \\ 0 & 2 & 4 \end{bmatrix}$ calculate the determinant.

11. Find the inverse of the matrix $\begin{pmatrix} -1 & 0 & 3 \\ -1 & 1 & -1 \\ 2 & 1 & 0 \end{pmatrix}$

UNIT 4: FUNCTIONS AND CONTROL STRUCTURES

FUNCTIONS

Most tasks are performed by calling a function in R. In fact, everything we have done so far is calling an existing function which then performs a certain task resulting in some kind of an output.

```
> length(c(1:1000))
[1] 1000
>
> max(seq(from = -20, to = 100, by = 10))
[1] 100
>
> min(seq(from = -20, to = 100, by = 10))
[1] -20
>
> sum(c(34,56,78,23,34,44))
[1] 269
>
> mean(c(34,56,78,23,34,44))
[1] 44.83333
>
> var(c(34,56,78,23,34,44))
[1] 387.3667
>
>
```

A function can be regarded as a collection of statements and is an object in R of class `function'. One of the strengths of R is the ability to extend R by writing new functions.

The general form of a function is given by:

Function_name = function (arg1, arg2, ...)
{

Body of function: a collection of valid statements

}

In the above display **arg1** and **arg2** in the function header are **input arguments** of the function. Note that a function doesn't need to have any input arguments. The body of the function consists of valid R statements. For example, the commands, functions and expressions one types, in the R console window. Normally, the last statement of the function body will be the return value of the function. This can be a vector, a matrix or any other data structure.

Some of the R built-in functions for basic statistical methods

Methods	R Functions
Histogram	hist
Stem and leaf display	stem
Box plot	boxplot
Time-series plot	ts.plot
Mean	mean
Median	median
Quantiles	quantile
Extremes	range
Variance	var
Standard deviation	sd
Covariance matrix	cov
Correlation	cor
Normal density, distribution, quantiles, random number	dnorm, pnorm, qnorm, rnorm
t density, distribution, quantiles, random number	dt, pt, qt, rt
Chi-square density, distribution, quantiles, random number	dchisq, pchisq, qchisq, rchisq
F density, distribution, quantiles, random number	df, pt, qt, rt
Binomial probabilities, distribution, quantiles, random number	dbinom, pbinom, qbinom, rbinom
Simple regression	lm
Multiple regression	lm
Analysis of variance	aov, lm, anova
Contingency table	xtabs, table
t-tests for means	t.test
Tests for proportion	prop.test
Chi-square test for independent	chisq.test
Various non-parametric functions	Friedmen.test, kruskal.test, Wilcox.test

Users can create their own functions. Some of the examples are below:

Example 4.1

```
> myfunction = function(x,y)
+ {
+   z <- x + y^2
+     return(z)
+ }
```

In the above example,

myfunction = function name
x, y inside the function = arguments

z, x and y = local variables
2 = global variable (do computation with arguments)
return (z) = output of the function

calling / using function

input x = 4
inputy = 7

answer = myfunction(inputx, inputy)

inputx, inputy and answer = Global variables

```
> inputx = 4
> inputy = 7
>
> answer = myfunction(inputx,inputy)
> answer
[1] 53
>
```

Or
Giving values directly into arguments

```
> answer = myfunction(4,7)
> answer
[1] 53
>
```

Example 4.2 Write a function that returns the product of minimum and maximum of the input vector.

Function

```
> Exerise = function ( X )
+ {
+
+ product = min (X) * max(X)
+ return(product)
+
+ }
>
>
```

Inputting values

```
> X = c(23,45,67,89,23,44,56)
> X
[1] 23 45 67 89 23 44 56
>
```

Output of product of min and max value

```
> Exerise ( X )
[1] 2047
>
```

Example 4.3 Write a function that converts temperatures from celsius to Fahrenheit.

Formula: [^{0}F] = (9/5)*[^{0}C] + 32

```
> # Inputting data
>
> Temperature = c(32,45,56,78)
> Temperature
[1] 32 45 56 78
> #Function
>
> CON_TEMP = function(celsius)
+ {
+ fahreheit = (celsius*(9/5))+32
+ return(fahreheit)
+ }
> #Output
>
> CON_TEMP (Temperature)
[1]  89.6 113.0 132.8 172.4
>
```

Example 4.4 Write a function to calculate factorial of input

```
> #input of data
>
> Data = 24
> # Function for factorial
>
> Factorial = function(X)
+ {
+  fact =1
+ while (X > 0)
+
+ {
+
+ fact = fact * x
+
+ X = X - 1
+
+ }
+
+ return(fact)
+
+ }
> Factorial (Data)
[1] 6153818012
>
```

Example 4.5

The following short function **meank** calculates the mean of a vector **x** by removing the **k** percent smallest and the **k** percent largest elements of the vector.

```
> meank = function(x, k)
+ {
+ xt = quantile(x, c(k,1-k))
+ mean( x[ x > xt[1] & x < xt[2] ])
+ }
>
> Test = rnorm(8)
> Test
[1]  0.1965621  0.4024894  0.7040447  0.2924340 -0.1679511 -0.3120777  0.4988932
[8] -0.7695666
>
> meank(Test, 0.2)
[1] 0.1808836
>
```

CONTROL STRUCTURES

Computation in R consists of sequentially evaluating statements. Statements such as **x<-1:10** or **mean(y)**, can be separated by either a semi-colon or a new line. Whenever the evaluator is presented with a syntactically complete statement, that statement is evaluated and the value returned. The result of evaluating a statement can be referred to as the value of the statement. The value can always be assigned to a symbol.

Both semicolon and new line can be used to separate statements. A semicolon always indicates the end of a statement while a new line may indicate the end of a statement. If the current statement is not syntactically complete new lines are simply ignored by the evaluator. If the session is interactive the prompt changes from '>' to '+'.

```
> x <- 0; x + 5
[1] 5
> y <- 1:10
> y
 [1]  1  2  3  4  5  6  7  8  9 10
> 1; 2
[1] 1
[1] 2
>
```

Statements can be grouped together using braces '{' and '}'. A group of statements is sometimes called a block. Single statement is evaluated when a new line is typed at the end of the syntactically complete statement. Blocks are not evaluated until a new line is entered after the closing brace.

```
> { x <- 0
+     x + 5
+ }
[1] 5
> |
```

The following shows a list of constructions to perform testing and looping. These constructions can also be used outside a function to control the flow of execution.

SHIFTING CONTROL

IF statements
Switch statements

LOOPING

For
Repeat
While

SHIFTING OF CONTROL

IF STATEMENTS

Conditional execution is available using **if** statement and the corresponding else statement. The **if/else** statement conditionally evaluates two statements. There is a condition which is evaluated and if the value is TRUE then the first statement is evaluated; otherwise the second statement will be evaluated. The **if/else** statement returns, as its output, the value of the statement that was selected. The general form of the if construction has the form

if (test)
{
...true statements...
}

else

{
...false statements...
}

where test is a logical expression like **x < 0**, **x < 0 & x > -8**. R evaluates the logical expression if it results in **TRUE** then it executes the true statements. If the logical expression results in **FALSE** then it executes the false statements. Note that for **simple if** it is not necessary to have the else block. The argument to the ***if*** statement is a logical expression.

Example 4.6

```
> x = 0.1
>   if( x < 0.2)
+   {
+      x <- x + 1
+      cat("increment that number!\n")
+   }
increment that number!
> x
[1] 1.1
>
```

The else statement can be used to specify an alternate option. In the example below note that the else statement must be on the same line as the ending brace for the previous if block.

Example 4.7

```
> x = 2.0
>   if ( x < 0.2)
+  {
+     x <- x + 1
+     cat("increment that number!\n")
+  } else
+  {
+     x <- x - 1
+     cat("nah, make it smaller.\n");
+  }
nah, make it smaller.
> x
[1] 1
>
```

Finally, the **if** statements can be chained together for multiple options. The **if** statement is considered a single code block, so more if statements can be added after the else.

Example 4.8

```
> x = 1.0
>  if ( x < 0.2)
+  {
+     x <- x + 1
+     cat("increment that number!\n")
+  } else if ( x < 2.0)
+  {
+    x <- 2.0*x
+    cat("not big enough!\n")
+  } else
+  {
+     x <- x - 1
+     cat("nah, make it smaller.\n");
+  }
not big enough!
> x
[1] 2
>
```

Example: 4.9 Adding two vectors in R of different length will cause R to recycle the shorter vector. The following function adds the two vectors by chopping off the longer vector so that it has the same length as the shorter. Vector **(Using functions)**

```
> myplus <- function(x, y){
+ n1 <- length(x)
+ n2 <- length(y)
+ if(n1 > n2){
+ z <- x[1:n2] + y
+ }
+ else{
+ z <- x + y[1:n1]
+ }
+ z
+ }
>
> myplus (1:16, 1:3)
[1] 2 4 6
>
```

SWITCH STATEMENTS

The switch takes an expression and returns a value in a list based on the value of the expression. How it does this depends on the data type of the expression.

The syntax of switch statements is

switch (object,
"value1" = {expr1},
"value2" = {expr2},
"value3" = {expr3},
{ other expressions }
)

If object has value **value1** then **expr1** is executed, if it has **value2**then **expr2** is executed and so on. If object has **no match** then **other expressions** is executed. Note that the block {other expressions} does not have to be present, the switch will return **NULL** in case object does not match any value. An expression **expr1** in the above construction can consist of multiple statements. Each statement should be separated with a **;** or on a separate line and surrounded by curly brackets.

Example 4.10

```
> X = as.integer (3)
> X
[1] 3
> Z = switch(X, 1,4,2,5,3)
> Z
[1] 2
> X = 3.5
> Z = switch(X, 1,4,2,5,3)
> Z
[1] 2
>
```

If the result of the expression is a string, then the list of items should be in the form "value N"=resultN, and the statement will return the result that matches the value.

Example 4.11

```
> y <- rnorm(5)# Generates 5 a random values using normal distibution
> y
[1] -0.06951449  1.66094671 -1.26762539 -0.95222126  0.43604385
>
> x <- "sd" # Standard deviation
>
> #Switch Statemnets to calculate SD of 5 values
>
> z <- switch(x,"mean"=mean(y),"median"=median(y),"variance"=var(y),"sd"=sd(y))
> z
[1] 1.168322
>
> x = "mean" # average
>
> z = switch(x,
+ "mean"=mean(y),
+ "median"=median(y),
+ "variance"=var(y),
+ "sd"=sd(y)
+ )
>
> z
[1] -0.03847412
>
```

LOOPING

FOR STATEMENTS

The for loop can be used to repeat a set of instructions, and it is used when you know in advance the values that the loop variable will have each time it goes through the loop. The basic format for the for loop is

for (variable in sequence)
{
Expression
}

Example 4.12

```
> for (lupe in seq(0,1,by=0.3))
+  {
+     cat(lupe,"\n");
+  }
0
0.3
0.6
0.9
>
```

Example 4.13

```
> x <- c(1,2,4,8,16)
>
> for (loop in x)
+ {
+ cat("Value of loop: ",loop, "\n")
+ }
Value of loop:  1
Value of loop:  2
Value of loop:  4
Value of loop:  8
Value of loop:  16
>
```

WHILE STATEMENTS

The **while** loop can be used to repeat a set of instructions, and it is often used when one do not know in advance how often the instructions will be executed.

The basic format for a while loop is

While (Condition)
{

Expression

}

Example 4.14

```
> lupe <- 1;
>  x <- 1
>  while(x < 4)
+  {
+     x <- rnorm(1,mean=2,sd=3)
+     cat("trying this value: ",x," (",lupe," times in loop)\n");
+     lupe <- lupe + 1
+  }
trying this value:  0.7177866  ( 1  times in loop)
trying this value:  0.3054286  ( 2  times in loop)
trying this value:  4.977732  ( 3  times in loop)
>
```

REPEAT STATEMENTS

The **repeat** loop is similar to the **while** loop. The difference is that it will always begin the loop the first time. The **while** loops will only start the loop if the condition is true the first time it is evaluated. Another difference is that

one should have to explicitly specify when to stop the loop using the **break** command. That is one need to execute the break statement to get out of the loop.

```
repeat
    {
        Expression

        break
    }
```

Example 4.15

```
> repeat
+ {
+     x <- rnorm(1)
+     if(x < -2.0) break
+ }
> x
[1] -2.557885
>
```

BREAK AND NEXT STATEMENTS

The break statement i s used to stop the execution of the current loop. It will break out of the current loop. The next statement is used to ski p the statements that follow and restart the current loop. If a for loop i s used then the next statement will update the loop variable.

Example 4.16

```
> X = runif(5)
> X
[1] 0.76393555 0.74710273 0.88863602 0.07665527 0.10348113
>
> for(lupe in X)
+  {
+      if (lupe > 2.0)
+
+      next
+
+      if( (lupe <0.6) && (lupe > 0.5))
+
+         break
+
+     cat("The value of lupe is ",lupe,"\n");
+  }
The value of lupe is  0.7639356
The value of lupe is  0.7471027
The value of lupe is  0.888636
The value of lupe is  0.07665527
The value of lupe is  0.1034811
>
```

DEBUGGING USER DEFINED R FUNCTIONS

The R language provides the user with some tools to track down unexpected behavior during the execution of (user written) functions. For example (a) A function may throw warnings at you. Although warnings do not stop the execution of a function and could be ignored, you should check out why a warning is produced. (b) A function stops because of an error. Now you must really fix the function if you want it to continue to run until the end. (c) Your function runs without warnings and errors; however the number it returns does not make any sense.

THE *trackback* FUNCTION

The first thing one can do when an error occurs is to call the function traceback. It will list the functions that were called before the error occurred. Consider the following two functions.

```
> myf = function(z)
+ {
+ X  = log(z)
+ if(X >0)
+ {
+  print("Postive")
+ }
+ else
+ {
+  print("Negative")
+ }
+ }
>
>
> testf = function(A)
+ {
+  myf(A)
+ }
>
>
> testf(-2)
Error in if (X > 0) { : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
>
```

Executing the command testf(-2) will result in an error, execute traceback to see the function calls before the error.

```
> traceback ( )
2: myf(A) at #3
1: testf(-2)
>
```

THE WARNING AND STOP FUNCTIONS

Errors and warnings message are produce while writing functions. In that situations, addition to putting ordinary print statements like print("Some message") in the function, one can use the function warning. For example,

```
> variation <- function(X)
+ {
+ if(min(X) <= 0)
+ {
+ warning("variation only useful for positive data")
+ }
+ sd(X)/mean(X)
+ }
>
> variation(rnorm(100))
[1] -8.699498
Warning message:
In variation(rnorm(100)) : variation only useful for positive data
>
```

If one wants to raise an error then they can use the function **stop**. In the above example by replacing **warning** by **stop** R would halt the execution.

```
> variation <- function(X)
+ {
+ if(min(X) <= 0)
+ {
+ #warning("variation only useful for positive data")
+ #Replacing warnig by stop
+ stop("variation only useful for positive data")
+ }
+ sd(X)/mean(X)
+ }
>
> variation(rnorm(100))
Error in variation(rnorm(100)) : variation only useful for positive data
>
```

R will treat your warnings and errors as normal R warnings and errors, which That means for example, the function traceback can be used to see the call stack when an error occurred.

STEPPING THROUGH A FUNCTION

With **traceback** one will know in which function the error occurred, where as it will not tell where error has occurred exactly. To find the error in the function one can use the function **debug**, which will tell R to execute the function in debug mode. If one want to step through everything then need to set **debug flag** for the main function and the functions that the main function calls:

By execute the function **testf**, R will display the body of the function and a browser environment is started.

```
> debug(testf)
> debug(Myf)
> testf(-2)
debugging in: testf(-2)
debug at #2: {
    Myf(A)
}
Browse[2]>
debug at #3: Myf(A)
Browse[2]>
debugging in: Myf(A)
debug at #2: {
    X = log(z)
    if (X > 0) {
        print("Postive")
    }
    else {
        print("Negative")
    }
}
Browse[3]>
debug at #3: X = log(z)
Browse[3]>
debug at #4: if (X > 0) {
    print("Postive")
} else {
    print("Negative")
}
Browse[3]>
Error in if (X > 0) { : missing value where TRUE/FALSE needed
In addition: Warning message:
In log(z) : NaNs produced
>
```

THE BROWSER FUNCTION

It may happen that an error occurs at the end of a lengthy function. To avoid stepping through the function line by line manually, the function browser can be used. Inside your function insert the **browser ( )** statement at a location where you want to enter the debugging environment.

F1 = function (S)
{
…….Some codes…….
browser ( )
…….Some codes…….
}

Run the function F1 as normally. When R reaches the **browser( )** statement then the normal execution is halted and the debug environment is started.

HYPOTHESIS TESTING

t-test

The R function **t.test( )** can be used to perform both one and two sample t-tests on vectors of data.

The function contains a variety of options and can be called as follows:

t.test (x, y = NULL, alternative = c ("two.sided", "less", "greater"), mu = 0, paired = FALSE, var.equal = FALSE, conf.level = 0.95)

Here **x** is a numeric vector of data values and **y** is an optional numeric vector of data values. If **y** is excluded, the function performs a **one-sample t-test** on the data contained in x, if it is included it performs a **two-sample t-tests** using both x and y.

The option **mu** provides a number indicating the **true value** of the **mean** (or difference in means if you are performing a two sample test) under the null hypothesis. The option **alternative** is a character string specifying the alternative hypothesis, and must be one of the following: "**two.sided**" (which is the default), "**greater**" or "**less**" depending on whether the alternative hypothesis is that the mean is different than, greater than or less than mu, respectively. For example the following call:

t.test (x, alternative = "less", mu = 10)

performs a one sample t-test on the data contained in x where the null hypothesis is that **mu =10** and the alternative is that **mu < 10**.

The option **paired** indicates whether or not you want a **paired t-test** (TRUE = yes and FALSE = no). If you leave this option out it defaults to FALSE.

The option **var.equal** is a logical variable indicating whether or not to assume the **two variances** as being **equal** when performing a two-sample t-test. If TRUE then the pooled variance is used to estimate the variance otherwise the **Welch** (or **Satterthwaite**) approximation to the degrees of freedom is used. If you leave this option out it defaults to FALSE.

Finally, the option **conf.level** determines the confidence level of the reported confidence interval for in the one-sample case and 1- 2 in the two-sample case.

ONE-SAMPLE t-TESTS

Example 4.17
An outbreak of Salmonella-related illness was attributed to ice cream produced at a certain factory. Scientists measured the level of Salmonella in 9 randomly sampled batches of ice cream. The levels (in MPN/g) were:

0.593	0.142	0.329	0.691	0.231	0.793	0.519	0.392	0.418

Is there evidence that the mean level of Salmonella in the ice cream is greater than 0.3 MPN/g?

Procedure

Let **μ** be the mean level of Salmonella in all batches of ice cream. Here the hypothesis of interest can be expressed as:

H_0: **μ** = 0.3
H_a: **μ** > 0.3

Hence, we will need to include the options **alternative="greater", mu=0.3**. Below is the relevant R-code and output:

```
> x = c(0.593, 0.142, 0.329, 0.691, 0.231, 0.793, 0.519, 0.392, 0.418)
> t.test(x, alternative="greater", mu=0.3)

        One Sample t-test

data:  x
t = 2.2051, df = 8, p-value = 0.02927
alternative hypothesis: true mean is greater than 0.3
95 percent confidence interval:
 0.3245133       Inf
sample estimates:
mean of x
0.4564444

>
```

From the output we see that the **p-value = 0.029**. Hence, there is moderately strong evidence that the mean Salmonella level in the ice cream is **above 0.3** MPN/g.

TWO SAMPLE **t – TEST**

Example 4.18 Six subjects were given a drug (treatment group) and an additional 6 subjects a placebo (control group). Their reaction time to a stimulus was measured (in **ms**). We want to perform a two-sample t-test for comparing the means of the treatment and control groups.

Control	**91**	**87**	**99**	**77**	**88**	**91**
Treatment	101	110	103	93	99	104

Procedure
Let μ_1 be the mean of the population taking medicine and μ_2 the mean of the untreated population. Here the hypothesis of interest can be expressed as:

H_0: $\mu_1 - \mu_2 = 0$
H_a: : $\mu_1 - \mu_2 < 0$

Here we will need to include the data for the treatment group in x and the data for the control group in y. We will also need to include the options **alternative="less", mu=0**. Finally, we need to decide whether or not the standard deviations are the same in both groups.

Below is the relevant R-code when assuming equal standard deviation and its output:

```
> Control = c(91, 87, 99, 77, 88, 91)
>  Treat = c(101, 110, 103, 93, 99, 104)
>  t.test(Control,Treat,alternative="less", var.equal=TRUE)

        Two Sample t-test

data:  Control and Treat
t = -3.4456, df = 10, p-value = 0.003136
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -6.082744
sample estimates:
mean of x mean of y
 88.83333 101.66667

>
```

From the output we see that the **p-value = 0.003136**. Therefore, it infers that there is different between treatment and control group.

Below is the relevant R-code when assuming unequal standard deviation and its output:

```
> Control = c(91, 87, 99, 77, 88, 91)
>  Treat = c(101, 110, 103, 93, 99, 104)
>  t.test(Control,Treat,alternative="less", var.equal=FALSE)

        Welch Two Sample t-test

data:  Control and Treat
t = -3.4456, df = 9.48, p-value = 0.003391
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
      -Inf -6.044949
sample estimates:
mean of x mean of y
 88.83333 101.66667

>
```

From both the output we see that the **p-value = 0.003136(equal) and 0.003391(Unequal)**. Therefore, it infers that there is different between treatment and control group.

PAIRED **t – TEST**

There are many experimental settings where each subject in the study is in both the treatment and control group. For example, in a matched pairs design, subjects are matched in pairs and different treatments are given to each subject in the pair. The outcomes are thereafter compared pair-wise. Alternatively, one can measure each subject twice, before and after a treatment. In either of these situations we can't use two-sample t-tests since the independence assumption is not valid. Instead we need to use a paired t-test. This can be done using the option **paired =TRUE**.

Example 4.19 A study was performed to test whether cars get better mileage on premium gas than on regular gas. Each of 10 cars was first filled with either regular or premium gas, decided by a coin toss, and mileage for that tank was recorded. The mileage was recorded again for the same car using the other kind of gasoline. We use a paired t – test to determine whether cars get significant better mileage with premium gas.

Regular	**16**	**20**	**21**	**22**	**23**	**22**	**27**	**25**	**27**	**28**
Premium	19	22	24	24	25	25	26	26	28	32

Below is the relevant R-code and output:

```
> reg = c(16, 20, 21, 22, 23, 22, 27, 25, 27, 28)
> prem = c(19, 22, 24, 24, 25, 25, 26, 26, 28, 32)
> t.test(prem,reg,alternative="greater", paired=TRUE)

        Paired t-test

data:  prem and reg
t = 4.4721, df = 9, p-value = 0.0007749
alternative hypothesis: true difference in means is greater than 0
95 percent confidence interval:
 1.180207      Inf
sample estimates:
mean of the differences
                      2

> |
```

The results show that the t-statistic is equal to **4.47** and the **p-value is 0.00075**. Since the *p*-value is very low, we reject the null hypothesis. There is strong evidence of a mean increase in gas mileage between regular and premium gasoline.

ANALYSIS OF VARIANCE

We are often interested in determining whether the means from more than two populations or groups are equal or not. To test whether the difference in means is statistically significant we can perform analysis of variance (ANOVA). In R we do this by using the function aov( ). If the ANOVA F-test shows there is a significant difference in means between the groups we may want to perform multiple comparisons between all pair-wise means to determine how they differ.

ONE WAY ANOVA

Example 4.20 A drug company tested three formulations of a pain relief medicine for migraine headache sufferers. For the experiment 27 volunteers were selected and 9 were randomly assigned to one of three drug formulations. The subjects were instructed to take the drug during their next migraine headache episode and to report their pain on a scale of 1 to 10 (10 being most pain).

Drug A	**4**	**5**	**4**	**3**	**2**	**4**	**3**	**4**	**4**
Drug B	6	8	4	5	4	6	5	8	6
Drug C	6	7	6	6	7	5	6	5	5

Solution : First we must read in the data into the appropriate format

```
> pain = c(4,5,4,3,2,4,3,4,4,6,8,4,5,4,6,5,8,6,6,7,6,6,7,5,6,5,5)
> drug = c(rep("A",9),rep("B",9),rep("C",9))
> data = data.frame(pain,drug)
```

Note the command rep("A",9) constructs a list of nine A‟s in a row. The variable drug is therefore a list of length 27 consisting of nine A‟s followed by nine B‟s followed by nine C‟s. If we print the data frame data we can see the format of the data to perform ANOVA.

```
> data
   pain drug
1     4    A
2     5    A
3     4    A
4     3    A
5     2    A
6     4    A
7     3    A
8     4    A
9     4    A
10    6    B
11    8    B
12    4    B
13    5    B
14    4    B
15    6    B
16    5    B
17    8    B
18    6    B
19    6    C
20    7    C
21    6    C
22    6    C
23    7    C
24    5    C
25    6    C
26    5    C
27    5    C
>
```


Next, the R function aov() can be used for fitting ANOVA models. The general form is

aov(response ~ factor, data=data_name)

where *response* represents the response variable and *factor* the variable that separates the data into groups. Both variables should be contained in the data frame called *data_name*. Once the ANOVA model is fit, one can look at the results using the summary( ) function. This produces the standard ANOVA table.

```
> results = aov(pain ~ drug, data=data)
> summary(results)
            Df Sum Sq Mean Sq F value   Pr(>F)
drug         2  28.22  14.111   11.91 0.000256 ***
Residuals   24  28.44   1.185
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
```

Studying the output of the ANOVA table above we see that the F-statistic is **11.91** with a **p-value** equal to **0.0003**. We clearly reject the null hypothesis of equal means for all three drug groups.

MULTIPLE COMPARISONS

The ANOVA F-test answers the question whether there are significant differences in the K population means. However, it does not provide us with any information about how they differ. Therefore when you reject H_0 in ANOVA, additional analyses are required to determine what is driving the difference in means. The function pairwise.t.test computes the pair-wise comparisons between group means with corrections for multiple testing. The general form is

pairwise.t.test (response, factor, p.adjust = method, alternative = c ("two.sided", "less", "greater"))

Here *response* is a vector of observations (the response variable), *factor* a list of factors and p.adjust is the correction method (e.g., "Bonferroni").

```
> pairwise.t.test(pain, drug, p.adjust="bonferroni")

        Pairwise comparisons using t tests with pooled SD

data:  pain and drug

  A       B
B 0.00119 -
C 0.00068 1.00000

P value adjustment method: bonferroni
>
```

The results state that the difference in means is not significantly different between drugs **B** and **C** (p-value = 1.00), but both are significantly different from drug **A** (p-values = **0.00119** and **0.00068**, respectively). Hence, we can conclude that the mean pain is significantly different for drug **A**.

Another multiple comparisons procedure is Tukey"s method (a.k.a. Tukey's Honest Significance Test). The function TukeyHSD( ) creates a set of confidence intervals on the differences between means with the specified family-wise probability of coverage. The general form is

TukeyHSD(x, conf.level = 0.95)

Here x is a fitted model object (e.g., an aov fit) and conf.level is the confidence level.

```
> TukeyHSD(results, conf.level = 0.95)
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = pain ~ drug, data = data)

$drug
        diff         lwr      upr     p adj
B-A 2.1111111  0.8295028 3.392719 0.0011107
C-A 2.2222222  0.9406139 3.503831 0.0006453
C-B 0.1111111 -1.1704972 1.392719 0.9745173

>
```

These results show that the **B-A** and **C-A** differences are **significant** (**p=0.0011** and **p=0.00065**, respectively), while the **C-B** difference is **not** (**p=0.97**). This confirms the results obtained using Bonferroni correction.

TWO-WAY ANALYSIS OF VARIANCE

Two-way ANOVA is used to compare the means of populations that are classified in two different ways, or the mean responses in an experiment with two factors. We fit two-way ANOVA models in R using the function **lm( ).** For example, the command:

> lm(*Response ~ FactorA + FactorB*)

fits a two-way ANOVA model without interactions. In contrast, the command

> lm(*Response ~ FactorA + FactorB + FactorA*FactorB*)

includes an interaction term. Here both *FactorA* and *FactorB* are categoricalvariables, while *Response* is quantitative.

Example 4.21

A study was performed to test the efficiency of a new drug developed to increase high-density lipoprotein (HDL) cholesterol levels in patients. 18 volunteers were split into 3 groups (Placebo/5 mg/10 mg) and the difference in HDL levels before and after the treatment was measured. The 18 volunteers were also categorized into two age groups (18-39/ =40).

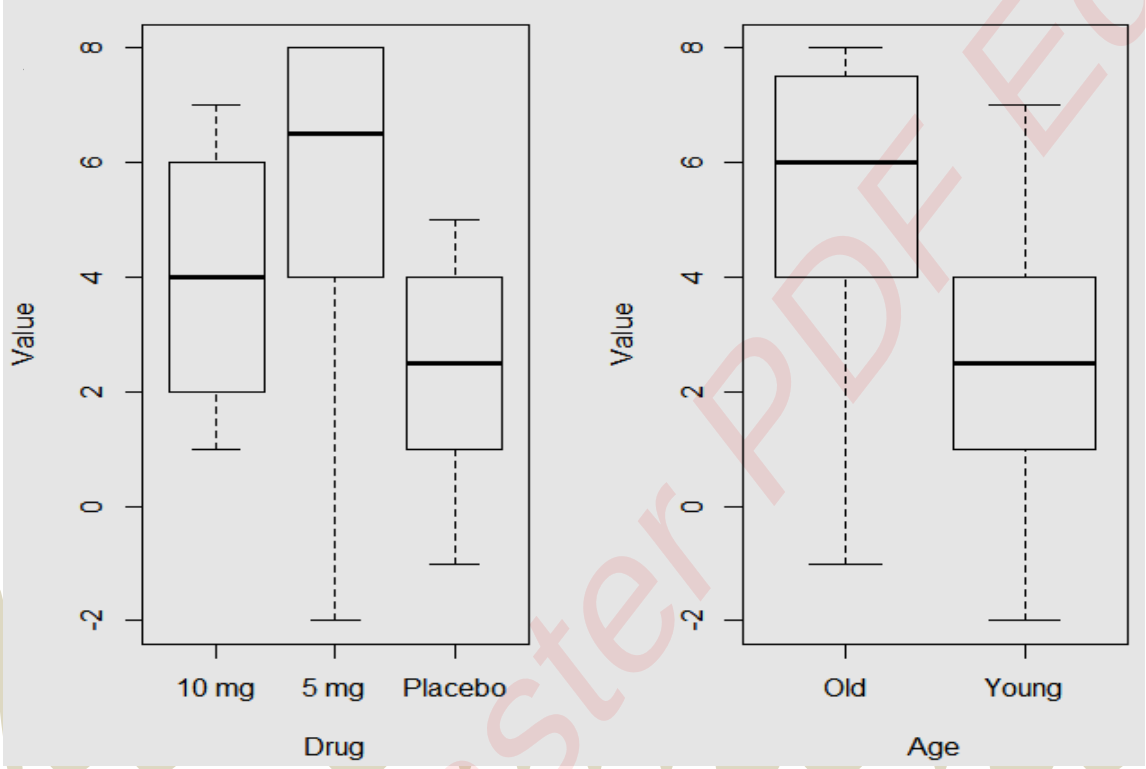
We are interested in determining the answers to the following questions: Does the amount of drug have an effect on the HDL level? Does the age of the patient have an effect on the HDL level? Is there an interaction between age and amount of drug? We begin by reading in the data and making some exploratory plots of the data.

Soulation Reading data from Excel files stored in F drive

```
> Two_ANO = read.csv("F:/DATA_TA.csv")
> Two_ANO
      Drug   Age Value
1  Placebo Young     4
2  Placebo   Old     3
3  Placebo   Old    -1
4  Placebo Young     2
5  Placebo Young     1
6  Placebo Young     5
7     5 mg Young    -2
8     5 mg   Old     8
9     5 mg Young     7
10    5 mg   Old     6
11    5 mg Young     4
12    5 mg   Old     8
13   10 mg Young     2
14   10 mg Young     1
15   10 mg   Old     5
16   10 mg   Old     7
17   10 mg   Old     6
18   10 mg Young     3
>
```

To make side-by-side boxplots:

```
> par(mfrow=c(1,2))
> plot(Value ~ Drug + Age, data=Two_ANO)
```



Judging by the boxplots there appears to be a difference in HDL level for the different drug levels. However, the difference is less pronounced between the two age groups.

An interaction plot displays the levels of one factor on the x-axis and the mean response for each treatment on the y-axis. In addition, it shows a separate line connecting the means corresponding to each level of the second factor. When no interaction is present the lines should be roughly parallel.

These types of plots can be used to determine whether an interaction term should be included in our ANOVA model.

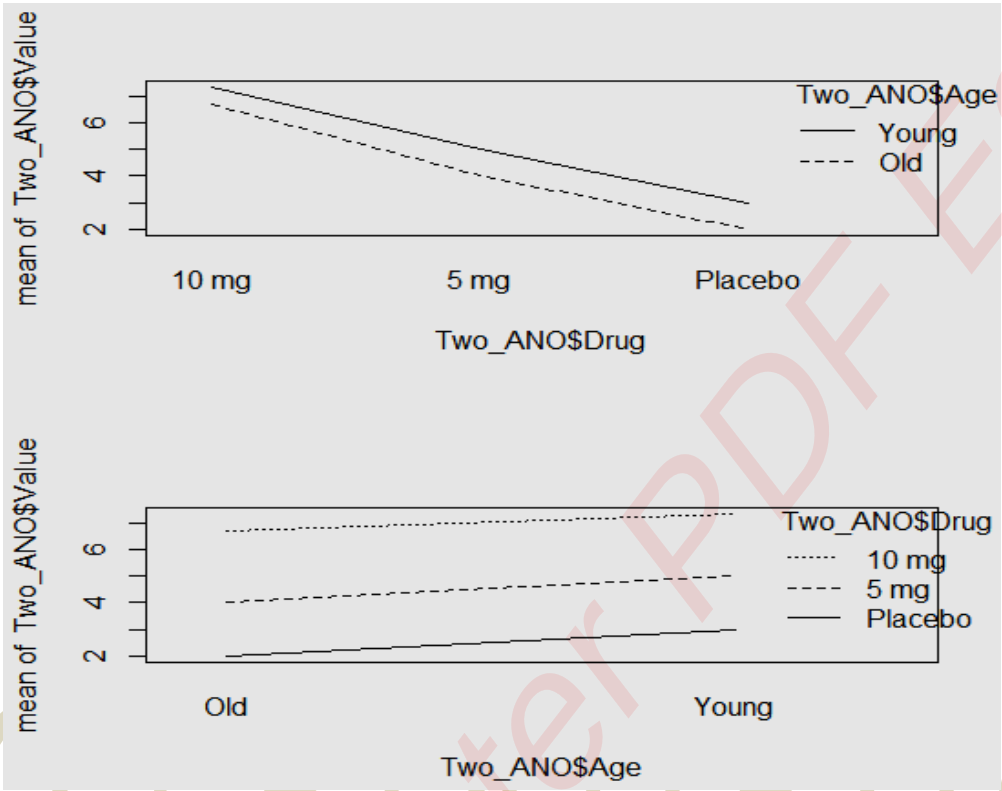
Interaction plots can be made in R using the command

```
> interaction.plot(factorA, factorB, Response)
```

If we switch the order of factor A and B we alter which variable is plotted on the x-axis

Create an interaction plot for the HDL data.

```
> par(mfrow=c(2,1))
> interaction.plot(Two_ANO$Drug, Two_ANO$Age, Two_ANO$Value)
> interaction.plot(Two_ANO$Age, Two_ANO$Drug, Two_ANO$Value)
```



The interaction plots look roughly parallel, but to confirm we fit a two-way ANOVA with an interaction term

```
> results = lm(Value ~ Drug + Age + Drug*Age, data=Two_ANO)
> anova(results)
Analysis of Variance Table

Response: Value
          Df Sum Sq Mean Sq F value Pr(>F)
Drug       2 56.778 28.3889  6.9054 0.0101 *
Age        1  3.391  3.3910  0.8248 0.3816
Drug:Age   2  0.109  0.0545  0.0133 0.9868
Residuals 12 49.333  4.1111
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
>
```

Studying the output of the ANOVA table we see that there is no evidence of a significant interaction effect (F=0.0133, p=0.9868). We therefore cannot conclude that there is an interaction between age and the amount of drug taken. The test for the main effect of treatment (F=6.9054, p<0.0101) shows a significant drug effect on the HDL level. Finally, the test for the main effect of age (F=0.8248, p=0.3.816) tells us there is not enough evidence to conclude that there is a significant age effect.

EXERCISE – UNIT 4

1. A researcher wishes to learn if a certain drug slows the growth of tumors. She obtained mice with tumors and randomly divided them into two groups. She then injected one group of mice with the drug and used the second group as a control. After 2 weeks, she sacrificed the mice and weighed the tumors. The weight of tumors for each group of mice is below.

Group A (Treatment)	**0.72**	**0.68**	**0.69**	**0.66**	**0.57**	**0.66**	**0.70**	**0.63**	**0.71**	**0.73**
Group B (Control)	0.71	0.83	0.89	0.57	0.68	0.74	0.75	0.67	0.80	0.78

The researcher is interested in learning if the drug reduces the growth of tumors. Her hypothesis is: The mean weight of tumors from mice in group A will be less than the mean weight of mice in group B.

2. A random sample of 22 fifth grade pupils have a grade point average of 5.0 in maths with a standard deviation of 0.452, whereas marks range from 1 (worst) to 6 (excellent). The grade point average (GPA) of all fifth grade pupils of the last five years is 4.7. Is the GPA of the 22 pupils different from the populations' GPA?

3. 10 participants were given 2 minutes to read a simple text and a complex text, each text containing 10 words. They were informed that after 10 minutes they would be asked to recall the words of both texts. The same participants were given both texts. Half of the participants were given the simple text first and half were given the complex text first.

Participants	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Simple text	9	4	7	5	10	7	9	4	8	5
Complex text	3	3	5	6	4	3	7	4	1	6

4. Twelve cars were equipped with radial tires and driven over a test course. Then the same 12 cars (with the same drivers) were equipped with regular belted tires and driven over the same course. After each run, the cars' gas economy (in km/l) was measured. Is there evidence that radial tires produce better fuel economy? (Assume normality of data, and use alpha = .05.)

Gas eco.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Simple text	4.2	4.7	6.6	7.0	6.7	4.5	5.7	6.0	7.4	4.9	6.1	5.2
Complex text	4.1	4.9	6.2	6.9	6.8	4.4	5.7	5.8	6.9	4.7	6.0	4.9

5. Suppose the National Transportation Safety Board (NTSB) wants to examine the safety of compact cars, midsize cars, and full-size cars. It collects a sample of three for each of the treatments (cars types). Using the hypothetical data provided below, test whether the mean pressure applied to the driver's head during a crash test is equal for each types of car. Use $\alpha = 5\%$.

Compact cars	Midsize cars	Full size cars
643	469	484
655	427	456
702	525	402

6. A firm wishes to compare four programs for training workers to perform a certain manual task. Twenty new employees are randomly assigned to the training programs, with 5 in each program. At the end of the training period, a test is conducted to see how quickly trainees can perform the task. The number of times the task is performed per minute is recorded for each trainee, with the following results:

Observation	Program1	Program2	Program 3	Program4
1	9	10	12	9
2	12	6	14	8
3	14	9	11	11
4	11	9	13	7
5	13	10	11	8

UNIT 5: CORRELATION AND REGRESSION

CORRELATION

A Correlation expresses the extent to which the two variables vary together. A positive correlation means that as one variable increases so does the other. For example, there is a strong positive correlation between the amount of exercise and the percentage of fat. A negative correlation is one when one variable increases as the other decreases. If two variables are measured on an interval scale use the Pearson product moment correlation coefficient. When data is ordinal use the Spearman Rank correlation coefficient.

In R language the

cor( ) function is used to produce correlation
cov( ) function to produce covariance.

Built-in to the base distribution of the program are three routines; for Pearson, Kendal and Spearman Rank correlations. A simplified format is cor(x, use=, method=) where

Option	Description
X	Matrix or data frame
use	Specifies the handling of missing data. Options are all.obs (assumes no missing data - missing data will produce an error), complete.obs (listwise deletion), and pairwise.complete.obs (pairwise deletion)
method	Specifies the type of correlation. Options are pearson, spearman or kendall.

Correlation/covariance among numeric variables

Eg: cor(bmi, use="complete.obs", method="kendall")

Unfortunately, neither cor( ) or cov( ) produce tests of significance, although we can use the cor.test( ) function to test a single correlation coefficient.

BIVARIATE CORRELATION

Bivariate correlation evaluates the degree of relationship between two quantitative variables. Pearson Correlation (r) is the most commonly used bivariate correlation technique, it measures the association between two quantitative variables without distinction between the independent and dependent variables.

In R language to run a correlation test we should type:

> cor(var1, var2, method = "method")

The above code generates correlations between the two variables. The default method is “pearson” we can omit this if that is what we want. If we type "kendall" or "spearman" then we will get the appropriate correlation coefficient.

Correlation coefficients code in r language	
The default correlation returns the pearson correlation coefficient	cor(var1, var2)
If you specify "spearman" you will get the spearman correlation coefficient	cor(var1, var2, method = "spearman")
If you use a dataset instead of separate variables you will return a matrix of all the pairwise correlation coefficients	cor(dataset, method = "pearson")

CORRELATION AND SIGNIFICANCE TESTS

Getting a correlation coefficient is generally only half the story, we also need to check whether the relationship is significant.

Correlation Significance tests		
The default method is "pearson"	cor.p	= cor.test(var1, var2)
If you specify "spearman" you will get the spearman correlation coefficient	cor.s	= cor.test(var1, var2, method = "spearman")

GRAPHING THE CORRELATION

Scatter Plot is used to graph the correlation and it is the simplest method which gives the degree of correlation between two variables. The basic plot is plot()

R has various default parameters set e.g. the axes are labelled as the factor name and the plotting symbol is set as an open circle.

Correlation graphs	
Use the basic defaults to create a scatter plot of your two variables	plot(x.var, y.var)
This changes the axes titles	plot(x.var, y.var, xlab="X-axis", ylab="Y-axis")
This changes the plotting symbol to a solid circle	plot(x.var, y.var, pch=16)
Adds a line of best fit to your scatter plot (don't do this for non-parametric plots).	abline(lm(y.var ~ x.var)

EXAMPLE 5.1 The data shows the number of minutes it took for 10 mechanics to assemble a piece of machinery in the morning x and in the evening y. Compute spearman's rank correlation coefficient.

X	11.1	10.3	12.0	15.1	13.7	18.5	17.3	14.2	14.8	15.3
Y	10.9	14.2	13.8	21.5	13.2	21.1	16.4	19.3	17.4	19.0

Solution:

```
> x<-c(11.1,10.3,12.0,15.1,13.7,18.5,17.3,14.2,14.8,15.3)
> y<-c(10.9,14.2,13.8,21.5,13.2,21.1,16.4,19.3,17.4,19.0)
> cor(x,y,method="spearman")
[1] 0.6969697
```

Explanation
Line1-2
Stores the values in the variables x and y

Line 3
The cor ( ) finds the correlation coefficient between the two variables x and y using Spearman's rank correlation method

Line 4
Gives the correlation value 0.6969697
It shows that there is a strong positive correlation between the two variables. (i,e)The number of minutes took for 10 mechanics to assemble a piece of machinery in morning and evenings are positively correlated.

REGRESSION

Regression is a statistical measure that attempts to determine the strength of the relationship between one dependent variable (usually denoted by Y) and a series of other changing variables (known as independent variables). The two basic types of regression are linear regression and multiple regression. Linear regression uses one independent variable to explain and/or predict the outcome of Y, while multiple regression uses two or more independent variables to predict the outcome.

The general form of equation for each type of regression is:

Linear Regression: $Y = a + bX + e$

Multiple Regression: $Y = a + b_1X_1 + b_2X_2 + B_3X_3 + ... + B_tX_t + e$
Where

Y= the variable that we are trying to predict X= the variable that we are using to predict Y
a= the intercept b= the slope e= the regression residual.
Using the regression equation, the dependent variable may be predicted from the independent variable. The slope of the regression line (b) is defined as the rise divided by the run. The y intercept (a) is the point on the y axis

where the regression line would intercept the y axis. The slope and y intercept are incorporated into the regression equation. The intercept is usually called the constant, and the slope is referred to as the coefficient. Since the regression model is usually not a perfect predictor, there is also an error term(e) in the equation.

REGRESSION LINE

The regression line (known as the *least squares line*) is a plot of the expected value of the dependent variable for all values of the independent variable. Technically, it is the line that "minimizes the squared residuals". The regression line is the one that best fits the data on a scatterplot.

The significance of the slope of the regression line is determined from the t-statistic. It is the probability that the observed correlation coefficient occurred by chance if the true correlation is zero. Some researchers prefer to report the F-ratio instead of the t-statistic. The F-ratio is equal to the t-statistic squared.

The t-statistic, for the significance of the slope is essentially a test to determine if the regression model (equation) is usable. If the slope is significantly different than zero, then we can use the regression model to predict the dependent variable for any value of the independent variable.

On the other hand, take an example where the slope is zero. It has no prediction ability, because for every value of the independent variable, the prediction for the dependent variable would be the same. Knowing the value of the independent variable would not improve our ability to predict the dependent variable. Thus, if the slope is not significantly different than zero, don't use the model to make predictions.

R SQUARE

The coefficient of determination (r-squared) is the square of the correlation coefficient. Its value may vary from zero to one. It has the advantage over the correlation coefficient in which it may be interpreted directly as the proportion of variance in the dependent variable that can be accounted for by the regression equation. For example, an r-squared value of .49 means that 49% of the variance in the dependent variable can be explained by the regression equation. The other 51% is unexplained.

STANDARD ERROR

The standard error of the estimate for regression measures the amount of variability in the points around the regression line. It is the standard deviation of the data points, as they are distributed around the regression line. The standard error of the estimate can be used to develop confidence intervals around a prediction.

REGRESSION STEP-BY-STEP

Here is a step by step guide to perform a regression. Just copy the commands you need (one at a time) and paste into R. Edit as required for your data set and variable names.

STEP –BY-STEP REGRESSION

- First create your data file. Use a spreadsheet and make each column a variable. Each row is a replicate. The first row should contain the variable names. Save this as a .CSV file.
- Read the data into R and save as some name - **your.data = resd.csv(file.choose ( ))**

- Allow the factors within the data to be accessible to R – **attach (your.data)**
- Have a first look at the data as a pairs graph (plots all combinations as scatter plots) – **paris (your.data)**
- Decide on the model, run it and assign the result to a new variable – **your.lm – lm(Y.var ~ X1.var + X2.var)**
- See the basic coefficients of your regression - **your.lm**
- A more detailed summary of your regression - **summary(your.lm)**
- Examine an individual coefficient - **your.lm$coeff["x1.var"]**
- Calculate the beta coefficient **- beat.X1 = your.lm$coeff ["X1.var"] * sd (X1.var) / sd(Y.var)**
- Display all your beta coefficients - **cat(beta.x1, beta.x2, beta.x3)**
- Calculate the R-squared components – **R2.X1 = beta.X1 * cor(Y.var, X1.Var)**
- Display all your R-squared values - **cat(R2.x1, R2.x2, R2.x3)**
- Plot a graph of two variables from your regression - **plot(x.var, y.var, xlab="x-label", ylab="y-label"))**
- Add a line of best fit - **abline(lm(y.var ~ x.var)**

Example 5.2 Find the equation of the regression line of y on x

x	y
1	5.5
2	8
3	10.5
4	13
5	15.5

Solution:

```
> x<-c(1,2,3,4,5)
> y<-c(5.5,8,10.5,13,15.5)
> reg<-lm(y~x)
> reg
```

#Output:

```
Call:
lm(formula = y ~ x)
Coefficients:
(Intercept)          x
 3.0         2.5
```

Explanation

Line 1-2
Stores the values in the variable x and y.

Line 3
>reg<-lm(y~x)
lm is used to fit linear models.It gives the coefficient and intercept values to fit the equation.

Line 4
>reg
It displays the values.
From the output the regression eqaution is y=3.0+2.5x

Example 5.3 Obtain a linear relationship between weight (kg) and height (cm) of 10 subjects.

Height	175	168	170	171	169	165	165	160	180	186
Weight	80	68	72	75	70	65	62	60	85	90

Solution:
>height = c(175, 168, 170, 171, 169, 165, 165, 160, 180, 186)
>weight = c(80, 68, 72, 75, 70, 65, 62, 60, 85, 90)
>model = lm(formula = height ~ weight, x=TRUE, y=TRUE)
>model

Call:
lm(formula = height ~ weight, x = TRUE, y = TRUE)

Coefficients:
(Intercept) weight
115.2002 0.7662
model <- lm(height ~ weight)
summary(model)

Call:
lm(formula = height ~ weight)

Residuals:
Min 1Q Median 3Q Max
-1.6622 -0.9683 -0.1622 0.5679 2.2979

Coefficients:
Estimate Std. Error t value Pr(>|t|)
(Intercept) 115.20021 3.48450 33.06 7.64e-10 ***
weight 0.76616 0.04754 16.12 2.21e-07 ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.405 on 8 degrees of freedom
Multiple R-squared: 0.9701, Adjusted R-squared: 0.9664
F-statistic: 259.7 on 1 and 8 DF, p-value: 2.206e-07

Explanation

The regression equation is

Height=115.2002+0.7662 weight

Since the pvalue of the test is 16.12 is greater than 0.05 we reject the hypothesis.Therefore the model we found is significant. The Multiple R-squared is the coefficient of determination. It provides a measure of how well future outcomes are likely to be predicted by the model. In this case the R suare value is 0.9701. Therefore 97.01% of data is well predicted.

LOGISTIC REGRSSION

A logistic regression is typically used when there is one dichotomous outcome variable (such as winning or losing), and a continuous predictor variable which is related to the probability or odds of the outcome variable. It can also be used with categorical predictors, and with multiple predictors.

Illustration of logistic regression was given by using built-in dataset of **mtcars .** In the examples below, we will use **vs** as the outcome variable, **mpg** as a continuous predictor, and **am** as a categorical (dichotomous) predictor.

```
> data(mtcars)
> mtcars
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
Merc 280C         17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
Merc 450SE        16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
Merc 450SL        17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
Merc 450SLC       15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
Fiat 128          32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
Honda Civic       30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
Toyota Corolla    33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
Toyota Corona     21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
Dodge Challenger  15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
AMC Javelin       15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
Camaro Z28        13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
Pontiac Firebird  19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
Fiat X1-9         27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
Porsche 914-2     26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
Lotus Europa      30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
Ford Pantera L    15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
Ferrari Dino      19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
Maserati Bora     15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
Volvo 142E        21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
>
```

Selecting studying variables using subset function

```
> DF =subset(mtcars, select = c(mpg, am,vs))
> DF
                   mpg am vs
Mazda RX4         21.0  1  0
Mazda RX4 Wag     21.0  1  0
Datsun 710        22.8  1  1
Hornet 4 Drive    21.4  0  1
Hornet Sportabout 18.7  0  0
Valiant           18.1  0  1
Duster 360        14.3  0  0
Merc 240D         24.4  0  1
Merc 230          22.8  0  1
Merc 280          19.2  0  1
Merc 280C         17.8  0  1
Merc 450SE        16.4  0  0
Merc 450SL        17.3  0  0
Merc 450SLC       15.2  0  0
Cadillac Fleetwood  10.4  0  0
Lincoln Continental 10.4  0  0
Chrysler Imperial   14.7  0  0
Fiat 128          32.4  1  1
Honda Civic       30.4  1  1
Toyota Corolla    33.9  1  1
Toyota Corona     21.5  0  1
Dodge Challenger  15.5  0  0
AMC Javelin       15.2  0  0
Camaro Z28        13.3  0  0
Pontiac Firebird  19.2  0  0
Fiat X1-9         27.3  1  1
Porsche 914-2     26.0  1  0
Lotus Europa      30.4  1  1
Ford Pantera L    15.8  1  0
Ferrari Dino      19.7  1  0
Maserati Bora     15.0  1  0
Volvo 142E        21.4  1  1
>
```

CONTINUOUS PREDICTOR AND DICHOTOMOUS OUTCOME

If the data set has one dichotomous and one continuous variable, and the continuous variable is a predictor of the **probability** the dichotomous variable, then a logistic regression might be appropriate. In this example, mpg is the continuous predictor variable, and vs is the dichotomous outcome variable.

```
> # Do the logistic regression - both of these have the same effect.
> # ("logit" is the default model when family is binomial.)
> logr.vm <- glm(vs ~ mpg, data=DF, family=binomial)
> #print information about the model
> logr.vm

Call:  glm(formula = vs ~ mpg, family = binomial, data = DF)

Coefficients:
(Intercept)          mpg
    -8.8331       0.4304

Degrees of Freedom: 31 Total (i.e. Null);  30 Residual
Null Deviance:      43.86
Residual Deviance: 25.53        AIC: 29.53
>
```

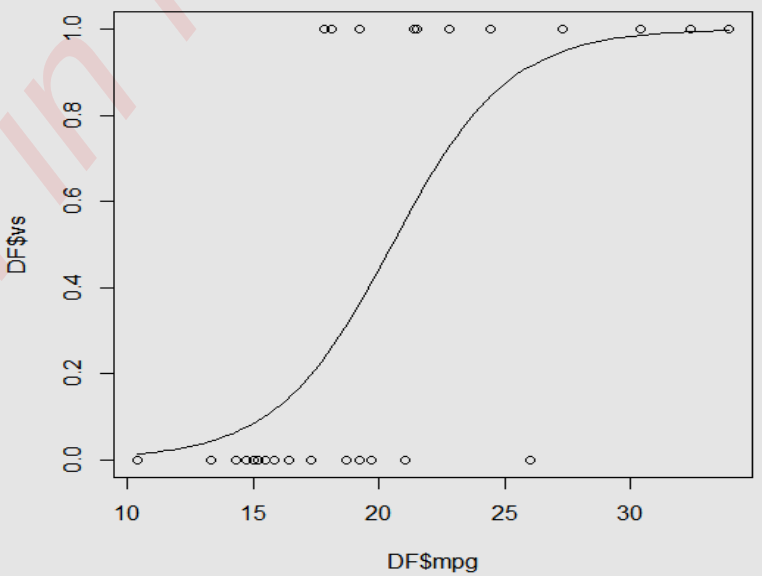
For more information about the model

```
> summary(logr.vm)

Call:
glm(formula = vs ~ mpg, family = binomial, data = DF)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.2127  -0.5121  -0.2276   0.6402   1.6980

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -8.8331     3.1623  -2.793  0.00522 **
mpg           0.4304     0.1584   2.717  0.00659 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.860  on 31  degrees of freedom
Residual deviance: 25.533  on 30  degrees of freedom
AIC: 29.533

Number of Fisher Scoring iterations: 6
```

PLOTTING

The data and logistic regression model can be plotted with ggplot2 or standard graphics

```
> library(ggplot2)
> plot(DF$mpg, DF$vs)
> curve(predict(logr.vm, data.frame(mpg=x), type="response"), add=TRUE)
>
```



DICHOTOMOUS PREDICTOR AND DICHOTOMOUS OUTCOME

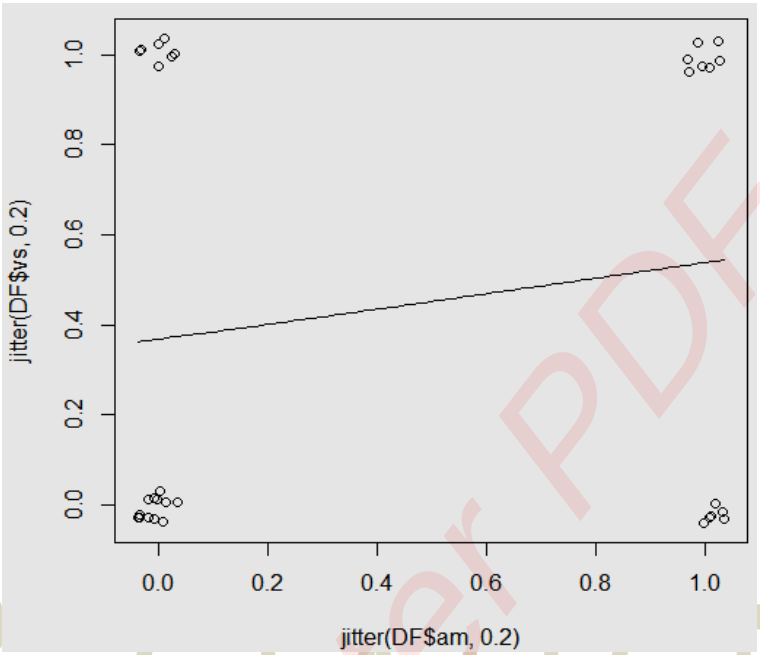
This proceeds in much the same way as above. In this example, am is the dichotomous predictor variable, and vs is the dichotomous outcome variable.

```
> # Do the logistic regression - both of these have the same effect.
> # ("logit" is the default model when family is binomial.)
> logr.va <- glm(vs ~ am, data=DF, family=binomial)
> #print information about the model
> logr.va

Call:  glm(formula = vs ~ am, family = binomial, data = DF)

Coefficients:
(Intercept)           am
    -0.5390       0.6931

Degrees of Freedom: 31 Total (i.e. Null);  30 Residual
Null Deviance:      43.86
Residual Deviance: 42.95        AIC: 46.95
> |
> # more information about the model
>
> summary(logr.va)

Call:
glm(formula = vs ~ am, family = binomial, data = DF)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.2435  -0.9587  -0.9587   1.1127   1.4132

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -0.5390     0.4756  -1.133    0.257
am            0.6931     0.7319   0.947    0.344

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.860  on 31  degrees of freedom
Residual deviance: 42.953  on 30  degrees of freedom
AIC: 46.953

Number of Fisher Scoring iterations: 4
```

PLOTTING

The data and logistic regression model can be plotted with ggplot2 or standard graphics, although the plots are probably less informative than those with a continuous variable. Because there are only 4 locations for the points to go, it will help to jitter the points so they do not all get over plotted.

```
> plot(jitter(DF$am, .2), jitter(DF$vs, .2))
> curve(predict(logr.va, data.frame(am=x), type="response"), add=TRUE)
> |
```



CONTINUOUS PREDICTOR AND DICHOTOMOUS PREDICTOR AND DICHOTOMOUS OUTCOME

This is similar to the previous examples. In this example, mpg is the continuous predictor, am is the dichotomous predictor variable, and vs is the dichotomous outcome variable.

```
> logr.vma <- glm(vs ~ mpg + am, data=DF, family=binomial)
>
> logr.vma

Call:  glm(formula = vs ~ mpg + am, family = binomial, data = DF)

Coefficients:
(Intercept)          mpg           am
   -12.7051       0.6809      -3.0073

Degrees of Freedom: 31 Total (i.e. Null);  29 Residual
Null Deviance:      43.86
Residual Deviance: 20.65        AIC: 26.65
> |
```

For more information

```
> summary(logr.vma)

Call:
glm(formula = vs ~ mpg + am, family = binomial, data = DF)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-2.05888  -0.44544  -0.08765   0.33335   1.68405

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -12.7051     4.6252  -2.747  0.00602 **
mpg           0.6809     0.2524   2.698  0.00697 **
am           -3.0073     1.5995  -1.880  0.06009 .
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.860  on 31  degrees of freedom
Residual deviance: 20.646  on 29  degrees of freedom
AIC: 26.646

Number of Fisher Scoring iterations: 6
```

MULTIPLE PREDICTORS WITH INTERACTIONS

It is possible to test for interactions when there are multiple predictors. The interactions can be specified individually, as with a + b + c + a:b + b:c + a:b:c, or they can be expanded automatically, with a * b * c. It is possible to specify only a subset of the possible interactions, such as a + b + c + a:c.

This case proceeds as above, but with a slight change: instead of the formula being vs ~ mpg + am, it is vs ~ mpg * am, which is equivalent to vs ~ mpg + am + mpg:am.

```
> # Do the logistic regression - both of these have the same effect.
> logr.vmai <- glm(vs ~ mpg * am, data=DF, family=binomial)
> logr.vmai <- glm(vs ~ mpg + am + mpg:am, data=DF, family=binomial)
> logr.vmai

Call:  glm(formula = vs ~ mpg + am + mpg:am, family = binomial, data = DF)

Coefficients:
(Intercept)          mpg           am       mpg:am
   -20.4784       1.1084      10.1055      -0.6637

Degrees of Freedom: 31 Total (i.e. Null);  28 Residual
Null Deviance:      43.86
Residual Deviance: 19.12        AIC: 27.12
>
```

For more information

```
> summary(logr.vmai)

Call:
glm(formula = vs ~ mpg + am + mpg:am, family = binomial, data = DF)

Deviance Residuals:
     Min        1Q    Median        3Q       Max
-1.70566  -0.31124  -0.04817   0.28038   1.55603

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept) -20.4784    10.5525  -1.941   0.0523 .
mpg           1.1084     0.5770   1.921   0.0547 .
am           10.1055    11.9104   0.848   0.3962
mpg:am       -0.6637     0.6242  -1.063   0.2877
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 43.860  on 31  degrees of freedom
Residual deviance: 19.125  on 28  degrees of freedom
AIC: 27.125

Number of Fisher Scoring iterations: 7
```


EXERCISE – UNIT 5

1. Five children aged 2, 3, 5, 7 and 8 years old weigh 14, 20, 32, 42 and 44 kilograms respectively.
   a) Find the equation of the regression line of age on weight.
   b)Based on this data, what is the approximate weight of a six year old child?

2. The success of a shopping center can be represented as a function of the distance (in miles) from the center of the population and the number of clients (in hundreds of people) who will visit. The data is given in the table below:

No. Customer (x)	8	7	6	4	2	1
Distance (y)	15	19	25	23	34	40

   a) Calculate the linear correlation coefficient.
   b) If the mall is located 2 miles from the center of the population, how many customers should the shopping center expect?
   c) To receive 500 customers, at what distance from the center of the population should the shopping centre be located?

3. The grades of five students in mathematics and chemistry classes are:

Mathematics	6	4	8	5	3. 5
Chemistry	6. 5	4. 5	7	5	4

Determine the regression lines and calculate the expected grade in chemistry for a student who has a 7.5 in mathematics.

4. A data set has a correlation coefficient of r = −0.9, with the means of marginal distributions of $\bar{X} = 1$ and $\bar{Y} = 2$. It is known that one of the following four equations corresponds to the regression of y on x

$$y = -x + 23x - y = 1\ 2x + y = 4\ y = x + 1$$

Select the correct line.

5. The heights (in centimeters) and weight (in kilograms) of 10 basketball players on a team are:

Height (X)	**186**	**189**	**190**	**192**	**193**	**193**	**198**	**201**	**203**	**205**
Weight (Y)	85	85	86	90	87	91	93	103	100	101

Calculate:

a) The regression line of y on x.
b) The coefficient of correlation.
c) The estimated weight of a player who measures 208 cm.

6. From the following data of hours worked in a factory (x) and output units (y), determine the regression line of y on x, the linear correlation coefficient and determine the type of correlation.

Hours (X)	80	79	83	84	78	60	82	85	79	84	80	62
Production (Y)	300	302	315	330	300	250	300	340	315	330	310	240

7. A group of 50 individuals has been surveyed on the number of hours devoted each day to sleeping and watching TV. The responses are summarized in the following table:

No. of sleeping hours (x)	6	7	8	9	10
No. of hours of television (y)	4	3	3	2	1
Absolute frequencies (f_i)	3	16	20	10	1

a) Calculate the correlation coefficient.
b) Determine the equation of the regression line of y on x.
c) If a person sleeps eight hours, how many hours of TV are they expected to watch?

8. The following table summarizes the results of an aptitude test given to six clerks to determine the correlation between test scores (x) and sales in the first month (y) in hundreds of dollars.

X	25	42	33	54	29	36
Y	42	72	50	90	45	48

a) Find the correlation coefficient and interpret the results.
b) Calculate the regression line of y on x and predict the sales of a vendor who obtains 47 on the test.

9. A company wants to know if there is a significant relationship between its advertising expenditures and its sales volume. The independent variable is advertising budget and the dependent variable is sales volume. A lag time of one month will be used because sales are expected to lag behind actual advertising expenditures. Data was collected for a six month period. All figures are in thousands of dollars. Is there a significant relationship between advertising budget and sales volume?

Independent	4.2	6.1	3.9	5.7	7.3	5.9
Dependent	27.1	30.4	25.0	29.7	40.1	28.8

10. With the growth of internet service providers, a researcher decides to examine whether there is a correlation between cost of internet service per month (rounded to the nearest dollar) and degree of customer satisfaction (on a scale of 1 - 10 with a 1 being not at all satisfied and a 10 being extremely satisfied). The researcher only includes programs with comparable types of services. A sample of the data is provided below. Compute correlation coefficient.

Dollars	11	18	17	15	9	5	12	19	22	25
Satisfaction	6	8	10	4	9	6	3	5	2	10

A handbook on

R-language: A programming software for statistical computing and graphics

---

MCC+ S TISTICS

Department of Statistics
Madras Christian College
Tambaram Chennai -600059