Global Scientific JOURNALS

GSJ: Volume 9, Issue 7, July 2021, Online: ISSN 2320-9186
www.globalscientificjournal.com

MEKDELA AMBA UNIVERSITY

COLLEGE OF NATURAL AND COMPUTATIONAL SCIENCE

DEPARTMENT OF STATISTICS

STATISTICAL COMPUTING- II LAB MANUAL FOR R AND SAS STATISTICAL SOFTWARE'S

BY:

Demsew Beletew (M.Sc., In Biostatistics)

Getachew Abate (B.Sc., In Statistics)

Brtukan Ftayehu (M.Sc., In Biostatistics)

JUNE, 2020

TULU AWULIYA, ETHIPIA

Preface

This manual is intended as a guide to data analysis with the R and SAS software's for statistical computing. R is an environment incorporating an implementation of the S programming language, which is powerful, flexible and has excellent graphical facilities (R Development Core Team, 2005). SAS is a sophisticated computer package containing many components. The capabilities of the entire package extend far beyond the needs of an introductory statistics course. In the manual we aim to give relatively brief and straightforward descriptions of how to conduct a range of statistical analyses using R and SAS. Each chapter deals with the analysis appropriate for one or several data sets. A brief account of the relevant statistical background is included in each session along with appropriate references, but our prime focus is on how to use R and SAS. We hope the Manual will provide students and researchers in many disciplines with a self-contained means of using R and SAS to analyze their data.

Manual Objective

The main objectives of this manual are as follows:

1) Successfully install and run R and SAS on computer
2) Teach enough R and SAS that are easy to do most common data manipulating, analyzing, comparing, and viewing tasks
3) Provide knowledge foundation so that learning more advanced R and SAS techniques is possible
4) Give general tips and suggestions about how to program in R and SAS
5) Illustrate the usefulness of two software's

Once, welcome to R and SAS, and we hope this manual motivates you to use R and SAS in your scientific career!

Acknowledgements

This Manual was inspired at least in part by the excellent statistical computing package R and SAS, and the author would like to acknowledge the team of developers for continuing to support and improve those software. We would like to express my gratitude towards our God who gave our strength and perseverance to complete this task. We would also like to thank the reviewers who made invaluable suggestions and comments. Finally, we would like to acknowledge our staff members for their constant support and encouragement.

Table of Contents

CHAPTER 1

Introduction to R

The focus of this lab is to introduce you to R and the R Commander (a graphical user interface to R). To use R to analyze data, you will need to become familiar with the technical components of this software package. This lab will help familiarize you with the R software, including how to access data files, the various base components and how to define new variables and how to enter data. Since R is open-source, you can freely download it and all of its components on your computer.

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Explain R.
- ✓ Download and install R and different R packages properly into your computer
- ✓ Load a script file, run lines from it, edit and save the script file. Set a working directory.
- ✓ Load a Workspace containing an R data frame, edit the dataset, and save the Workspace.
- ✓ Install an R 'package'; import data from .csv (comma separated value) file format and .xlsx (Excel) file format.
- ✓ Know different data handling and manipulation techniques by using R software

1.1. Defining and Downloading R

The R system for statistical computing is an environment for data analysis and graphics. The root of R is the S language, developed by John Chambers and colleagues (Becker et al., 1988, Chambers and Hastie, 1992, Chambers, 1998) at Bell Laboratories (formerly AT&T, now owned by Lucent Technologies) starting in the 1960s. The S language was designed and developed as a programming language for data analysis tasks but in fact it is a full-featured programming language in its current implementations. The development of the R system for statistical computing is heavily influenced by the open source idea: The base distribution of R and a large number of user contributed extensions are available under the terms of the Free Software Foundation's GNU General Public License in source code form. This license has two major implications for the data analyst working with R. The complete source code is available and thus the practitioner can investigate the details of the implementation of a special method, can make changes and can distribute modifications to colleagues. As a side-effect, the R system for statistical computing is available to everyone. All

scientists, especially including those working in developing countries, have access to state-of-the-art tools for statistical data analysis without additional costs. With the help of the R system for statistical computing, research really becomes reproducible when both the data and the results of all data analysis steps reported in a paper are available to the readers through an R transcript file. R is most widely used for teaching undergraduate and graduate statistics classes at universities all over the world because students can freely use the statistical computing tools. The base distribution of R is maintained by a small group of statisticians, the R Development Core Team. A huge amount of additional functionality is implemented in add-on packages authored and maintained by a large group of volunteers. The main source of information about the R system is the World Wide Web with the official home page of the R project being http://www.R-project.org

All resources are available from this page: The R system itself, a collection of add-on packages, manuals, documentation and more. The intention of this chapter is to give a rather informal introduction to basic concepts and data manipulation techniques for the R novice. Instead of a rigid treatment of the technical background, the most common tasks are illustrated by practical examples and it is our hope that this will enable readers to get started without too many problems.

Installing R

The R system for statistical computing consists of two major parts: the base system and a collection of user contributed add-on packages. The R language is implemented in the base system. Implementations of statistical and graphical procedures are separated from the base system and are organized in the form of packages. A package is a collection of functions, examples and documentation. The functionality of a package is often focused on a special statistical methodology. Both the base system and packages are distributed via the Comprehensive R Archive Network (CRAN) accessible under http://CRAN.R-project.org

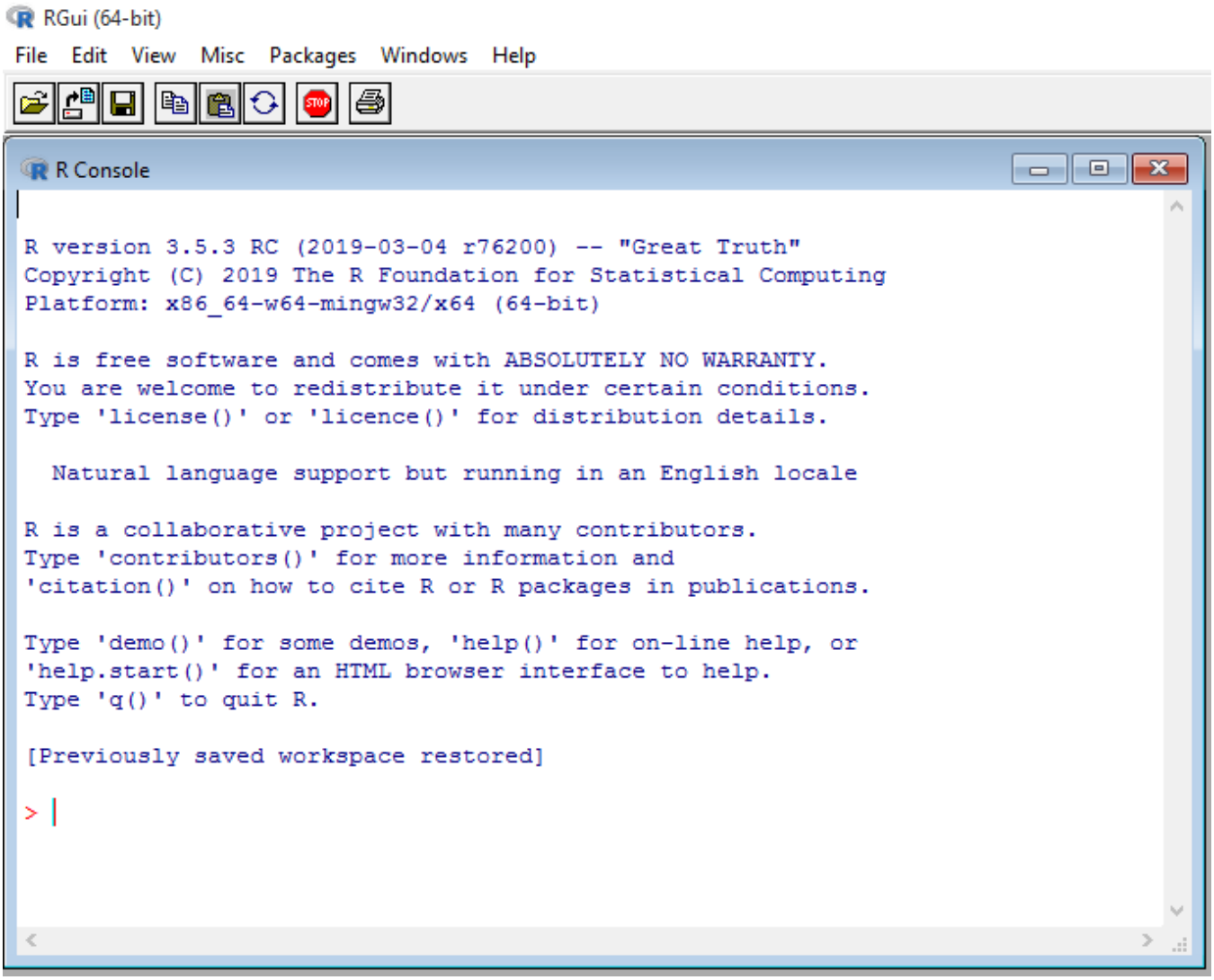
The Base System and the First Steps

Thebasesystemisavailableinsourceformandinprecompiledformforvarious Unix systems, Windows platforms and Mac OS X. For the data analyst, it is sufficient to download the precompiled binary distribution and install it locally. Windows users follow the link

http://CRAN.R-project.org/bin/windows/base/release.htm

Download the corresponding file (currently named rw2040.exe), execute it locally and follow the instructions given by the installer.

Depending on the operating system, R can be started either by typing 'R' on the shell (UNIX systems) or by clicking on the R symbol (as shown above) created by the installer (Windows). R comes without any frills and on startup shows simply a short introductory message including the version number and a prompt '>':



One can change the appearance of the prompt by > options (prompt = "R> ") and we will use the prompt R> for the display of the code examples throughout this manual. Essentially, the R system evaluates commands typed on the R prompt and returns the results of the computations. The end of a command is indicated by the return key. Virtually all introductory texts on R start with an example using R as pocket calculator, and so do we:

R> x <- sqrt (25) + 2 this simple statement asks the R interpreter to calculate$\sqrt{25}$ and then to add 2. The result of the operation is assigned to an R object with variable name x. The assignment operator <- binds the value of its right hand side to a variable name on the left hand side. The value of the object x can be inspected simply by typing

R> x

[1] 7

Which, implicitly, calls the print method:

R> print(x)

[1] 7

Packages

The base distribution already comes with some high-priority add-on packages, namely Kern Smooth, MASS, boot, class, cluster, foreign, lattice, mgcv, nlme, nnet, rpart, spatial, survival, base, datasets, grDevices, graphics, grid, methods, splines, Stats, stats4, tcltk, tools and utils.

The packages listed here implement standard statistical functionality, for example linear models, classical tests, a huge collection of high-level plotting functions or tools for survival analysis; many of these will be described and used in later session. Packages not included in the base distribution can be installed directly from the R prompt. Given that an Internet connection is available, a package is installed by supplying the name of the package to the function **install. Packages**. If, for example, add-on functionality for robust estimation of covariance matrices via **sandwich** estimators is required, the sandwich package (Zeileis, 2004) can be downloaded and installed via

R> install. Packages ("sandwich")

The package functionality is available after attaching the package by

R> library("sandwich")

A comprehensive list of available packages can be obtained from

http://CRAN.R-project.org/src/contrib/PACKAGES.html

Note that on Windows operating systems, precompiled versions of packages are downloaded and installed. In contrast, packages are compiled locally before they are installed on UNIX systems.

1.2. Getting online help in R

Online help that comes with the base distribution or packages, electronic manuals and publications work in the form of books etc. The help system is a collection of manual pages describing each user-visible function and data set that comes with R. A manual page is shown in a pager or web

browser when the name of the function we would like to get help for is supplied to the help function

R> help("mean")

or, for short,

R>? mean

- Help is also available in HTML format by running > help. Start ()
- Which will launch a web browser that allows the help pages to be browsed with hyperlinks.
- The help. Search command (alternatively??) allows searching for help in various ways. Example: >?? Solve #Or

  > help.search("solve")
- Try? help.search for details and more examples.

1.3. Commands and Executions

Technically R is an expression language with a very simple syntax. When R is ready for input, it prints out its prompt (default) greater than sign (>); it is an assignment operator. Command lines entered at the console are limited to about 4095 bytes (not characters).

If commands are stored in an external file, say commands .R in the working directory work, they may be executed at any time in an R session with the command

> source("commands .R")

For Windows Source is also available on the File menu. The function sink,

> sink ("record .lis")

Will divert all subsequent output from the console to an external file, record.lis. The command

> sink () restores it to the console once again.

Removing objects from R:

- Use rm("object") or remove("object")

  > Age<-45

  > rm(Age) #or

  > remove(Age)
- The set of symbols which can be used in R: Can be created using letters, digits and the. (dot) symbol. e.g. Weight, Wt. male
- Must not start with a digit or a. Followed by a digit or vice versa. Some names are used by the system. e.g. c, q, t, C, D, F, I, T, diff, df, pt, Don't use those as an object.

1.4. Objects and simple manipulation in R

1.4.1. Data Objects in R

The data handling and manipulation techniques explained in this session will be illustrated by means of a data set of 2000 world leading companies, the Forbes 2000 list for the year 2004 collected by 'Forbes Magazine'. This list is originally available from

http://www.forbes.com

And, as an R data object, it is part of the *HSAUR package* (Source: From Forbes.com, New York, New York, 2004. With permission.). In a first step, we make the data available for computations with in R. The data function searches for data objects of the specified name ("Forbes2000") in the package specified via the **package** argument and, if the search was successful, attaches the data object to the global environment:

```
R> data ("Forbes2000", package = "HSAUR")
R> ls ()
[1] "Forbes2000" "a" "x"
```

The output of the ls function lists the names of all objects currently stored in the global environment, and, as the result of the previous command, a variable named Forbes2000 is available for further manipulation. The variable x arises from the pocket calculator. As one can imagine, printing a list of 2000 companies via

```
R> print(Forbes2000)
```

1.4.2. Vectors and assignment

R operates on named data structures. A vector represents a set of elements of the same mode whether they are logical, numeric (integer or double), complex, character or lists. The simplest such structure is the numeric vector, which is a single entity consisting of an ordered collection of numbers. To set up a vector named x, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

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

This is an assignment statement using the function c () which in this context can take an arbitrary number of vector arguments and whose value is a vector got by concatenating its arguments end to end. A number occurring by itself in an expression is taken as a vector of length one. Notice that the assignment operator ('<-'), which consists of the two characters '<' ("less than") and '-' ("minus") occurring strictly side-by-side and it 'points' to the object receiving the value of the

expression. In most contexts the '=' operator can be used as an alternative. Assignment can also be made using the function assign (). An equivalent way of making the same assignment as above is with:

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

The usual operator, <-, can be thought of as a syntactic short-cut to this. Assignments can also be made in the other direction, using the obvious change in the assignment operator. So the same assignment could be made using

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

If an expression is used as a complete command, the value is printed and lost. So now if we were to use the command

> 1/x

The reciprocals of the five values would be printed at the terminal (and the value of x, of course, unchanged). The further assignment

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

Would create a vector y with 11 entries consisting of two copies of x with a zero in the middle place.

Functions that return a single value

```
>length(x)      # the number of elements in x
> sum(x)        # the sum of the values of x
> mean(x)       # the mean of the values of x
> var(x)        # the variance of the values of x
> sd(x)         # the standard deviation of the values of x
> min(x)        # the minimum value from the values of x
> max(x)        # the maximum value from the values of x
> prod(x)       # the product of the values of x
> range(x)      # the range of the values of x (smallest and largest)
```

Character vectors

To set up a character/string vector y consisting of 4 fruit names use:

>y= c ("banana", "orange", "mango", "lemon")

Character strings are entered using either matching double (" ") or single (' ') quotes, but are printed using double quotes (or sometimes without quotes). They use C-style escape sequences, using \ as the escape character, so \\ is entered and printed as \\, and inside double quotes “is entered as \”.

Common useful escape sequences are:

"\n" for new line

"\t" for tab

"\b" for backspace

Example:

```
> cat("Abebe","\n","zelalem","\n")
```

Character vectors can be concatenated using c ()

```
> y= c ("banana", "orange", "mango", "lemon")
> c(y,"tomato")
[1] "banana" "orange" "mango” “lemon” “tomato"
```

Logical Vectors

A logical vector is a vector whose elements are TRUE, FALSE or NA (not available).

Note TRUE and FALSE are often abbreviated as T and F respectively, however T and F are just variables which are set to TRUE and FALSE by default, but are not reserved words and hence can be overwritten by the user, Logical vectors are generated by conditions.

Example:

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

Sets temp as a vector of the same length as x with values FALSE corresponding to elements of x where the condition is not met and TRUE where it is met.

The logical operators are <, <=, >, >=, == for exact equality and! = for inequality. In addition, if c1 and c2 are logical expressions, then c1&c2 is their intersection (“and”), c1 I c2 is their union (“or”), and! c1 is the negation of c1.

Missing values

When an element or value is “not available” or a “missing value” in the statistical sense, a place within a vector may be reserved for it by assigning it the special value NA. an operation is incomplete, the result cannot be known and not available. The function is.na(x) gives a logical vector of the same size as x with value TRUE if and only if the corresponding element in x is NA(where NA is value not available or a missing value).

Generating regular sequences

R has a number of facilities for generating commonly used sequences of numbers. For example 1:30 is the vector c(1, 2, ..., 29, 30). The colon operator has high priority within an expression, so, for example 2*1:15 is the vector c(2, 4, ..., 28, 30). Put n <- 10 and compare the sequences 1:n-1 and 1:(n-1).

The construction 30:1 may be used to generate a sequence backwards.

The function seq() is a more general facility for generating sequences. It has five arguments, only some of which may be specified in any one call. The first two arguments, if given, specify the beginning and end of the sequence, and if these are the only two arguments given the result is the same as the colon operator. That is seq(2,10) is the same vector as 2:10. Arguments to seq(), and to many other R functions, can also be given in named form, in which case the order in which they appear is irrelevant. The first two arguments may be named from=value and to=value; thus seq(1,30), seq(from=1, to=30) and seq(to=30, from=1) are all the same as 1:30. The next two arguments to seq() may be named by=value and length=value, which specify a step size and a length for the sequence respectively. If neither of these is given, the default by=1 is assumed.

For example

```
> seq(-5, 5, by=.2) -> s3
```

generates in s3 the vector c(-5.0, -4.8, -4.6, ..., 4.6, 4.8, 5.0). Similarly

```
> s4 <- seq(length=51, from=-5, by=.2)
```

Generates the same vector in s4.

The fifth argument may be named along=vector, which is normally used as the only argument to create the sequence 1, 2, ..., length(vector), or the empty sequence if the vector is empty (as it can be).

A related function is rep() which can be used for replicating an object in various complicated ways. The simplest form is

```
> s5 <- rep(x, times=5)
```

Which will put five copies of x end-to-end in s5. Another useful version is

```
> s6 <- rep(x, each=5)
```

Which repeats each element of x five times before moving on to the next.

1.4.3. Arrays, matrices, Lists and Data frame

Arrays

An array can be considered as a multiply subscripted collection of data entries, for example numeric. R allows simple facilities for creating and handling arrays, and in particular the special case of matrices.

A dimension vector is a vector of non-negative integers. If its length is k then the array is k-dimensional, e.g. a matrix is a 2-dimensional array. The dimensions are indexed from one up to the values given in the dimension vector.

A vector can be used by R as an array only if it has a dimension vector as its dim attribute. Suppose, for example, z is a vector of 1500 elements. The assignment

> dim(z) <- c (3,5,100)

Gives it the dim attribute that allows it to be treated as a 3 by 5 by 100 array.

Other functions such as matrix () and array () are available for simpler and more natural looking assignments. The values in the data vector give the values in the array in the same order as they would occur in FORTRAN, that is "column major order," with the first subscript moving fastest and the last subscript slowest. For example if the dimension vector for an array, say a, is c (3,4,2) then there are 3×4× 2 = 24 entries in a and the data vector holds them in the order a[1,1,1], a[2,1,1], ..., a[2,4,2], a[3,4,2].

As well as giving a vector structure a dim attribute, arrays can be constructed from vectors by the array function, which has the form

> Z<- array (data_vector, dim_vector)

For example, if the vector h contains 24 or fewer, numbers then the command

>Z<- array (h, dim=c(3,4,2))

Would use h to set up 3 by 4 by 2 array in Z. If the size of h is exactly 24 the result is the same as

> Z<- h; dim(Z) <- c(3,4,2)

Arrays can be one-dimensional: such arrays are usually treated in the same way as vectors (including when printing), but the exceptions can cause confusion.

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 two ways.

Method 1: Using the function dim:

Example

```
>x <- c (1:8)
>dim(x) <- c (2,4)
 > x          [,1]  [,2]  [,3]  [,4]
      [1,]    1     3     5     7
      [2,]    2     4     6     8
```

Method 2: Using the function matrix:

```
 > x <- matrix(c(1:8),2,4, byrow=F)
 > x
      [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
```

An equivalent expression:

```
> x<-matrix(c(1:8),nrow=2,ncol=4)
```

- By default, the matrix is filled by column. To fill the matrix by row, specify byrow = T as argument in the matrix function.
- Use the function cbind to create a matrix by binding two or more vectors as column vectors. The function rbind is used to create a matrix by binding two or more vectors as row vectors.

Example

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

- Matrix operations (multiplication, transpose, etc.) can easily be performed in R using a few simple functions like:

Name	**Operation**
dim()	Dimension of the matrix (number of rows and columns)
as.matrix()	Used to coerce an argument into a matrix object
%*%	Matrix multiplication
t()	Matrix transpose
det()	Determinant of a square matrix
solve()	Matrix inverse; also solves a system of linear equations
eigen()	Computes eigenvalues and eigenvectors

Lists

An R list is an object consisting of an ordered collection of objects known as its components. There is no particular need for the components to be of the same mode or type, and, for example, a list could consist of a numeric vector, a logical value, a matrix, a complex vector, a character array, a function, and so on. Lists are created with the list () command:

Lst<-list (object-1, object-2, …, object-m)

Here is a simple example of how to make a list:

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

Components are always numbered and may always be referred to as such. Thus if Lst is the name of a list with four components, these may be individually referred to as Lst[[1]], Lst[[2]], Lst[[3]] and Lst[[4]]. If, further, Lst[[4]] is a vector subscripted array then Lst[[4]][1] is its first entry.

If Lst is a list, then the function length(Lst) gives the number of (top level) components it has.

Components of lists may also be named, and in this case the component may be referred to either by giving the component name as a character string in place of the number in double square brackets, or, more conveniently, by giving an expression of the form

> name$component_name

for the same thing.

This is a very useful convention as it makes it easier to get the right component if you forget the number.

So in the simple example given above:

Lst$name is the same as Lst[[1]] and is the string "Fred",

Lst$wife is the same as Lst[[2]] and is the string "Mary",

Lst$child.ages[1] is the same as Lst[[4]][1] and is the number 4.

Additionally, one can also use the names of the list components in double square brackets, i.e., Lst[["name"]] is the same as Lst$name. This is especially useful, when the name of the component to be extracted is stored in another variable as in

> x <- "name"; Lst[[x]]

It is very important to distinguish Lst[[1]] from Lst[1]. '[[...]]' is the operator used to select a single element, whereas '[...]' is a general subscripting operator. Thus the former is the first object in the list Lst, and if it is a named list the name is not included. The latter is a sublist of the list Lst consisting of the first entry only. If it is a named list, the names are transferred to the sublist.

The names of components may be abbreviated down to the minimum number of letters needed to identify them uniquely. Thus Lst$coefficients may be minimally specified as Lst$coe and Lst$covariance as Lst$cov.

The vector of names is in fact simply an attribute of the list like any other and may be handled as such. Other structures besides lists may, of course, similarly be given a names attribute also.

Data frame

Data frames: regarded as an extension to matrices, can have columns of different data types and are the most convenient data structure for data analysis in R. Data frames are lists with the constraint that all elements are vectors of the same length.

The command data.frame() creates a data frame:

dat<-data.frame(object-1,object-2,…,object-m)

Example:

```
>name= c("Eden","Solomon","Zelalem","Kidist")
>age=c(18,22,25,27)
> sex=c("F","M","M","F")
 > stud=data.frame(name,age,sex)
 >stud
      name       age   sex
1    Eden       18     F
2   Solomon   22     M
```

```
3   Zelalem    25    M
4   kidist     27    F
```

- To display the column names:

```
> names(stud)   #OR  colnames(stud)      [1] "name" "age"  "sex“
```

- To display the row names:

```
> rownames(stud)  [1] "1" "2" "3" "4"
```

- You can use the “names” function to change the column names:

```
> names(stud)<-c("name","age","sex")
> stud
     name      age  sex
1    Eden       18   F
2    Solomon    22   M
3    Zelalem    25   M
4    kidist     27   F
```

- Similarly, use the “row.names” function to change the row names:

```
> row.names(stud)<-c("Wrt","Ato","Ato2","Wro")
> stud
```

- **Note**: Duplicate row.names are not allowed!!!

1.4.4. Reading (Importing) Data

Large data objects will usually be read as values from external files rather than entered during an R session at the keyboard. R input facilities are simple and their requirements are fairly strict and even rather inflexible. There is a clear presumption by the designers of R that you will be able to modify your input files using other tools, such as file editors or Perl1 to fit in with the requirements of R. Generally, this is very simple. If variables are to be held mainly in data frames, as we strongly suggest they should be, an entire data frame can be read directly with the read.table() function. There is also a more primitive input function, scan(), that can be called directly. For more details on importing data into R and also exporting data, see the R Data Import/Export manual. In R you can import text files with the function read.table.

Syntax: read.table(“file name", arguments)

The function may have arguements to specify the header, the column separator, the number of lines to skip, the data types of the columns, etc.

The functions read.csv and read.delim are functions to read "comma separated values" files and tab delimited files.

Suppose we have a text le data.txt that contains the following text:

Author: John Davis

Date: 18-05-2007

Some comments..

Col1, Col2, Col3, Col4

23, 45, A, John

34, 41, B, Jimmy

12, 99, B, Patrick

- The data without the first few lines of text can be imported to an R data frame using the following R syntax:

```
myfile <- "C:\\Temp\\R\\Data.txt"
mydf <- read.table(myfile, skip=3, sep=",", header=TRUE)
mydf
```

Existing Data in R

- A number of datasets are supplied with R (in package datasets)
- To see the list of datasets currently available use the following command:

```
>data()
```

- Datasets usually store several variables. The different variables have names, but initially, these names are not directly accessible as variables.
- It can be accessed the values by name or we can use the attach statement.

```
>attach(filename)   #Or

> filename$varname
```

- Editing data in R with following R command

```
> NewFileName<-edit(OldFileName)
```

Lab Exercise 1

To do the entire exercise, read the given data into R properly based on the description of the data.

1. Read the following numbers in to R-editor window
    a) X=2*16 and print x
    b) Z=19+65 and print z
    c) W=x+z and print w
    d) Save the numbers in R-new script window in drive D?
2. Read the following numbers in to R-console window?
    a) Y=√896 and print y
    b) W= 35/7 and print w
    c) Save your workspace in user created folder named "stat2" in drive D?
3. Read the following numbers in to R and perform manipulation
    a) 896! B) 4^3 c) 896 C 50 d) e^{89} e) log(896)
4. Create a vector x and z with value of 20, 21, 34, 56, 23 and m, m, p, p, and o respectively?
5. Create a vector W consisting a values from
    a) 1 to 10
    b) 102, 104, 106, 108, 110, 112
    c) Repeating 1,2,3 and 4 three times
    d) Repeating number two (2) ten times
6. Read M1 3x3 matrix into R from the following vector and answer the following matrix operations?

    X=c (15, 14, -13, 13, 17, 12, 12, 10, 18)
    a) Determinant of M1
    b) Dimensions of M1
    c) Transpose of M1
    d) Inverse of M1 and its multiplication with M1
    e) Eigen value and Eigen vector of M1
    f) Display the element of M1 in third row and second column
7. Consider the following data and create data frame named student data in R

CGPA	2.4	2.6	2.83	3.6	3.25	2.75
Study hour/week	5	4	6	8	7	3
Sex	M	M	F	F	M	M

8. Enter the following data on notepad and save it as text file and then Read into R

Age	26	56	76	55	65
SBP	121	115	125	122	130
Weight	55	65	70	60	75

CHAPTER 2

Probability and sampling distributions

In this chapter we briefly review without proofs some definitions and concepts in probability and statistics. Many introductory and more advanced texts can be recommended for review and reference.

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Provide a comprehensive set of statistical tables by using R
- ✓ Examine the distribution of a set of data
- ✓ Calculate the probability of any distribution by R
- ✓ Generate a sample by using R software

2.1. R as a set of statistical tables

The R suite of programs provides a simple way for statistical tables of just about any probability distribution of interest. R allows for the calculation of;

- ✓ Probabilities (including cumulative)
- ✓ The evaluation of probability density/mass functions
- ✓ Percentiles, and
- ✓ The generation of pseudo-random variables following a number of common distributions.

- ➢ Therefore, R is useful to provide a comprehensive set of statistical tables.
- ➢ The following table gives examples of various function names in R along with additional arguments.

Distribution	R name	arguments
Normal	norm	mean, sd
Chi-squared	chisq	df, ncp
F	f	df1, df2, ncp
Student's t	t	df, ncp
Log-normal	lnorm	meanlog, sdlog
Logistic	logis	location, scale
Poisson	pois	lambda
Binomial	binom	size, prob

For each distribution, R provides the following four commands;

dxxx: density function of the xxx distribution

pxxx: distribution function of the xxx distribution ('p' for probability)

qxxx: quintile function of the xxx distribution

rxxx: random number generator for the xxx distribution where 'xxx' is the R name of the distribution.

The pxxx and qxxx functions all have logical arguments lower.tail and log.p and the dxxx ones have log. This allows, e.g., getting the cumulative (or "integrated") hazard function, H(t) = −log(1−F(t)), by - pxxx(t, ..., lower.tail = FALSE, log.p = TRUE) or more accurate log-likelihoods (by dxxx(..., log = TRUE)), directly. In addition, there are functions ptukey and qtukey for the distribution of the studentized range of samples from a normal distribution, and dmultinom and rmultinom for the multinomial distribution. Further distributions are available in contributed packages, notably SuppDists (https://CRAN.R-project.org/package=SuppDists). Here are some examples

```
> ## 2-tailed p-value for t distribution
> 2*pt(-2.43, df = 13)
> ## upper 1% point for an F(2, 7) distribution
> qf(0.01, 2, 7, lower.tail = FALSE)
```


See the on-line help on RNG for how random-number generation is done in R.

Example:

```
>dbinom(3,size=10,prob=.25)     # P(X=3) for X ~ Bin(n=10, p=.25)

> dpois(0:2, lambda=4)          # P(X=0), P(X=1), P(X=2) for X ~ Poisson(4)

> pbinom(3,size=10,prob=.25)    # P(X < 3) in the above distribution

> pnorm(12,mean=10,sd=2)        # P(X < 12) for X~N(mu = 10, sigma =2)

> qnorm(.75,mean=10,sd=2)       # 3rd quartile of N(mu = 10,sigma = 2)

> qchisq(.10,df=8)              # 10th percentile of x2(8)

> qt(.95,df=20)                 # 95th percentile of t(20)

>rnorm(100)                     # simulate(generate) 100 standard normal RVs
```

```
> 2*pt(-2.43, df = 13)              # 2-tailed p-value for t distribution

> qf(0.01, 2, 7, lower.tail = FALSE)      # upper 1% point for an F(2, 7) distribution
```

R as sampler

In R, we can simulate different situations with the sample () function. e.g if we want to pick five numbers at random from the set 1:40, then we can write Sample(1:40,5). The default behavior of sample is sampling without replacement. The sample will not contain the same number twice, If we want sampling with replacement, then we need to add the argument, replace =T sample(c("H", "T"), 10, replace =T). We can also generate random numbers from normal distribution by specifying the value of mean and standard deviation.

```
>rnorm(10, mean=7, sd=5)  # this can generate 10 random numbers with mean of 7 and standard deviation 5
```

2.2. Examining the distribution of a set of data

Given a (univariate) set of data we can examine its distribution in a large number of ways. The simplest is to examine the numbers. Two slightly different summaries are given by **summary** and **fivenum**.


```
>summary(var)

>fivenum(var)
```

And also we can examine by displaying the data through the following graphs:

```
>stem(var)              # Steam and leaf of the var.
>hist(var)              #Default histogram of var.
>boxplot(var)           # a box plot of var
>plot(ecdf(var))        #the empirical cumulative  distribution function of var.
>x <- rt(250, df = 5)  # A random sample of size 250 from t distribution with 5 df
> qqnorm(var)          # QQ plot  for normality of var.
> qqline(var)          #make a line on the above QQ plot
```

We can make a Q-Q plot against the generating distribution by

```
>qqplot(qt(ppoints(250), df = 5), x, xlab ="Q-Q plot for t dsn")
>qqline(x)
```

Formally, R provides the Shapiro-Wilk test and the KolmogorovSmirnov test to examine whether the given data follows a normal distribution or not.

```
>shapiro.test(var)   #Shapir-Wilk test
>ks.test(var, "pnorm", mean = mean(var),  sd = sqrt(var(var)))       #kolmogorov-smirnov test
```

2.3. One- and two-sample t-tests

The main function that performs these sorts of tests is t.test().

Its syntax is:

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

Arguments:

- **x**, **y:** numeric vectors of data values. If y is not given, a one sample test is performed.
- **alternative**: a character string specifying the alternative hypothesis, must be one of `"two.sided"' (default), "greater" or "less". You can specify just the initial letter.
- **mu**: a number indicating the true value of the mean (or difference in means if you are performing a two sample test). Default is 0.
- **paired**: a logical indicating if you want the paired t-test (default is the independent samples test if both x and y are given).
- **var.equal**: (for the independent samples test) a logical variable indicating whether to treat the two variances as being equal. If `TRUE', then the pooled variance is used to estimate the variance. If 'FALSE' (default), then the Welsh suggestion for degrees of freedom is used.
- **conf.level**: confidence level (default is 95%) of the interval estimate for the mean appropriate to the specified alternative hypothesis.

Note that from the above, t.test () not only performs the hypothesis test but also calculates a confidence interval. However, if the alternative is either a "greater than" or "less than" hypothesis, a lower (in case of a greater than alternative) or upper (less than) confidence bound is given.

Example: Test the hypotheses that the average height content of containers of certain lubricant is 10 liters if the contents of a random sample of 10 containers are 10.2, 9.7, 10.1, 10.3, 10.1, 9.8, 9.9, 10.4, 10.3, and 9.8 liters. Use the 0.01 level of significance and assume that the distribution of contents is normal.

```
>x=c(10.2,9.7,10.1,10.3,10.1,9.8,9.9,10.4, 10.3,9.8)

>t.test(x, mu = 10, conf.level = 0.99)
```

- The output of the above command will be:

```
One Sample t-test

data:  x

t = 0.7717, df = 9, p-value = 0.46

alternative hypothesis: true mean is not equal to 10

99 percent confidence interval:

 9.807338  10.312662

sample estimates:

 mean of x

 10.06
```


Example: Consider the following sets of data on the latent heat of the fusion of ice (cal/gm) from Rice.

Method A: 79.98 80.04 80.02 80.04 80.03 80.03 80.04 79.97 80.05 80.03 80.02 80.00 80.02

Method B: 80.02 79.94 79.98 79.97 79.97 80.03 79.95 79.97

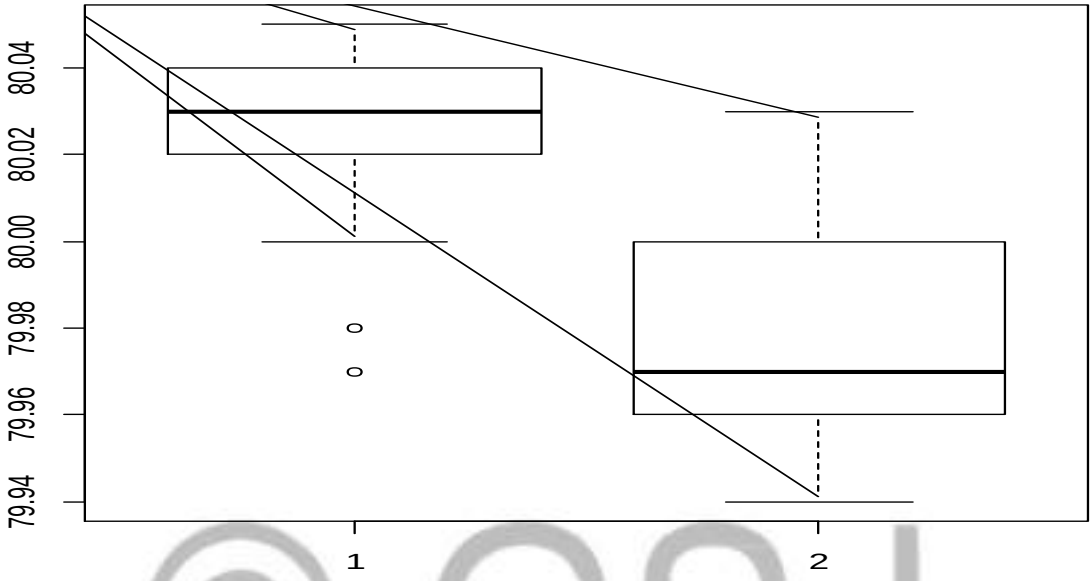
- Box plots provide a simple graphical comparison of the two samples.

```
>A=c(79.98,80.04,80.02,80.04,80.03,80.03,80.04,79.97,80.05,80.03,80.02, 80.00,80.02)

>B=c(80.02,79.94,79.98,79.97,79.97,80.03,79 .95,79.97)
```

>boxplot(A,B)

- which indicates that the first group tends to give higher results than the second.



- To test for the equality of the means of the two samples, we can use an unpaired t-test by:

> t.test (A, B)

- This will give you the following output:

Welch Two Sample t-test

data: A and B t = 3.2499, df = 12.027, p-value = 0.006939 alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval: 0.01385526 0.07018320

sample estimates: mean of x mean of y

80.02077 79.97875

Which indicate a significant difference, assuming normality. By default, the R function does not assume equality of variances in the two samples.

- We can use the F test to test for equality in the variances, provided that the two samples are from normal populations.

```
> var.test(A, B)

F test to compare two variances

data:  A and B

F = 0.5837, num df = 12, denom df = 7, p-value = 0.3938

alternative hypothesis: true ratio of variances is not equal to 1

95 percent confidence interval:  0.1251097  2.1052687

sample estimates:  ratio of variances  0.5837405
```

Which shows no evidence of a significant difference, and so we can use the classical t-test that assumes equality of the variances.


```
>t.test(A, B, var.equal=TRUE)

Two Sample t-test

data:  A and B

t = 3.4722, df = 19, p-value = 0.002551

alternative hypothesis: true difference in means is not equal to 0

95 percent confidence interval:  0.01669058  0.06734788

sample estimates:

mean of x  mean of y

80.02077  79.97875
```

Example: the recovery time (in days) is measured for 10 patients taking a new drug and for 10 different patients taking a placebo. We wish to test the hypothesis that the mean recovery time for patients taking the drug is less than fort those taking a placebo (under an assumption of normality and equal population variances). The data are:

With drug: 15, 10, 13, 7, 9, 8, 21, 9, 14, 8

Placebo: 15, 14, 12, 8, 14, 7, 16, 10, 15, 12

Answer

```
> drug <- c(15, 10, 13, 7, 9, 8, 21, 9, 14, 8)

> plac <- c(15, 14, 12, 8, 14, 7, 16, 10, 15, 12)

> t.test(drug, plac, alternative ="less", var.equal = T)

Two Sample t-test

data:  drug and plac

t = -0.5331, df = 18, p-value = 0.3002

alternative hypothesis: true difference in means is less than 0

95 percent confidence interval:

    -Inf 2.027436

sample estimates:

mean of x mean of y

    11.4      12.3
```


Example: an experiment was performed to determine if a new gasoline additive can increase the gas mileage of cars. In the experiment, six cars are selected and driven with and without the additive. The gas mileages (in miles per gallon, mpg) are given below.

Car	1	2	3	4	5	6
mpg w/ additive:	24.6	18.9	27.3	25.2	22.0	30.9
mpg w/o additive:	23.8	17.7	26.6	25.1	21.6	29.6

Since this is a paired design, we can test the claim using the paired t–test (under an assumption of normality for mpg measurements). This is performed by:

```
>add <-c(24.6, 18.9, 27.3, 25.2, 22.0, 30.9)

>noadd <-c(23.8, 17.7, 26.6, 25.1, 21.6, 29.6)
```

```
>t.test(add, noadd, paired=T, alt = "greater")

        Paired t-test

data:  add and noadd

t = 3.9994, df = 5, p-value = 0.005165

alternative hypothesis: true difference in means is greater than 0

95 percent confidence interval:

 0.3721225       Inf

sample estimates:

mean of the differences

              0.75
```

Lab Exercise 2

1. Generate a random data from binomial probability distribution sample size 5 with number of trial 10 and probability of success 0.5?

2.Suppose that a marks of students follows normal distribution with mean 80 and standard deviation of 4 then what is the probability that marks of randomly selected students to be

   a) Less than 70
   b) Greater than 70

3. Generate 2000 random sample from normal distribution with mean 20 and standard deviation 4, compute mean and standard deviation of random sample, are they close to theoretical values?
4. The exponential (rate=β) is a special case gamma i.e when shape parameter (α=1) that is gamma (α=1, β) is exponential (β). Verify that gamma (α=1, β) is similar to exponential (β) by generating random sample from both distribution? *(hint to verify the theorem of distribution use β=50)*
5. Suppose electric light bulbs produced at Company-Z have probability 0.005 of being defective. Suppose electric light bulbs are shipped in cartons containing 50, what is the probability that randomly chosen carton contains no more than three defective electric light bulbs?

CHAPTER 3

Writing your own functions

R is an expression language in the sense that it's only command type is a function or expression which returns a result. Even an assignment is an expression whose result is the value assigned, and it may be used wherever any expression may be used; in particular multiple assignments are possible.

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Define a function that takes arguments.
- ✓ Return a value from a function.
- ✓ Test a function.
- ✓ Set default values for function arguments.
- ✓ Explain why we should divide programs into small, single-purpose functions.

As we have seen informally along the way, the R language allows the user to create objects of mode function. These are true R functions that are stored in a special internal form and may be used in further expressions and so on. In the process, the language gains enormously in power, convenience and elegance, and learning to write useful functions is one of the main ways to make your use of R comfortable and productive. The basic template for a function is

```
function_name <- function (function_arguments)
{
function_body (statments)
function_return_value (answer)
}
```

Each of these is important. Let's cover them in the order they appear, the function name, can be just about anything {even functions or variables previously defined so be careful. Once you have given the name, you can use it just like any other function {with parentheses.

For example, to define a standard deviation function using the var function we can do

```
std <- function (x) sqrt(var(x))
```

This has the name std. It is used thusly

```
data <- c(1,3,2,4,1,4,6)
```

std(data)

If you call it without parentheses you will get the function definition itself

std function (x) sqrt(var(x))

Conditional execution: if(), if()else and ifelse()

Syntax:

if(condition) {commands1}

if(condition) {commands1} else {commands2}

ifelse (conditions vector, yes vector, no vector)

- The command if() evaluates 'commands1' if the logical expression 'condition' returns TRUE. Here 'commands1' is a single command or a sequence of commands separated with ';'.
- The command if() else evaluates 'commands1' if the logical expression 'condition' returns TRUE, otherwise it evaluates 'commands2'.
- The command ifelse() returns a vector of the same length as 'conditions vector' with elements selected from either 'yes vector' or 'no vector' depending on whether the element of 'conditions vector' is TRUE or FALSE.

Example:

> x <- 4

> if (x == 5) { x <- x+1 } else { x <- x*2 }

> x [1] 8

- > if (x != 5 & x>3) { x <- x+1 ; 17+2 } else { x <- x*2 ; 21+5 }

[1] 19

> x [1]9

- > y <- 1:10

> ifelse (y<6, y^2, y-1)

[1] 1 4 9 16 25 5 6 7 8 9

- > z <- 6:-3

> sqrt(z) # Produces a warning

[1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000 NaN

[9] NaN NaN

Warning message:

In sqrt(z) : NaNs produced

- sqrt(ifelse(z>=0,z, NA)) # No warning

[1] 2.449490 2.236068 2.000000 1.732051 1.414214 1.000000 0.000000 NA

[9] NA NA

Example: Adding two vectors in R of different length will cause R to recycle the shorter vector. The following adds the two vectors by chopping of the longer vector so that it has the same length as the shorter

> x=1:10; y=2:8

> n1 <- length(x)

> n2 <- length(y)

> if(n1 > n2){ z <- x[1:n2] + y} else{z <- x + y[1:n1]}

> z

[1] 3 5 7 9 11 13 15

Commands may be grouped together in braces, {expr_1; ...; expr_m}, in which case the value of the group is the result of the last expression in the group evaluated. Since such a group is also an expression it may, for example, be itself included in parentheses and used a part of an even larger expression, and so on.

Loops: for(), while() and repeat()

Syntax:

for (var in set) {commands}

while (condition) {commands}

repeat {commands}

Where

- ✓ the object set is a vector,
- ✓ commands is a single command or a sequence of commands and
- ✓ var is a variable which may be used in commands.

- The command for() is the R version of **'for each element in the set do ...'.**
- The command while() is the R version of **'as long as the condition is TRUE do ...'.**
- The command repeat() is the R version of **'repeat until I say break**'. The command 'break' stops any loop; control is then transferred to the first statement outside the loop. The command 'next' halts the processing of the current iteration and advances the looping index.

Loops: for ()

Example:

- >x=c(20,22,27,38,42)
  >summ=0
  >for(i in 1:length(x)){summ=summ + x[i]}
  >summ
  [1] 149
- >x <- 0
  > for (i in 1:5) {if (i==3) {next}; x <- x + i}
  > x # i=3 is skipped, so x <- 1+2+4+5
  [1] 12

Loops: while () and repeat ()

- > y <- 1; j <- 1
  > while (y < 12 & j < 8) {y <- y*2 ; j <- j + 1}

```
> y; j
[1] 16
 [1] 5
```

- ```
  > z <- 3
  > repeat { z<- z^2; if ( z>100 ) { break }}
  > z # the loop stopped after 81^2, so  z==81^2
   [1]  6561
  ```

- Probably one of the most powerful aspects of the R language is the ability of a user to write functions.
- When doing so, many computations can be incorporated into a single function and (unlike with scripts) intermediate variables used during the computation are local to the function and are not saved to the workspace.
- In addition, functions allow for input values used during a computation that can be changed when the function is executed.

The general format for creating a function is

```
fname<-function(arg1, arg2, ...) { R code }
```

Where fname is any allowable object name and arg1, arg2, ... are function arguments.

- As with any R function, they can be assigned default values.
- When you write a function, it is saved in your workspace as a function object.
- Use the command return() for returning a value. If you need to return several values put them into a list and return the list.

Example:

Consider a function that returns the maximum of two scalars or the statement that they are equal.

```
> f1 <- function (a, b) { if(is.numeric(c(a,b))) { if(a < b)  return(b)  else if(a > b)  return(a)
else print("The values are equal") } else print("Character inputs not allowed.") }
```

- Consider a function that computes the zero(s) of a quadratic equation.
```

```
g=function(a,b,c){if(b**2-4*a*c==0) cat("x=",-+b/(2*a),"\n") else
cat("x=",(b+sqrt(b**2-+4*a*c))/(2*a),(-b-sqrt(b**2 4*a*c)) / +(2*a),"\n")}
```

- **Binary operators**: are operators of functions with two arguments.
  - The arguments of such operators are found in either side of the function name.
- R allows us to write our own binary operators of the form: %anything%.

  Syntax: "%anything%"=function (…){…}
  - The matrix multiplication operator, %*%, and the outer product matrix operator %o% are other examples of binary operators defined in this way.

Consider the following binary operator that calculates the dot product of two vectors.

```
> "%H%"=function(x,y){
Summ=0
for(i in 1:length(x)){summ=summ+x[i]*y[i]}
summ}
> c(1,2,3)%H%c(2,3,4)
 [1] 20
```

Lab Exercise3

1. Create a function which will receive a vector x as an argument, compute and return its mean
2. Create a simple function called f1, which adds a pair of numbers
3. Create a function to calculate and return the two sample t-test by taking two independent samples, showing all the steps.
4. Create a function in R that takes two number a and b as an input and then calculate z=a*b and y=b^a and display z and y, let, a=3, b=5

   zy=function (a, b)
5. Create a function that receive the following 3x3 matrix named M as a vector

   10 11 12

   15 18 25

   69 78 89

   Compute and return, transpose, determinant and inverse of matrix in the form of list

CHAPTER 4

Graphical procedures

Graphical facilities are 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. Graphs are important tools for exploring data in statistics and other fields/streams. There are a number of graphs in Statistics, like Pie chart, histogram, scatter, box, steam and leaf, QQ plot…

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Draw graphs using the base and lattice graphics packages
- ✓ Apply fundamental principles of analytic graphics
- ✓ Develop simple summaries and exploratory graphs that optimize data visualization at the beginning stages of data analysis
- ✓ Explain how to visualize data using various types of displays.
    - ➢ Tables
    - ➢ Plots
    - ➢ Charts and Graphs

4.1. Plotting Commands

- ➢ There are a variety of plotting commands in R.
    - ➢ plot()
    - ➢ Automatically produces simple plots for vectors, functions or data frames
- ➢ Specifying labels:
    - ➢ main – provides a title
    - ➢ xlab – label for the x axis
    - ➢ ylab – label for the y axis
- ➢ Specifying range limits
    - ➢ ylim=c(ymin, ymax) # 2-element vector gives range for x axis
    - ➢ xlim=c(xmin, xmax) # 2-element vector gives range for y axis
- ➢ To add additional data use

```
> points(x,y)
> lines(x,y)
```

- Different plotting commands

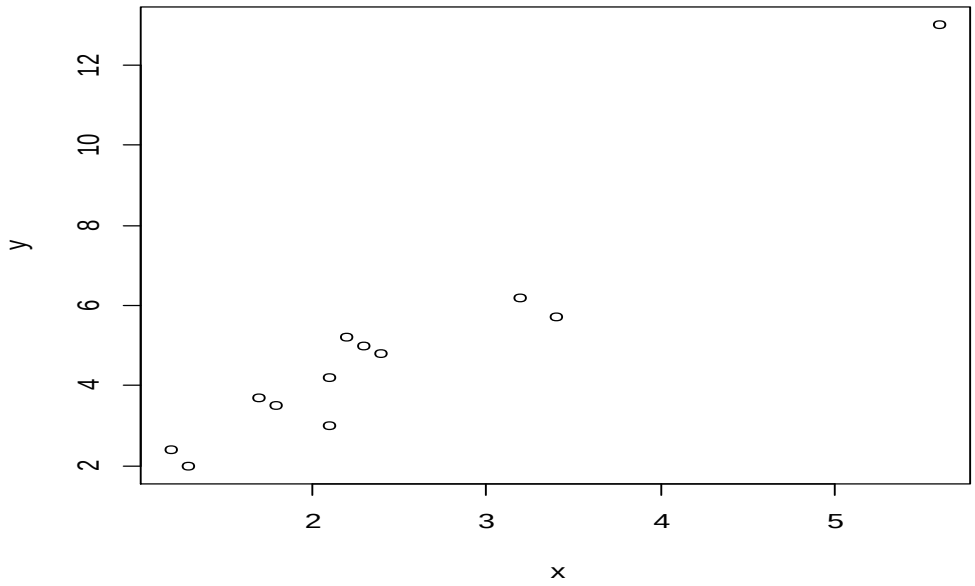
```
> pie()          # Used to plot a Pie Chart
> boxplot()      # Plots Box-and-whiskers plot
> hist()         # Used to plot histograms
> barplot()      # Used to plot a bar graph (Chart)
> legend()       # Adds a legend to a figure
> stem()         # Plots steam and leaf plot
```

4.2. Graphics Parameters

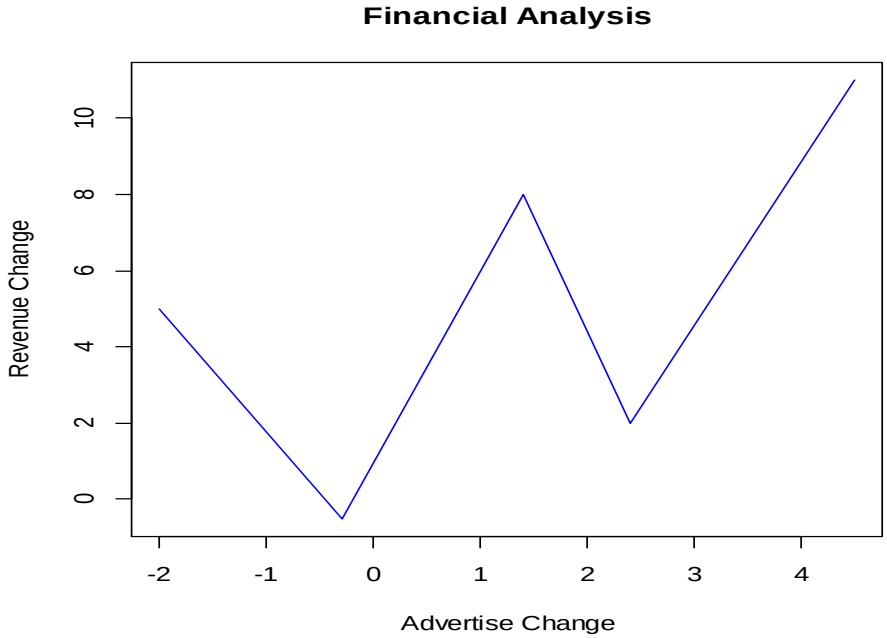
The following sessions detail many of the commonly-used graphical parameters. The R help documentation for the par() function provides a more concise summary; this is provided as a somewhat more detailed alternative. Graphics parameters will be presented in the following form:

- text() – writes inside the plot region, could be used to label data points.
- mtext() – writes on the margins, can be used to add multiline legends
  - text() and mtext() functions can print mathematical expressions created with expression().
- mfrow or mfcol options Take 2 dimensional vector as an argument
  - The first value specifies the number of rows
  - The second specifies the number of columns
- plot types: type="l"
- line types: lty= (takes values from 1 to 8)
- plotting characters: pch=(takes values square (0); circle (or octagon) (1); triangle (2); cross (3); X (4); diamond (5) and inverted triangle (6))
- plotting colors: col=1,2,3,…

```
> x <- c(1.2,3.4,1.3,2.1,5.6,2.3,3.2,2.4,2.1,1.8,1.7,2.2)
> y <- c(2.4,5.7,2.0,3,13,5,6.2,4.8,4.2,3.5,3.7,5.2)
> plot(x,y)
```




```
> x <- c(-2,-0.3,1.4,2.4,4.5)
> y <- c(5,-0.5,8,2,11)
> plot(x, y, type="l", col="blue",xlab="Advertise Change", ylab="Revenue Change",
main="Financial Analysis")
```

Bar Graph: Summary data Consider six clinics treating patients who are smokers, exsmokers and non-smokers. Perform the following operation to create the data.

> clinics <- matrix(c(30, 55, 60, 20, 45, 70, 50, 10, 20, 55, 70, 120, 27, 34,22, 23, 14, 33), nrow = 6)

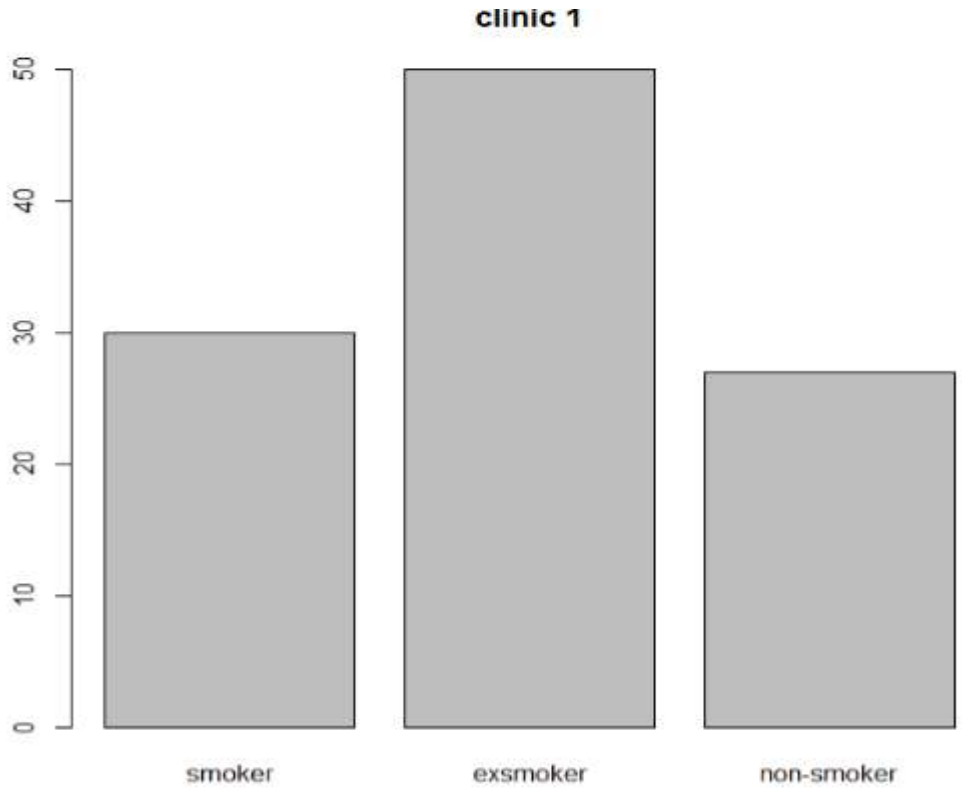
> dimnames(clinics) <- list(paste(c("clinic"), 1:6), c("smoker", "ex-smoker", "non-smoker"))

> clinics

	smoker	exsmoker	non-smoker
clinic 1	30	50	27
clinic 2	55	10	34
clinic 3	60	20	22
clinic 4	20	55	23
clinic 5	45	70	14
clinic 6	70	120	33

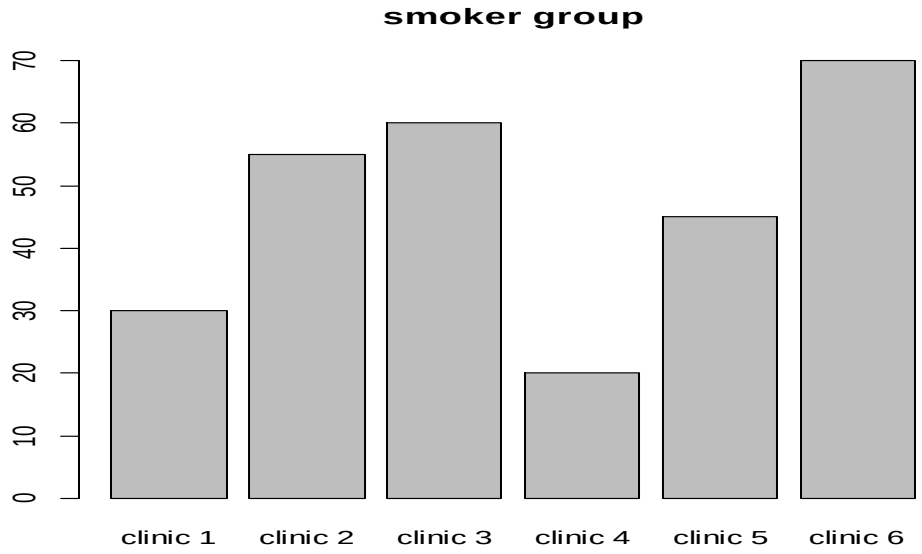
- The following code plots a bar chart for the first clinic (clinic 1) with names on the x-axis (smoker, exsmoker and nonsmoker)

  > barplot(clinics[1,],names=dimnames(clinics)[[2]], main="clinic1")

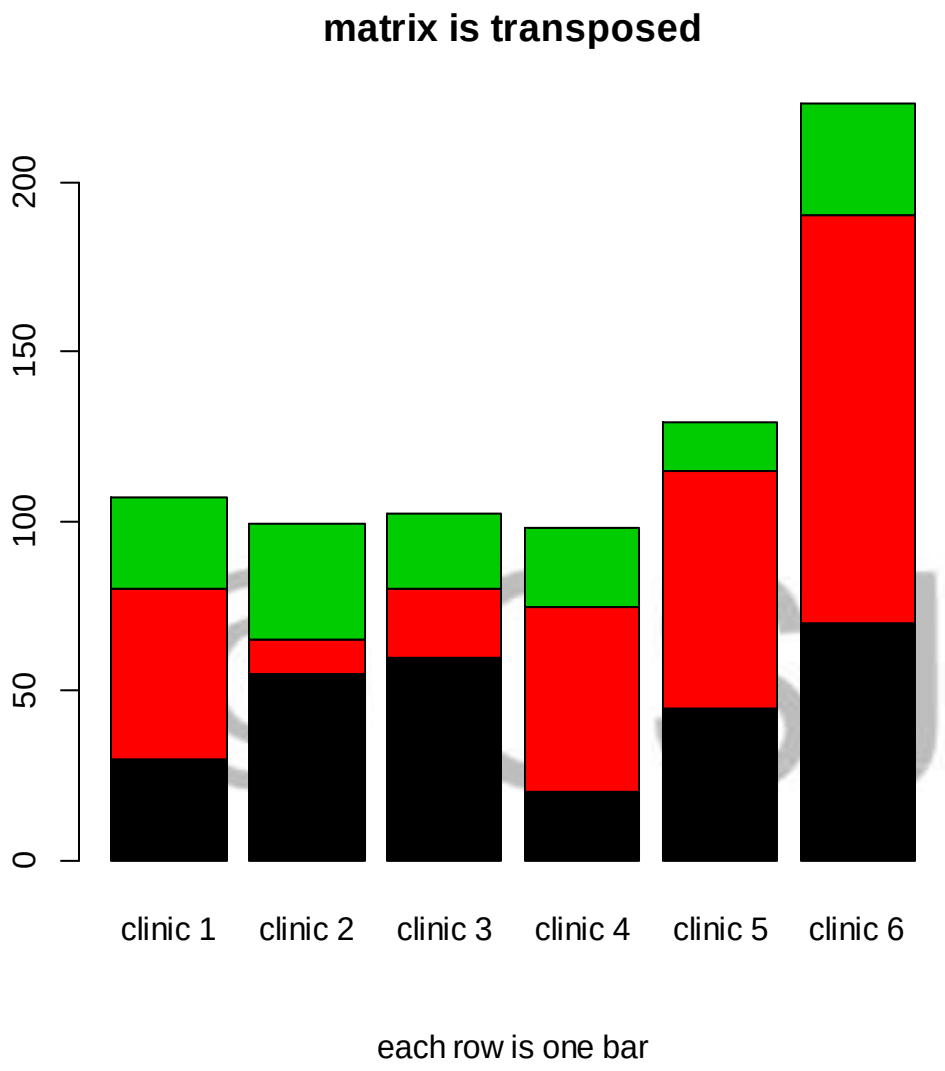


- The following code plots a bar chart for the all clinics who are only smokers.

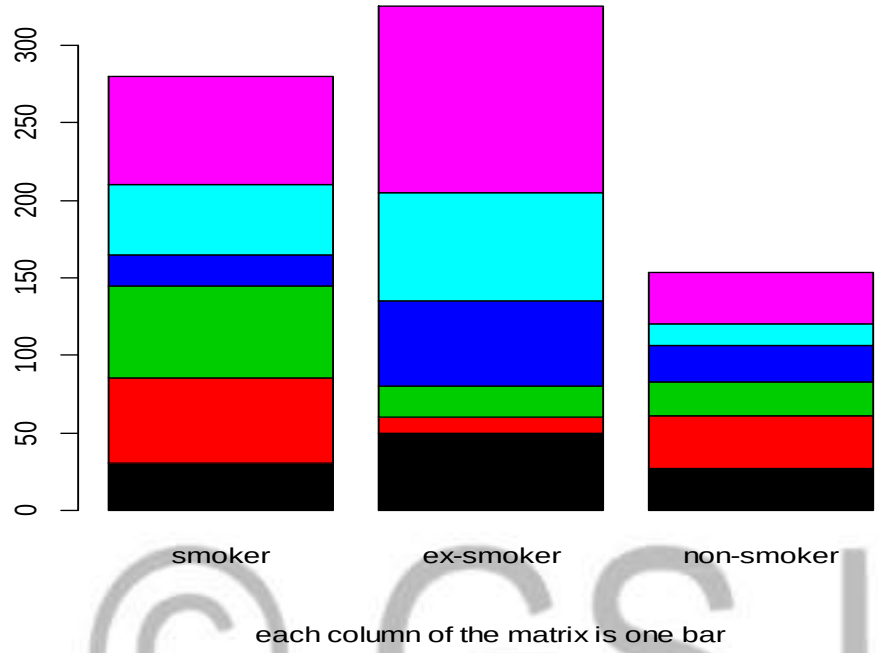
```
> barplot(clinics[,1],names=dimnames(clinics)[[1]], main="smoker group")
```



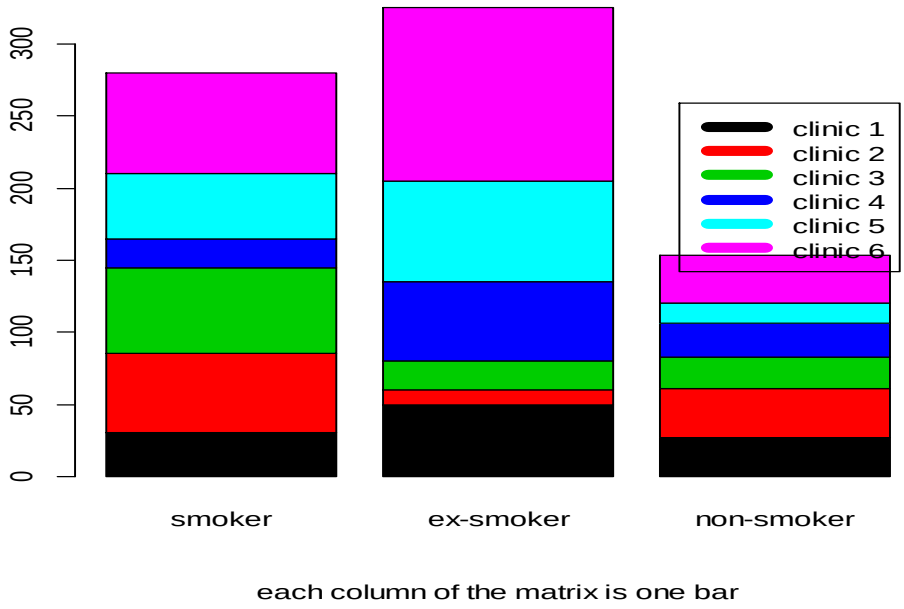
>barplot(t(clinics), names=dimnames(clinics)[[1]], main='matrix is transposed', sub='each row is one bar', col=1:10)



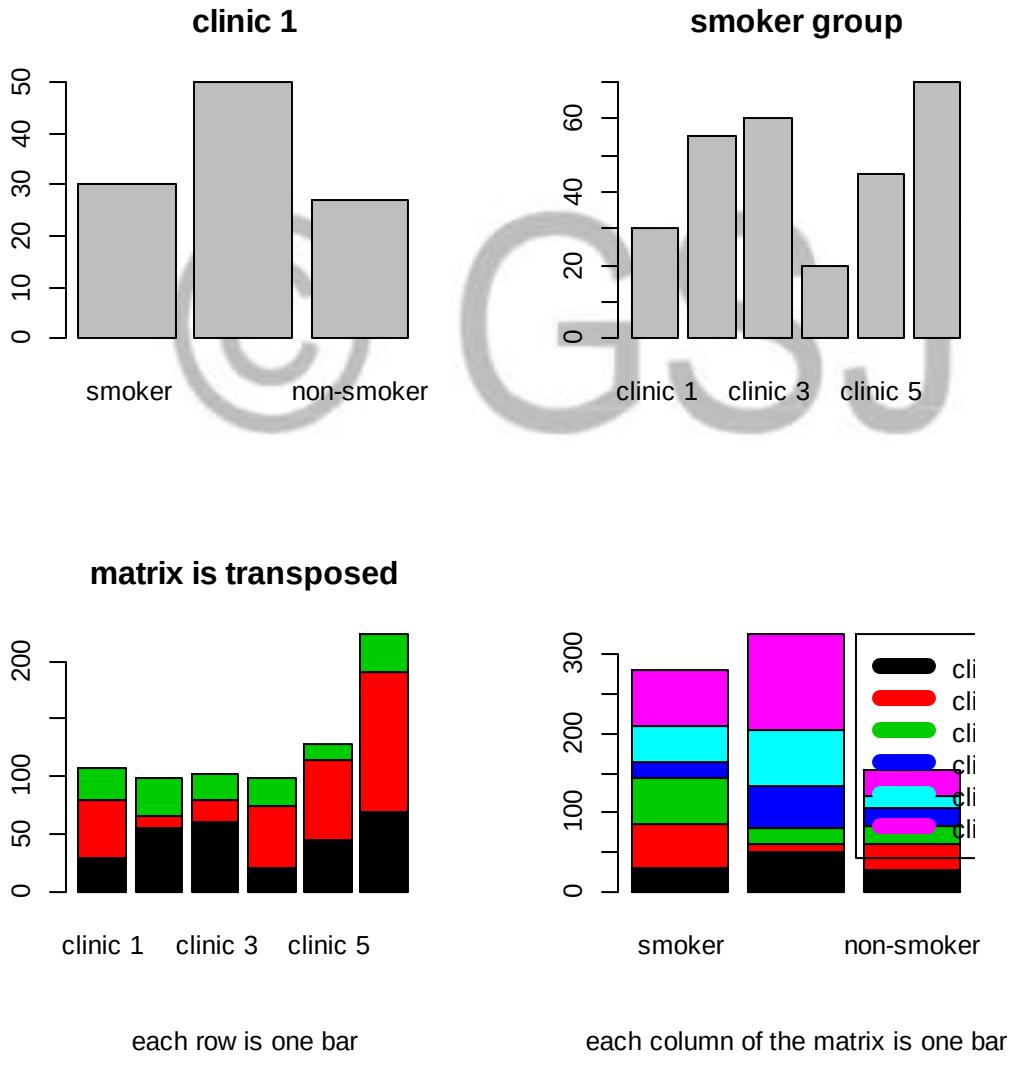
>barplot(clinics,names=dimnames(clinics)[[2]],sub='each column of the matrix is one bar', col=1:6)



>barplot(clinics,names=dimnames(clinics)[[2]],sub= 'each column of the matrix is one bar', col=1:6) legend (locator(1), dimnames(clinics)[[1]] ,col=1:6,lty=1,lwd=8)

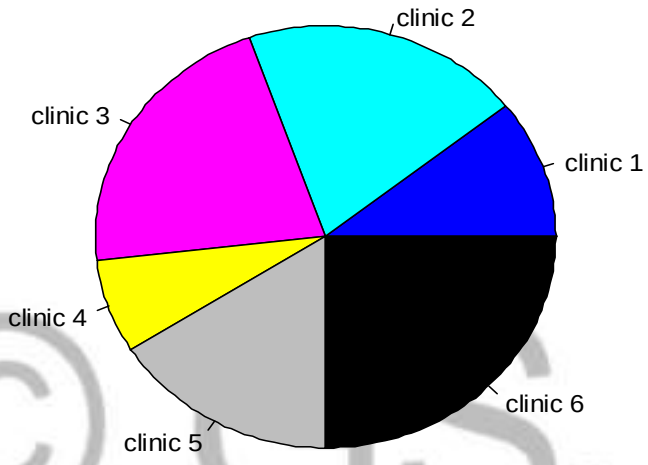


```
> par(mfrow=c(2,2))
> barplot(clinics[1,],names=dimnames(clinics)[[ 2]],main="clinic 1")
> barplot(clinics[,1],names=dimnames(clinics)[[ 1]],main="smoker group")
> barplot(t(clinics),names=dimnames(clinics)[[1 ]],main='matrix is transposed',sub='each row is one bar', col=1:10)
> barplot(clinics,names=dimnames(clinics)[[2]], sub='each column of the matrix is one bar', col=1:6)
> legend (locator(1), dimnames(clinics)[[1]] ,col=1:6,lty=1,lwd=8)
```



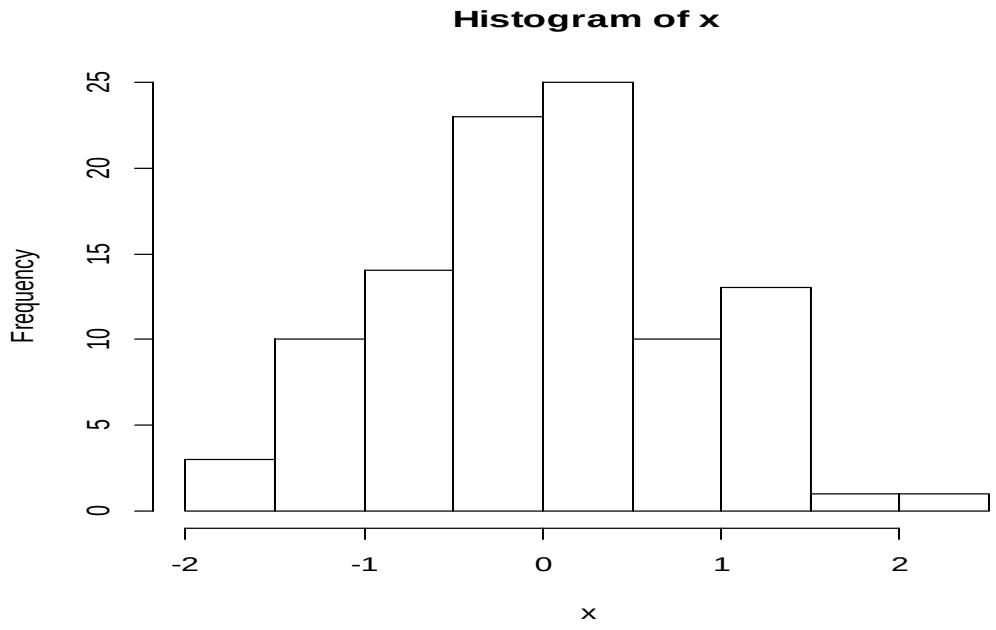
Pie chart:

```
par(mfrow=c(1,1))
 > clinic.names <- dimnames(clinics)[[1]]
> smoker.names <- dimnames(clinics)[[2]]
> pie(clinics[,1],names=clinic.names,col=20:25)
```



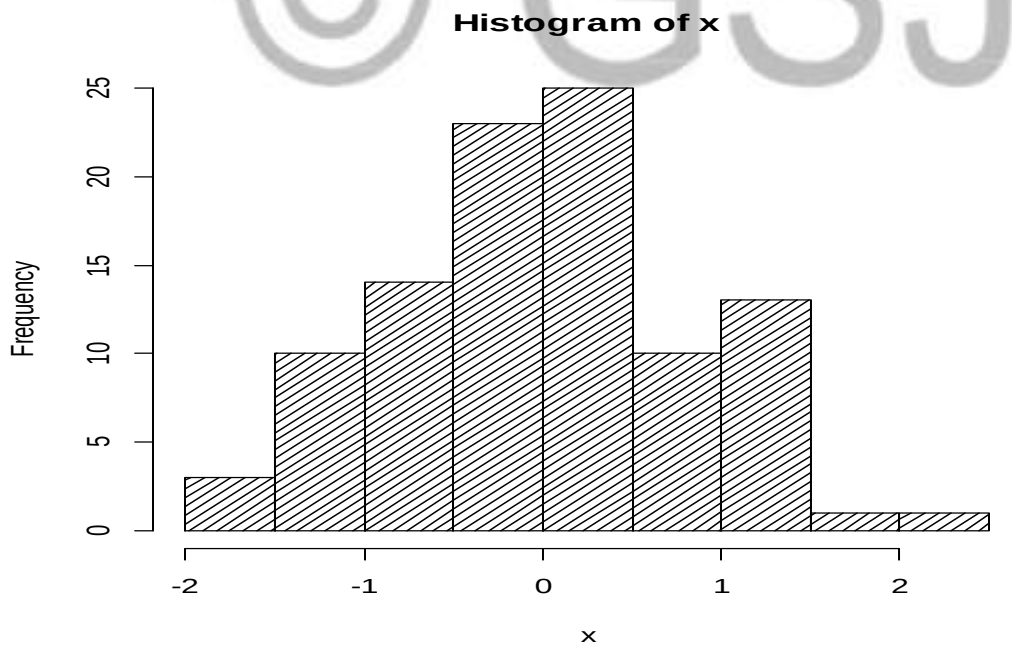
Histogram:

```
set.seed(121343)   # this to fix the numbers to be generated
> x <- rnorm(100)
#Default histogram
> hist(x)
```



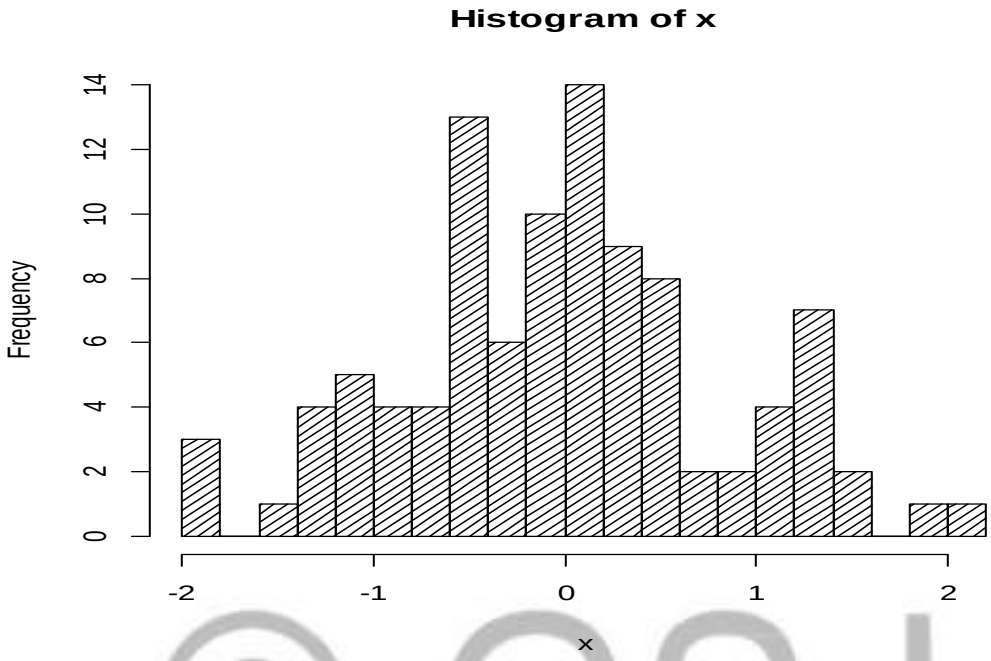
#With shading

> hist(x, density=20)



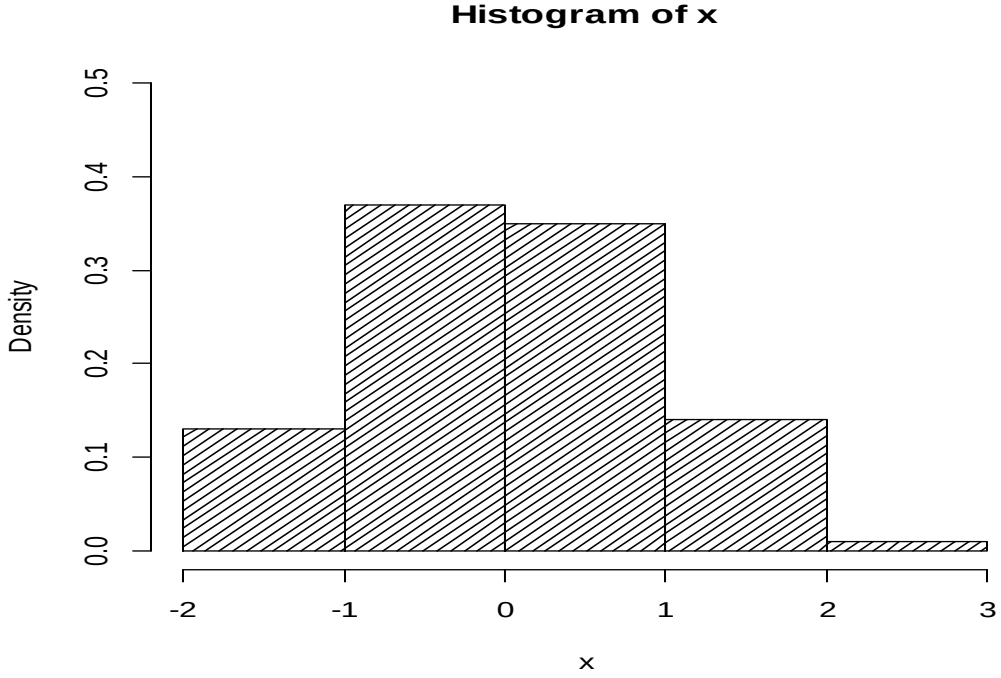
#With specific number of bins

> hist(x, density=20, breaks=20)



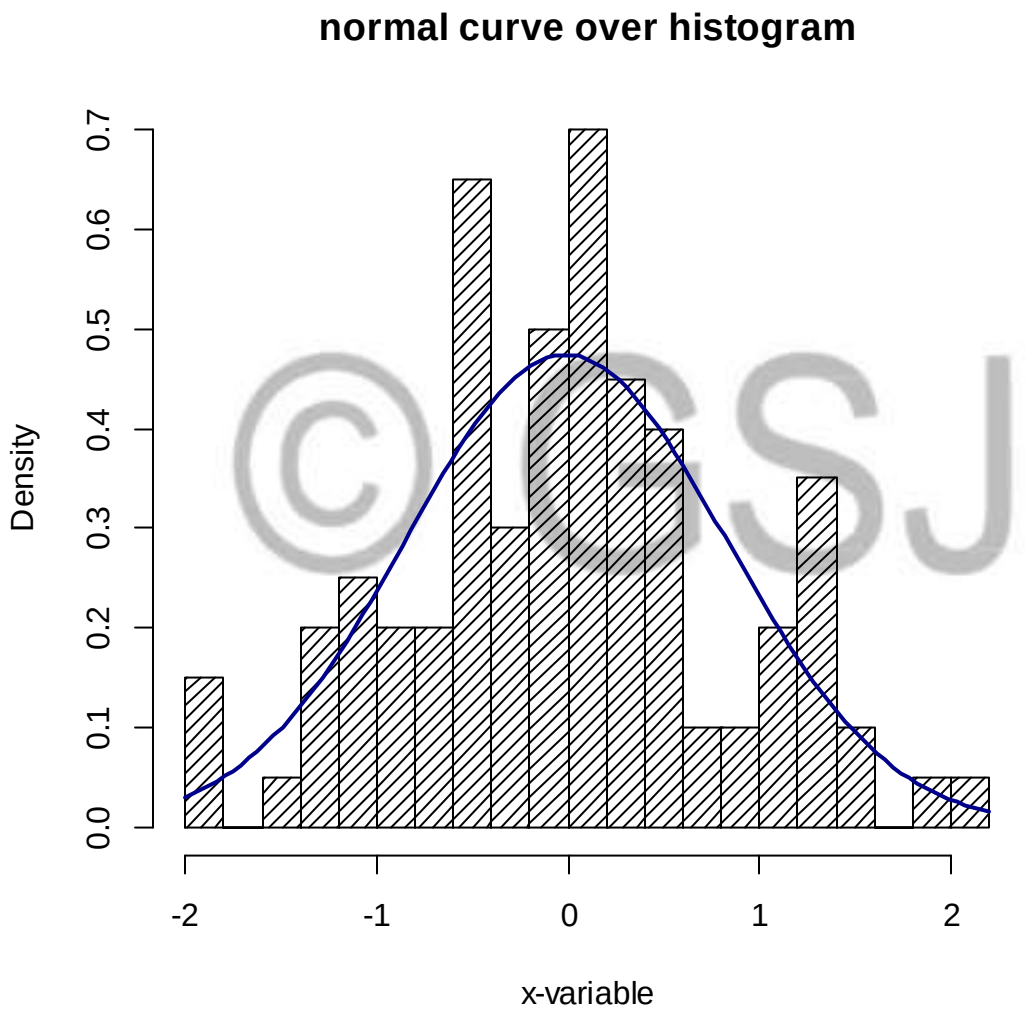
Proportion, instead of frequency, also specifying y-axis

> hist(x, density=20, breaks=3:3,ylim=c(0,.5), prob=TRUE)



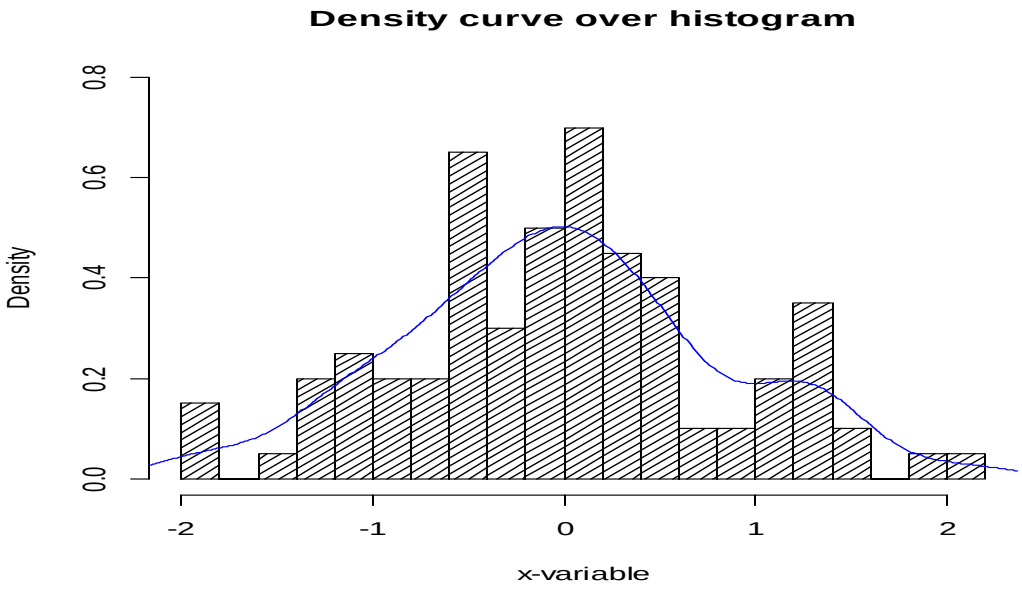
- Overlay normal curve with x-lab and ylim, colored normal curve

```
> m<-mean(x)
> std<-sqrt(var(x))
> hist(x, density=20, breaks=20, prob=TRUE,xlab="x-variable", ylim=c(0, 0.7),
main="normal curve over histogram")
> curve(dnorm(x, mean=m,sd=std),col="darkblue",lwd=2,add=TRUE)
```



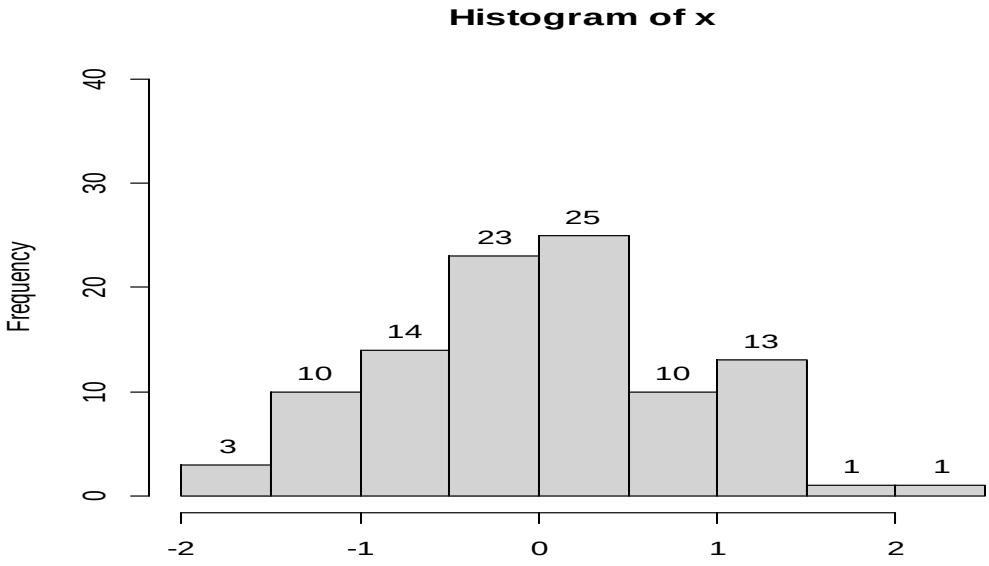
- Overlay density curve with x-lab and ylim

```
> hist(x, density=20, breaks=20, prob=TRUE,xlab="x-variable", ylim=c(0,
0.8),main="Density curve over histogram")
 > lines(density(x), col = "blue")
```



- hist(x) is an object
- names(xh) will show all of its components

```
> xh<-hist(x)
> plot(xh, ylim=c(0, 40), col="lightgray",xlab="", main="Histogram of x") >
text(xh$mids, xh$counts+2, label=c(xh$counts))
```



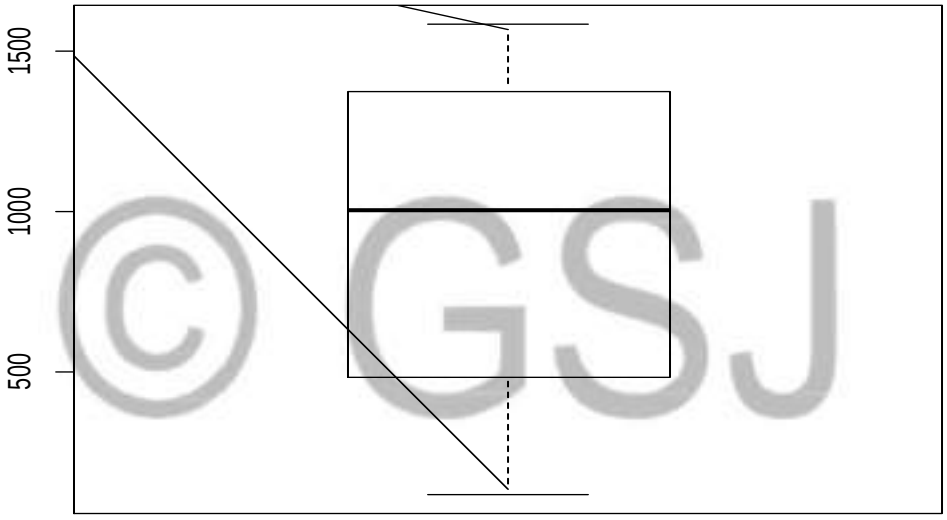
Box plot

Syntax

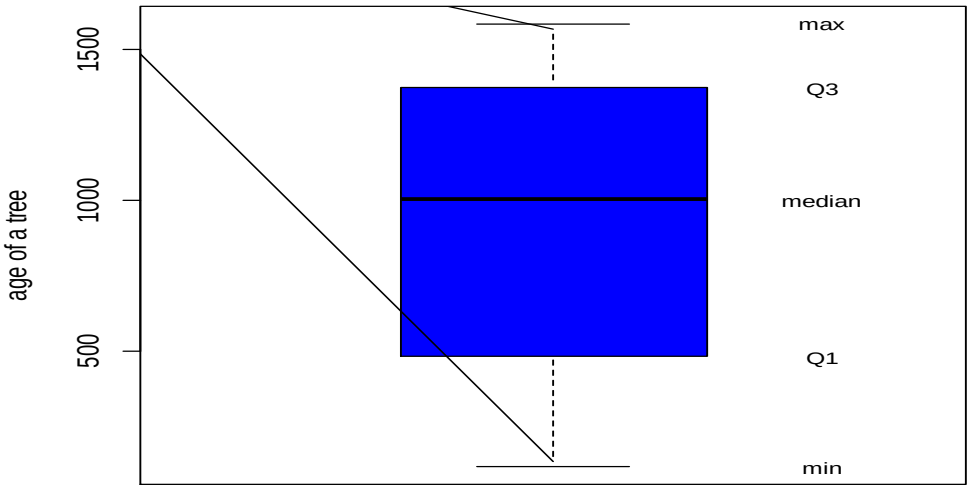
- boxplot(Cont. Variable)
- boxplot(Cont. Variable , xlab="write", boxwex=.4, col="darkblue")
- boxplot(Cont. Variable ~categorical_variable)

Example:

```
> attach(Orange)
> boxplot(age)
```

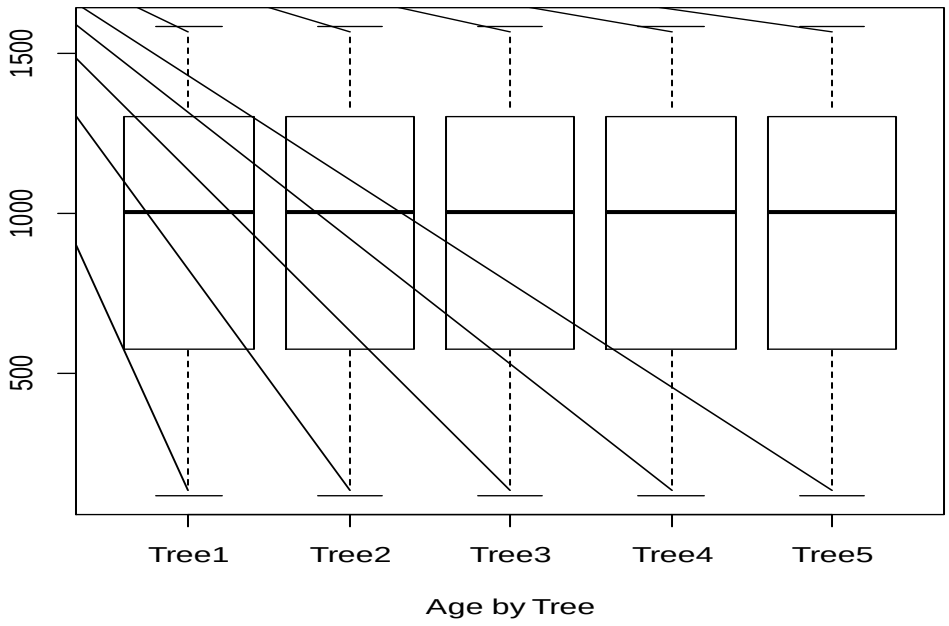



```
> boxplot(age, ylab="age of a tree",col=4)
> f <- fivenum(age)
> text(rep(1.35,5),f,labels=c("min","Q1","median","Q3", "max"),cex=0.8)
```



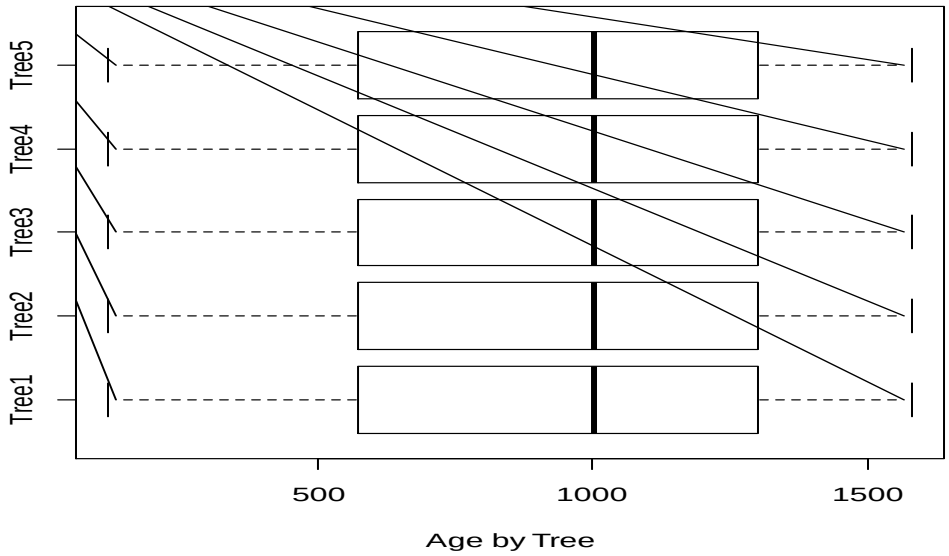
➢ *Rep (1.35,5) is location for labels*

```
> Tlab<-as.vector(c("Tree1","Tree2", "Tree3","Tree4","Tree5"))
>  Treef<-factor(Tree, label=Tlab)
>   boxplot(age ~ Treef, xlab="Age by Tree")
```



If a horizontal boxplot is needed, run following R code

> boxplot(age ~ Treef,horizontal=TRUE, xlab="Age by Tree



Stem and Leaf plots

For a univariate set of data, it is possible to examine its distribution in a number of ways of which steam and leaf plot is one.

Example:

```
> x<-c(42, 23, 43, 34, 49, 56, 31, 47, 61, 54, 46, 34, 26)
> stem(x)
The decimal point is 1 digit(s) to the right of the |
 2 | 36
 3 | 144
 4 | 23679
 5 | 46
 6 | 1
```

- *Notice that there are 5 categories for these 13 numbers, with stems for the 10s digit and leaves for the 1s digit.*

- The stem and leaf plot can be scaled to have more stems by changing the scale option:

```
>stem(x, scale = 2)
 The decimal point is 1 digit(s) to the right of the |
```

2 | 3

2 | 6

3 | 144

3 |

4 | 23

4 | 679

5 | 4

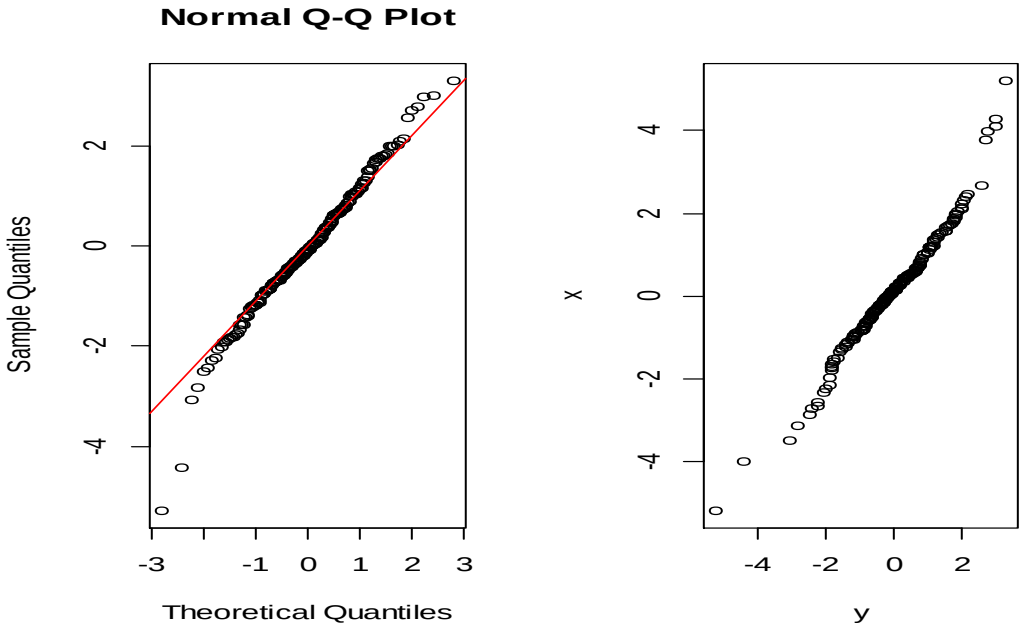
5 | 6

6 | 1

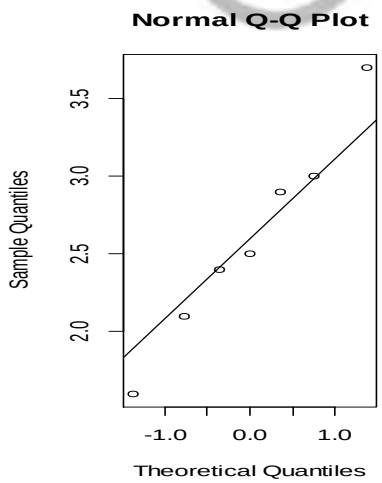
Quantile plots

- qqnorm(x)
  - plots the numeric vector x against the expected Normal order scores (a normal score plot)
- qqline(x)
  - adds a straight line to such a plot by drawing a line through the distribution and data quartiles.
  - this line passes through the first and third quartiles
- qqplot(x, y)
  - plots the quantiles of x against those of y to compare their respective distributions.

```
> set.seed(123456)
> y <- rt(200, df = 5)
> x <-rt(300, df = 5)
> par(mfrow=c(1,2))
> qqnorm(y)
 > qqline(y, col = 2)
> qqplot(y, x)
```




```
> mydata = c(2.4, 3.7, 2.1, 3, 1.6, 2.5, 2.9)
> myquant=qqnorm(mydata)
> qqline(mydata)
```



- *myquant; contains the theoretical quantiles and the original data*

❖ If the observations in mydata come from a normal distribution, then the above plot of mydata versus their population quantiles should give a straight line.

Lab Exercise4

1. Consider the following data on blood type, sex, religious, height and CGPA of 11 randomly samples from statistics department second year students:

Blood type	O	O	A	AB	O	AB	B	O	O	AB	A
Sex	M	M	M	F	M	M	M	M	F	M	M
Religious	O	O	M	M	P	P	O	O	P	O	O
Height	1.55	1.75	1.72	1.65	1.60	1.74	1.56	1.72	1.64	1.68	1.82
CGPA	2.4	2.6	2.8	2.9	3.2	3.0	2.6	2.2	2.2	2.8	2.4

a) Construct one way, two way and higher order table for the categorical variable
b) Construct simple, stacked (component) and compound (side by side) bar graph
c) Construct pie chart
d) Construct simple box plot of CGPA
e) Construct comparative box plot of CGPA of the students by sex
f) Construct histogram of CGPA and height

CHAPTER 5

Statistical Model in R

The requirements for fitting statistical models are sufficiently well defined to make it possible to construct general tools that apply in a broad spectrum of problems. R provides an interlocking suite of facilities that make fitting statistical models very simple. As we mention in the introduction, the basic output is minimal, and one needs to ask for the details by calling extractor functions.

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Run basic models
- ✓ Carryout simulation for estimation and hypothesis test
- ✓ Use the R software for statistical computation and analysis
- ✓ Program maximum likelihood estimators

5.1. Regression

Regression Analysis is a statistical tool for the investigation of relationship between variables. The investigator wants to ascertain causal effect of one variable on another. The investigator also typically assesses the "Statistical significance" of the estimated relationship (the degree of confidence that the true relationship is close to the estimated relationship). Regression may be simple or multiple, linear or non-linear. The template for a statistical model is a linear regression model with independent, homoscedastic errors.

$$y = \chi\beta + \varepsilon$$

General form:

- Response ~expression
  - y ~ x
  - y ~ 1 + x
    - Both imply the same simple linear regression model of y on x. The first has an implicit intercept term, and the second an explicit one.
  - y ~ 0 + x
  - y ~ -1 + x
  - y ~ x - 1

- Simple linear regression of y on x through the origin (that is, without an intercept term)

- The operator ~ is used to define a model formula in R.

❖ Some useful extractors and output from R code.

- coef(object)
  - Extract the regression coefficient (matrix).
  - Long form: coefficients (object).
- deviance(object)
  - Residual sum of squares, weighted if appropriate.
- formula(object)
  - Extract the model formula.
- plot(object)
  - Produce four plots, showing residuals, fitted values and some diagnostics.
- predict (object, newdata=data.frame)
  - The data frame supplied must have variables specified with the same labels as the original. The value is a vector or matrix of predicted values corresponding to the determining variable values in data.frame.
- print(object)
  - Print a concise version of the object. Most often used implicitly.
- residuals(object)
  - Extract the (matrix of) residuals, weighted as appropriate.
  - Short form: resid(object).
- step(object)
  - Select a suitable model by adding or dropping terms and preserving hierarchies.
  - The model with the smallest value of AIC (Akaike's An Information Criterion) discovered in the stepwise search is returned.
- summary(object)
  - Print a comprehensive summary of the results of the regression analysis.
- vcov(object)
  - Returns the variance-covariance matrix of the main parameters of a fitted model

object.

- confint(model_Name, level=0.95)
  - Confidence interval formation for the model parameters
- fitted(model_Name)
  - Used to display predicted values depending on the fitted model
- influence(model_Name)
  - used for regression diagnostics
- shapiro.test()
  - Used to test specifically normality
- ks.test(res,"pnorm", mu,sigma)
  - Generally used for any type of distribution and it has to be specified (like;pnorm) and its parameters (like;mu,sigma).
- lm() is the basic function for fitting ordinary simple/multiple models.
- fitted.model=lm(formula, data=, subset=)

Example: Consider Orange data in R Orange

```
>Orange.lm<-lm(circumference~age,data=Orange)
```

Will give the following output if run " Orange.lm"

```
Call:
lm(formula = circumference ~ age, data = Orange)
Coefficients:
(Intercept)        age
   17.3997      0.1068
```

- Will give the following output if run " coef(Orange.lm)"

```
> coef(Orange.lm)
(Intercept)        age
 17.3996502   0.1067703
```

- Will give the following output if run " deviance(Orange.lm)"

```
> deviance(Orange.lm)
[1] 18594.74
```

- Will give the following output if run " formula (Orange.lm)"

  > formula (Orange.lm)

  circumference ~ age

- To display the predicted values: we will use the following code

```
> fitted(Orange.lm)
     1         2         3         4         5         6         7         8
 29.99855  69.07649  88.29515 124.59706 148.83392 163.88854 186.31030  29.99855
.....................
    33        34        35
148.83392 163.88854 186.31030
```

- Will give the following output if run "confint(Orange.lm,level=0.95)"

```
> confint(Orange.lm,level=0.95)
                  2.5 %    97.5 %
(Intercept) -0.14328303  34.9425835
age          0.08993141   0.1236092
```

- The list of elements of the results of an analysis can found using the names () function

```
> names(Orange.lm)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> names(summary(Orange.lm))
 [1] "call"          "terms"         "residuals"     "coefficients"
 [5] "aliased"       "sigma"         "df"            "r.squared"
 [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

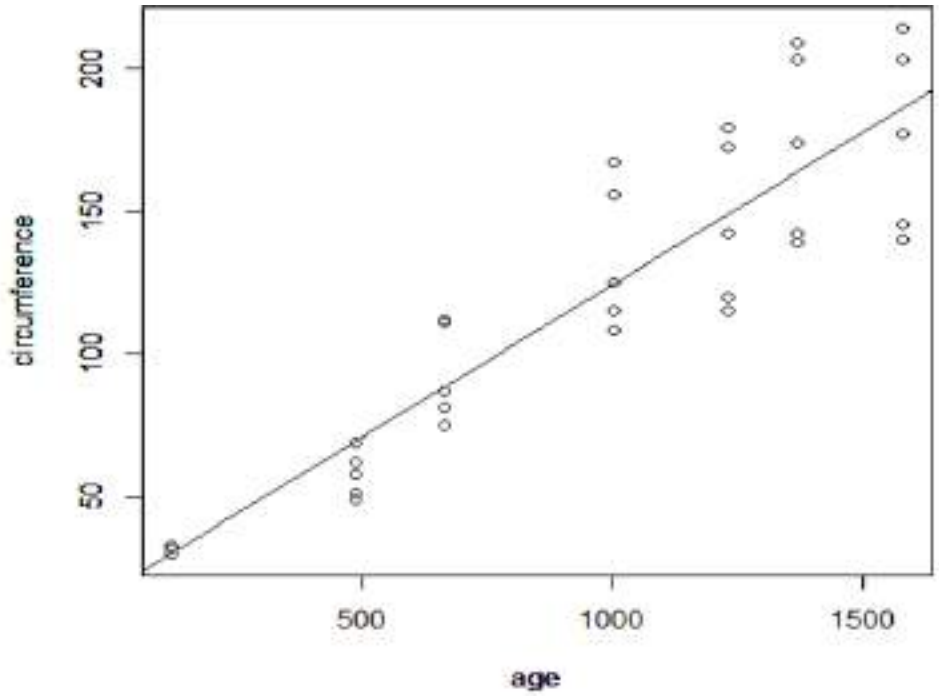
- To extract the elements, the following notation can be used:

```
>Orange.lm$df.residual
[1] 33
> summary(Orange.lm)["r.squared"] $r.squared
```

```
[1] 0.8345167
#Or
>summary(Orange.lm)$r.squared
[1] 0.8345167
```

- You can plot the regression line using the function abline()

```
> plot(age,circumference)
> abline(Orange.lm)
```



5.2. ANOVA Model

Used with Three or more groups to test for MEAN difference, Applicable for a continuous response and a categorical predictor. The hypothesis is:

- H0: The means of all the groups are equal.
- Ha: Not all the means are equal
  - doesn't say how or which ones differ.
- The general syntax (expression for fitting ANOVA is:

```
aov(response ~ predictor, data=)
```

Example: Consider the following data with three groups of diet type and recorded weight.

```
> group <- c(1,1,1,2,2,2,3,3,3)
```

```
> weight <-c(43, 40, 35, 41, 47, 54, 39, 34,37)
>  group <- as.factor(group)
> analysis.aov<-aov(weight~group)
> aov(weight~group)
Call:
  aov(formula = weight ~ group)
Terms:
                group Residuals
Sum of Squares 184.8889   130.0000
Deg. of Freedom       2         6
Residual standard error: 4.654747
Estimated effects may be unbalanced
```

- To get detailed output, first fit the linear model as follows

```
> analysis2 <- lm(weight ~ group)
> summary(analysis2)    #Estimate of diet Effects
Call:
lm(formula = weight ~ group)
Residuals:
   Min     1Q  Median     3Q    Max
-6.3333 -2.6667  0.3333  2.3333  6.6667

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   39.333      2.687  14.636 6.39e-06 ***
group2         8.000      3.801   2.105   0.0799 .
group3        -2.667      3.801  -0.702   0.5092
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Residual standard error: 4.655 on 6 degrees of freedom

Multiple R-squared: 0.5872, Adjusted R-squared: 0.4495

F-statistic: 4.267 on 2 and 6 DF, p-value: 0.07037

- To get detailed output, first fit the linear model as follows

```
> anova(analysis2)
Analysis of Variance Table
Response: weight
          Df Sum Sq Mean Sq F value  Pr(>F)
group      2 184.89  92.444  4.2667 0.07037 .
Residuals  6 130.00  21.667
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

- Multiple comparision
- Tukey Honestly Significant Differences

```
> analysis.aov<-aov(weight~group)
>  TukeyHSD(analysis.aov)
  Tukey multiple comparisons of means
    95% family-wise confidence level
Fit: aov(formula = weight ~ group)
$group
         diff        lwr       upr     p adj
2-1  8.000000  -3.661237 19.661237 0.1689400
3-1 -2.666667 -14.327904  8.994570 0.7714179
3-2 -10.666667 -22.327904  0.994570 0.0692401
```

Generalized Linear Model

Generalized linear models enable one to model response variables that follow any distribution from the exponential family. The R function glm() fits generalized linear models, Has the same specification as lm() or aov(). The distribution of the response needs to be specified

- ## Dobson (1990) Page 93: Randomized Controlled Trial:

```
> counts <- c(18,17,15,20,10,20,25,13,12)
> outcome <- gl(3,1,9)
> treatment<-gl(3,3)
> print(d.AD <- data.frame(treatment,outcome, counts))
  treatment outcome counts
1         1       1     18
2         1       2     17
3         1       3     15
4         2       1     20
5         2       2     10
6         2       3     20
7         3       1     25
8         3       2     13
9         3       3     12
> glm.D93 <- glm(counts ~ outcome + treatment, family = poisson())
> anova(glm.D93)
Analysis of Deviance Table
Model: poisson, link: log
Response: counts
Terms added sequentially (first to last)
          Df Deviance Resid. Df Resid. Dev
NULL                      8    10.5814
outcome    2   5.4523        6     5.1291
treatment  2   0.0000        4     5.1291
> summary(glm.D93)
Call:
glm(formula = counts ~ outcome + treatment, family = poisson())
Deviance Residuals:
      1          2          3          4          5          6          7          8
```

-0.67125 0.96272 -0.16965 -0.21999 -0.95552 1.04939 0.84715 -0.09167

9

-0.96656

Coefficients:

	Estimate	Std. Error	z value	Pr(>|z|)	
(Intercept)	3.045e+00	1.709e-01	17.815	<2e-16	***
outcome2	-4.543e-01	2.022e-01	-2.247	0.0246	*
outcome3	-2.930e-01	1.927e-01	-1.520	0.1285	
treatment2	8.717e-16	2.000e-01	0.000	1.0000	
treatment3	4.557e-16	2.000e-01	0.000	1.0000	

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

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 10.5814 on 8 degrees of freedom

Residual deviance: 5.1291 on 4 degrees of freedom

AIC: 56.761

Number of Fisher Scoring iterations: 4

1.5. Time Series Models

- An AR model expresses a time series as a linear function of its past values. The order of the AR model tells how many lagged past values are included. The simplest AR model is the first-order autoregressive, or AR (1), model.

$$y_t = \alpha_1 y_{t-1} + e_t$$

Where

- y_t mean adjusted series in t, y_{t-1} is series in the previous year
- α_1 is the lag-1 autoregressive coefficient
- e_t is random noise/shock, noise/residual.

- The moving average (MA) model is a form of ARMA model in which the time series is regarded as a moving average (unevenly weighted) of a random shock series e_t

- The first-order moving average, or MA (1), model is given by

$$y_t = e_t + c_1 e_{t-1}$$

where e_t and e_{t-1} residual at t and t-1, c_1 is the first order moving average coefficient

- The autoregressive model includes lagged terms on the time series itself,
- the moving average model includes lagged terms on the noise or residuals.
- including both types of lagged terms, autoregressive-moving average, or ARMA, models can be found.
- The order of the ARMA model is included in parentheses as ARMA(p,q),
- where is the autoregressive order and q the moving-average order, the simplest ARMA model is first-order autoregressive and first order moving average, or ARMA (1,1).

$$y_t = \alpha_1 y_{t-1} + c_1 e_{t-1} + e_t$$

- The following steps should be used in modeling
  - Identification: (AR, MA, ARMA) using acf and pacf
  - Estimation: coefficients, by least square or iterative
  - Diagnostic: randomness of error, estimated coefficients significantly different from zero

Attention: this is a provoke about your time series knowledge for more about time series modelling please have a look at your Course, TIME SERIES ANALYSIS

- Let us consider the following data (Age of Death of Successive Kings of England)

```
> death<-c(60,43,67,50,56,42,50,65,68,43,65,34, 47,34,49,41,13,35,53,56,16,43,69,59,48,59,86,
55,68,51,33,49,67,77,81,67,71,81,68,70,77,56)
```

- Once you have read/stored the time series data into R,
  - The next step is to store the data in a time series object in R, so that you can use R's many functions for analysing time series data.
  - To store the data in a time series object, we use the ts() function in R.
  - For example, to store the data in the variable 'death' as a time series object in R

```
> kingsdeath<-ts(death)
```

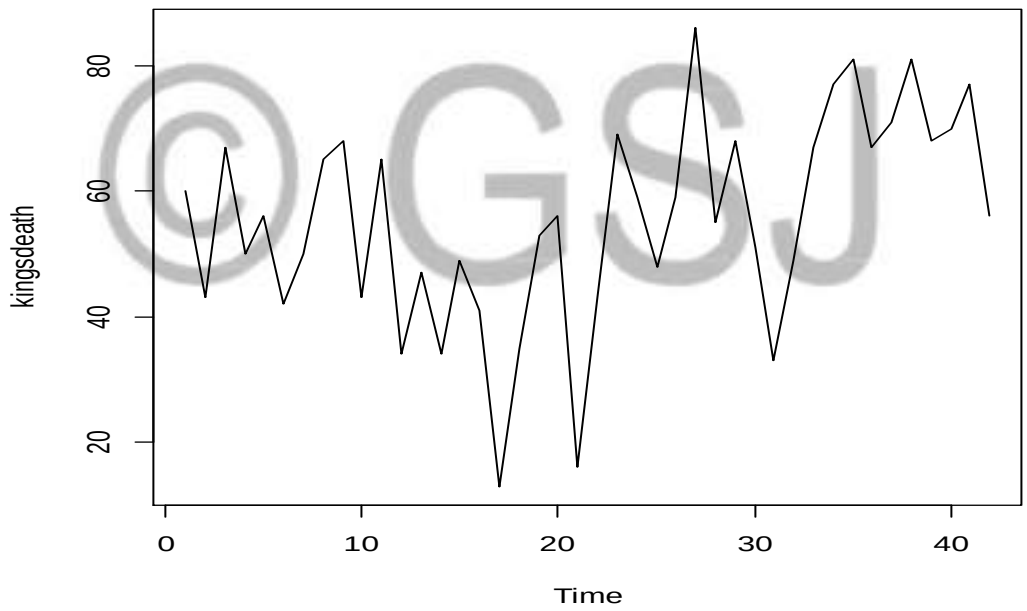
Note: read data into R using the scan() function, which assumes that your data for successive

time points is in a simple text file with one column.

```
> ts(death)
Time Series:
Start = 1
End = 42
Frequency = 1
 [1] 60 43 67 50 56 42 50 65 68 43 65 34 47 34 49 41 13 35 53 56 16 43 69 59 48
[26] 59 86 55 68 51 33 49 67 77 81 67 71 81 68 70 77 56
```

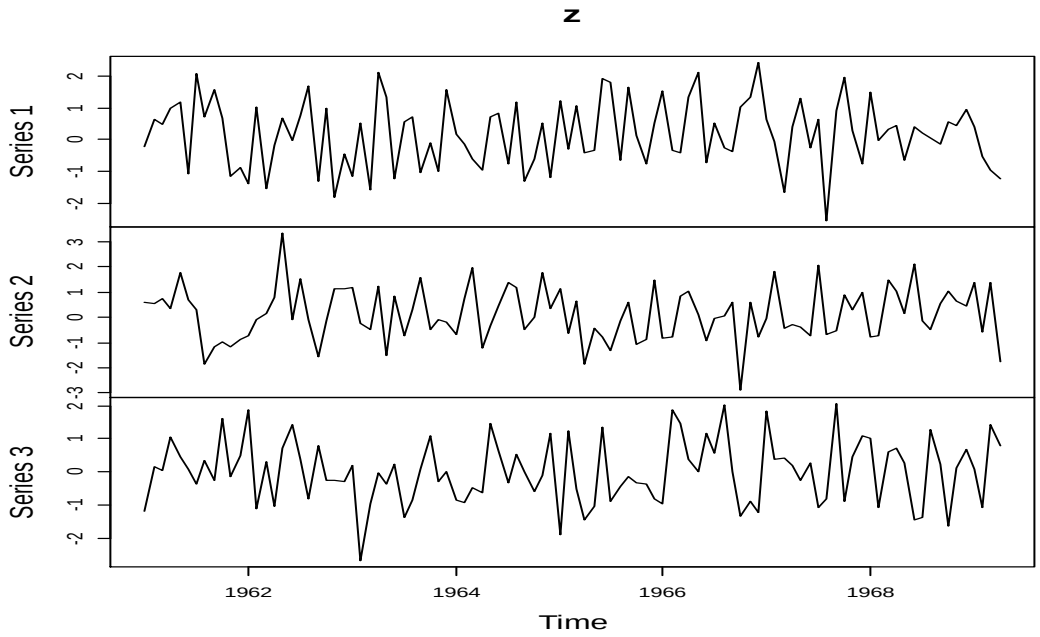
- Plotting Time Series

```
> plot.ts(kingsdeath)
```

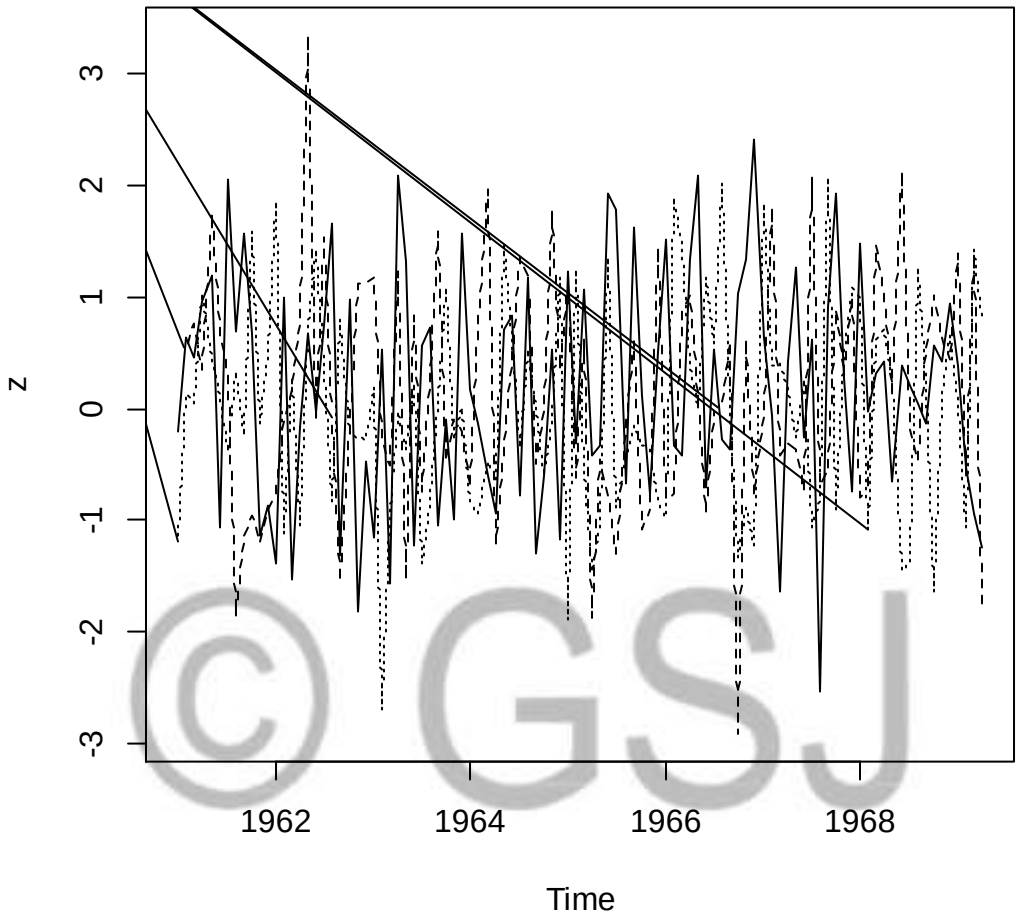


The following generates from a normal distribution 300 elements and classifies into 100 rows and 3 columns and converts to time series data.

```
> z <-ts(matrix(rnorm(300), 100, 3), start=c(1961, 1), frequency=12)
>  plot(z)
```



>plot(z, plot.type="single", lty=1:3)

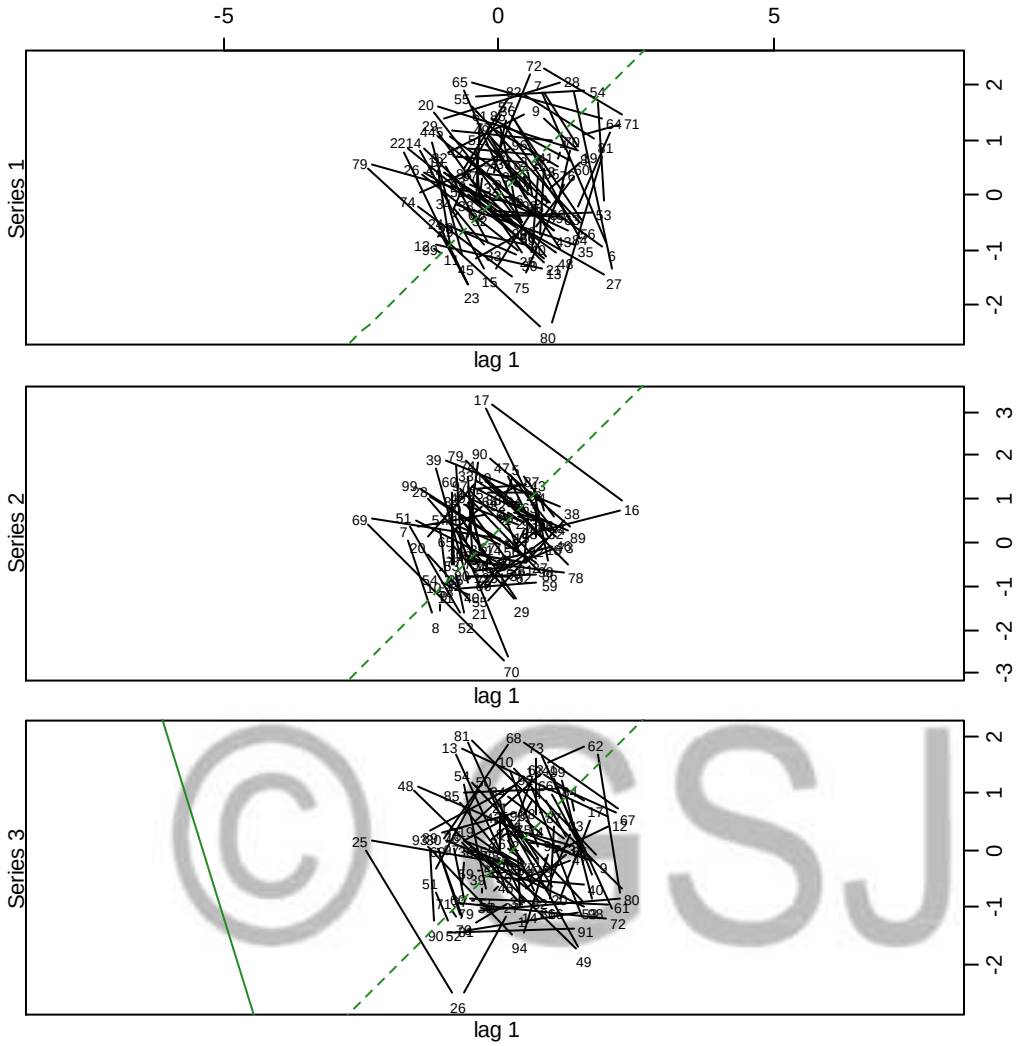


- Plot time series against lagged versions of themselves, helps visualizing 'auto-dependence' even when auto-correlations vanish.
- The following command plots lags against the observed.

  > lag.plot(x, lags = 1, layout = NULL, set.lags = 1:lags, main = NULL, asp = 1, diag = TRUE, diag.col = "gray", type = "p", oma = NULL, ask = NULL, do.lines = (n <= 150), labels = do.lines, ...)

Example: plot

> lag.plot(z, 1, diag.col = "forest green")



- Fit an autoregressive time series model to the data, by default selecting the complexity by AIC.
- Syntax:

  > ar(x, aic = TRUE, order.max = NULL, method=c("yule-walker", "burg", "ols", "mle", "yw"), na.action, series, ...)

Example: On England kings age at death data

```
> kingsdeath.ar <-ar(kingsdeath,aic=TRUE,method="ols",order.max=2)
> kingsdeath.ar
Call:
ar(x = kingsdeath, aic = TRUE, order.max = 2, method = "ols")
```

Coefficients:

1

0.4006

Intercept: -0.108 (2.368)

Order selected 1 sigma^2 estimated as 229.9

- The following used to predict time series model.

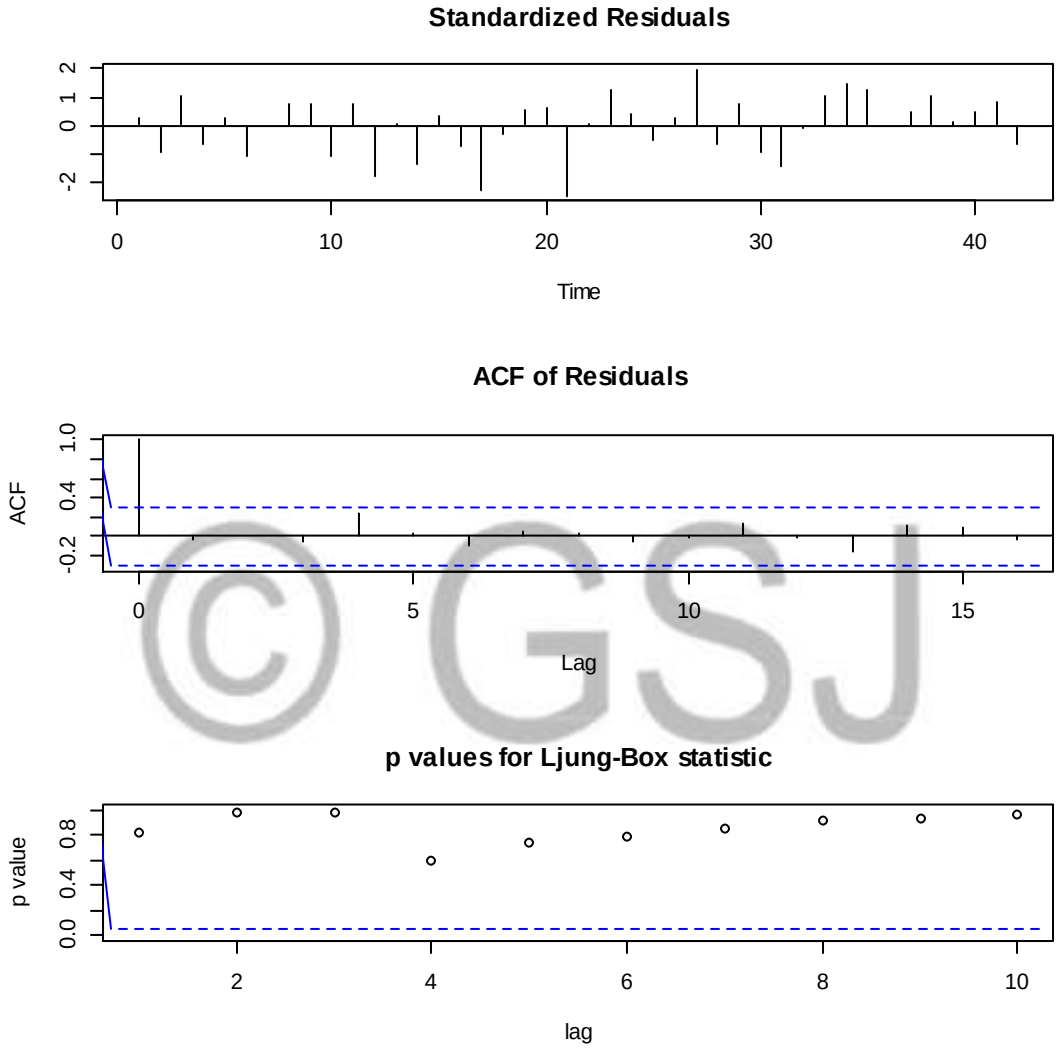
```
> predict(kingsdeath.ar, n.ahead=10)
$pred
Time Series:
Start = 43
End = 52
Frequency = 1
 [1] 55.46385 55.24907 55.16303 55.12856 55.11476 55.10923 55.10701 55.10612
 [9] 55.10577 55.10562
$se
Time Series:
Start = 43
End = 52
Frequency = 1
 [1] 15.16372 16.33518 16.51544 16.54418 16.54879 16.54953 16.54965 16.54967
 [9] 16.54967 16.54967
Warning message:
In object$var.pred * vars :
  Recycling array of length 1 in array-vector arithmetic is deprecated.
  Use c() or as.vector() instead.
```

- Fit an ARIMA model to a univariate time series.
  - Syntax
  - arima(x, order = c(0, 0, 0), seasonal = list(order = c(0, 0, 0), period = NA), xreg = NULL, include.mean = TRUE, transform.pars = TRUE, fixed = NULL, init = NULL, method = c("CSS-

ML", "ML", "CSS"), n.cond, optim.control = list(), kappa = 1e6)

> kingsdeath.arima <- arima(kingsdeath, c(3, 0, 0))

>tsdiag(kingsdeath.arima) # Diagnostic fitted model



> predict(kingsdeath.arima,5) # forecasting for 5 years

$pred

Time Series:

Start = 43

End = 47

Frequency = 1

[1] 58.84069 60.44720 57.45848 56.97526 57.00756

```
$se

Time Series:

Start = 43

End = 47

Frequency = 1

[1] 14.70236 15.52621 15.66580 16.02394 16.15544
```

Example The following are data about a population in 1994.

Month	Population (thousand persons)	Number of birth (persons)	Number of death (persons)
1	10 273	10 238	13 888
2	10 270	9 285	12 825
3	10 267	10 105	12 516
4	10 265	9 617	11 753
5	10 262	9 548	12 328
6	10 260	9 717	11 839
7	10 258	9 965	11 848
8	10 257	9 980	11 722
9	10 256	9 844	10 968
10	10 252	9 021	12 542
11	10 249	8 740	11 743
12	10 246	9 538	12 917

a) Create a sequence chart about the number of birth and death
b) Create a linear trend model for the number of birth
c) Give estimation for the number of birth in January 1995

Lab Exercise 5

1. Consider the following data

Income (Y)	Age (x1)	Hours worked per day(x2)
2841	29	12
1876	21	8
2934	62	10
1552	18	10
3065	40	11
3670	50	11
2005	65	5
3215	44	8
1930	17	8
2010	70	6
3111	20	9
2882	29	9
1683	15	5
1817	14	7
4066	33	12

a) Compute correlation matrix and construct matrix scatter plot
b) Fit multiple linear regression model
c) Check all assumptions of the model
d) Identify outlier and influential observations
e) Test significance of the model and individual regression coefficient
f) What will be the income of household for age=54 and hours worked per day=4

2. The following are the changes on the blood pressure from patients when given three different drugs where used under controlled conditions followed for 18 months.

Placebo	8.4	9.8	8.1	4.0	3.9	4.1	9.3
Standard	18.2	20.1	17.6	16.8	18.8	19.7	19.1
New	19.4	22.7	19.1	18.4	25.9	20.4	21.7

a) Test whether the differences among mean change on the blood pressure are attributed to types of drugs at 5% level of significance.

b) If differences among mean changes on the blood pressure are statistically significant, identify where the difference lies using pairwise mean comparison and draw your conclusions.

3. consider the following data collected to determine whether there is really association between an employee performance in the company training program and his/her ultimate success in job the company take a sample of 458 cases from its very extensive files as shown below:

   Below average=50, 20, 15

   Average=70, 89, 50

   Above average=39, 60, 65

   Is Provide sufficient evidence to indicate performance in the training and success in the jobs are dependent at 5% level of significance?

4. A student investigates two lifestyle factors affecting blood pressure: diet and smoking. For a number of her female classmates. She determines their smoking habits, the amount of sodium in their diet, and their systolic blood pressure (in mmHg). Analyze her data as factorial design ANOVA. Are mean separation techniques indicated? Explain.

	Sodium intake	
	Moderate	High
Non-smokers	129	132
	125	137
	129	130
	132	148
	125	154
	128	158
Smokers	140	165
	126	152
	120	140
	137	167
	142	142
	147	177

5. There are four available forth is comparative study of tire performance. It is beloved that tires wearing out in different rat at different location of a car. Tires were installed in four different location Right-front (RF), left-front (LF), Right-Rear (RR), and Left-Rear (LR). The measurements of the wearing of tires in this investigation are listed in the following table from ablating square Design setting. Three factors are considered in this study. They are position, car and the different tires studied in this investigation.

Position					
Car		RF	LF	RR	LR
	1	A(32)	B(33)	C(47)	D(53)
	2	B(36)	D(53)	A(42)	C(54)
	3	C(51)	A(44)	D(62)	B(49)
	4	D(81)	C(78)	B(72)	A(73)

Then Test null hypothesis that

i. There is no significant difference in average tire wearing between different tire positions
ii. There is no significant differences in average tire wearing between different cars used
iii. No significant differences in average tire wearing between different brands of tires
iv. draw your conclusions with respect to three factors are considered

CHAPTER 6

Introduction to SAS

SAS is a comprehensive statistical software system which integrates utilities for storing, modifying, analyzing, and graphing data. SAS is an abbreviation Statistical Analysis System. There is a large range of modules that can be added to the basic system, known as BASE SAS. Here we concentrate on the STAT and GRAPH modules in addition to the main features of the base SAS system.

Chapter Objectives:

At the end of this chapter the learners will be able to:

- ✓ Explain SAS language
- ✓ Print from SAS using "proc print"
- ✓ Execute and interpret the "proc means" procedure.
- ✓ Distinguish between temporary and permanent SAS data sets
- ✓ Create new variables
- ✓ Use the "set" statement to select a subset of data
- ✓ Use "If-then-else-then statements
- ✓ Explain the effects of missing data
- ✓ Use the "length" statement
- ✓ Use SAS mathematical expressions and SAS functions
- ✓ Know different data handling and manipulation techniques by using SAS software
- ✓ Carryout simulation for estimation and hypothesis test
- ✓ Use SAS to carry out simple graphs and data analysis

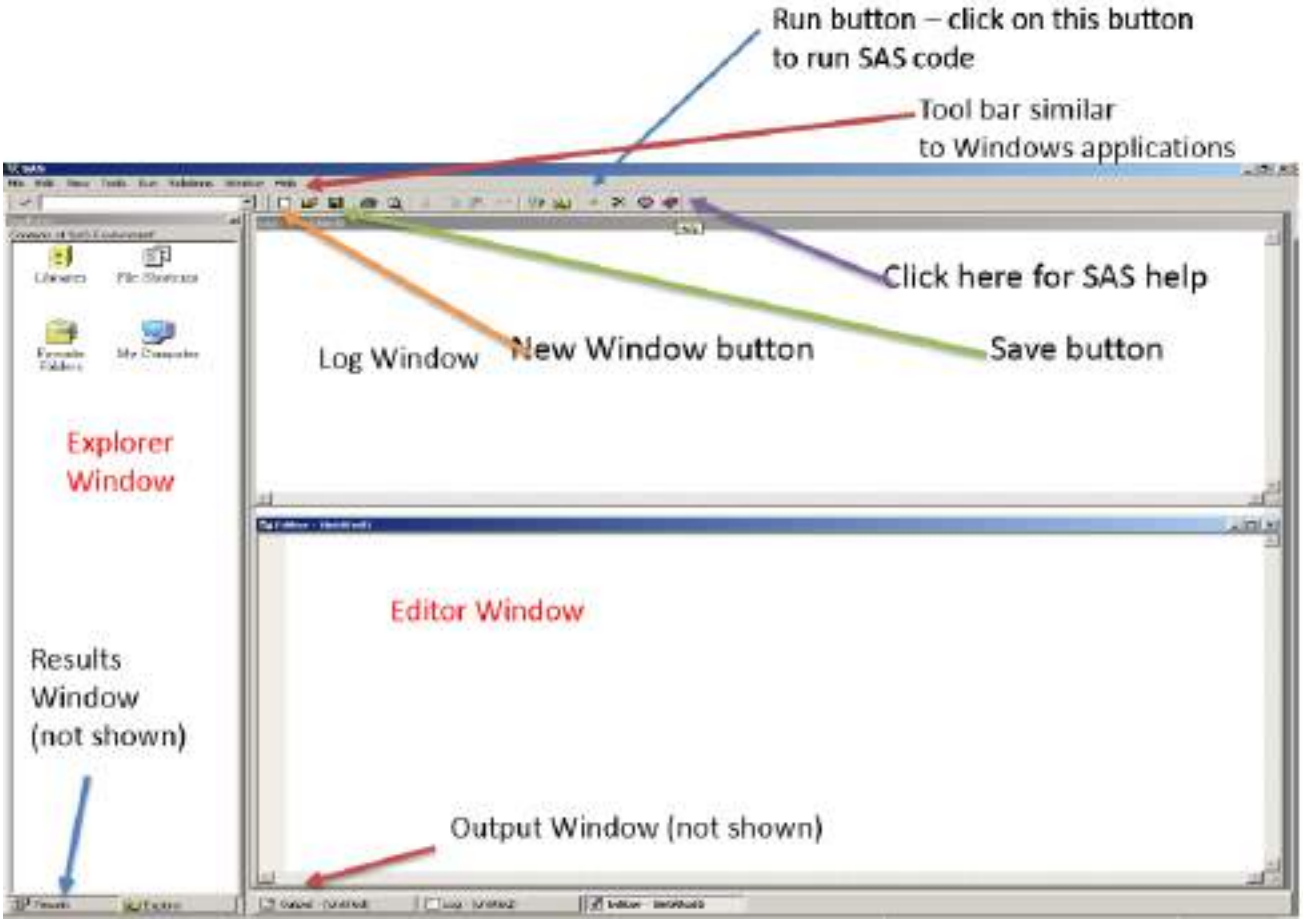
6.1. SAS Language

At the heart of SAS is a programming language composed of statements that specify how data are to be processed and analyzed. The statements correspond to operations to be performed on the data or instructions about the analysis. A SAS program consists of a sequence of SAS statements grouped together into blocks, referred to as “steps.” These fall into two types: data steps and procedure (proc) steps. A data step is used to prepare data for analysis. It creates a SAS data set and may reorganize the data and modify it in the process. A proc step is used to perform a particular type of analysis, or statistical test, on the data in a SAS data set. A typical program might comprise

a data step to read in some raw data followed by a series of proc steps analyzing that data. If, in the course of the analysis, the data need to be modified, a second data step would be used to do this.

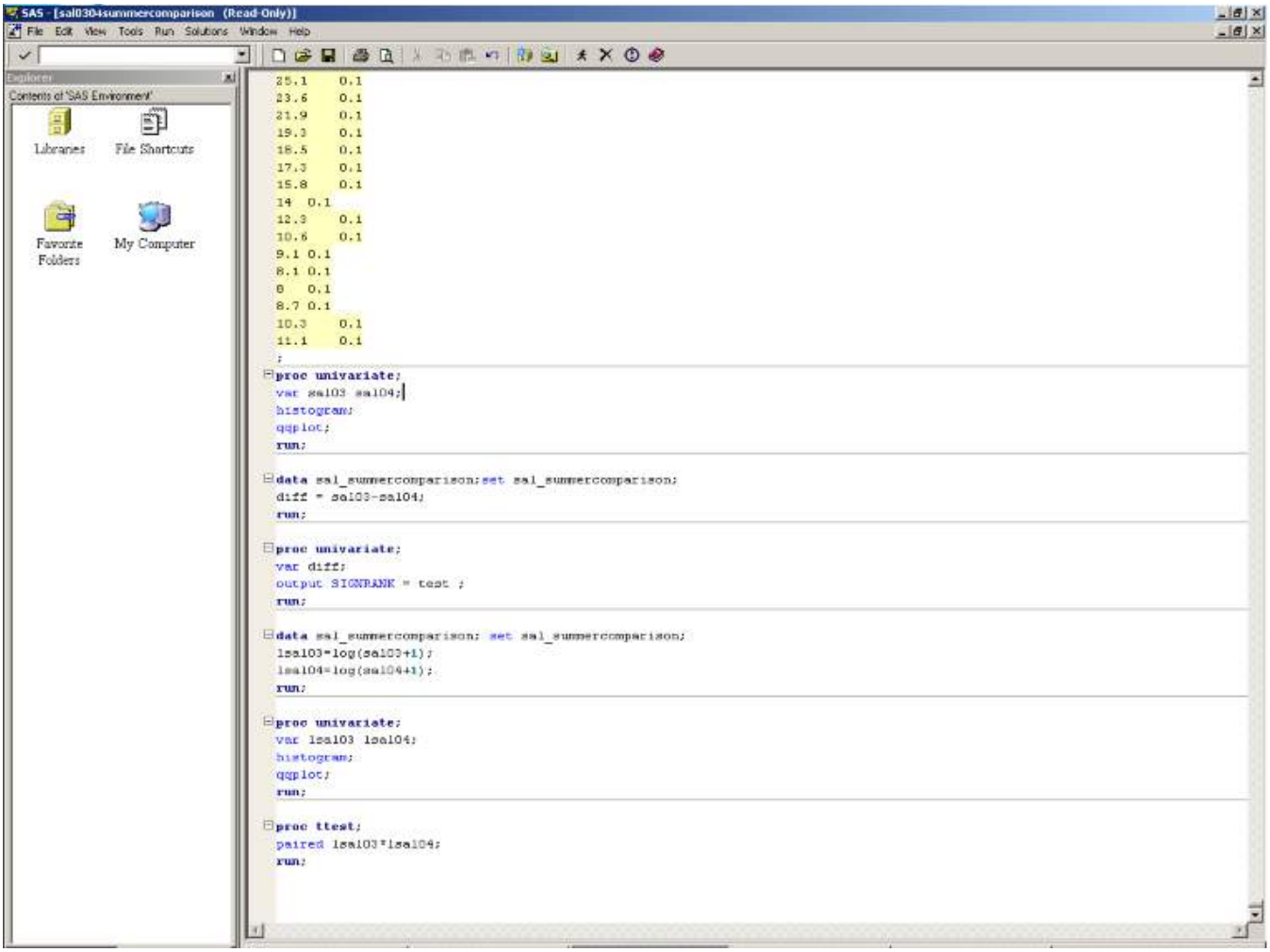
The SAS system is available for a wide range of different computers and operating systems and the way in which SAS programs are entered and run differs somewhat according to the computing environment. We describe the Microsoft Windows interface, as this is by far the most popular, although other windowing environments, such as X-windows, are quite similar.

SAS user interface



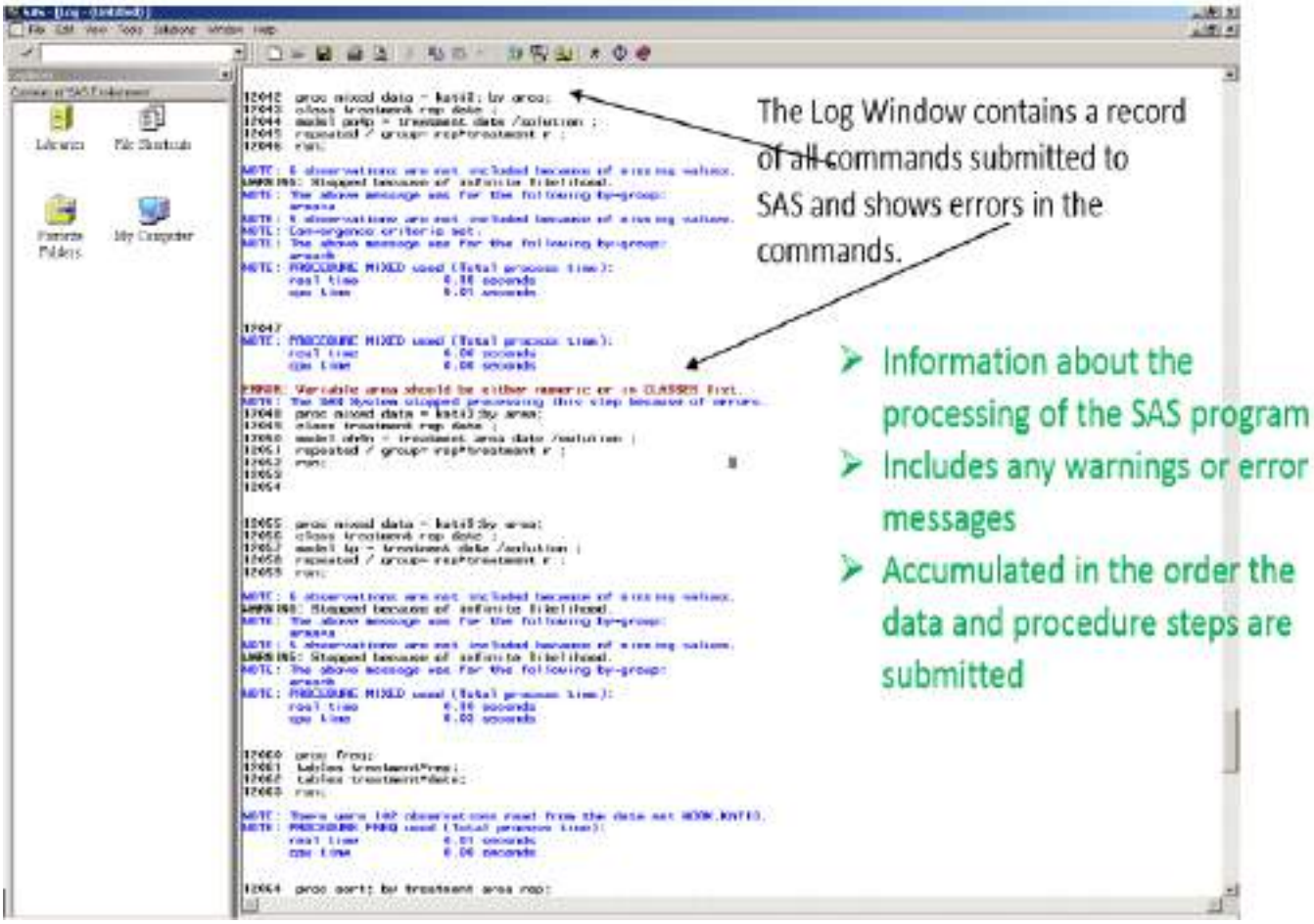
When SAS is started, there are five main windows open, namely the Editor, Log, Output, Results, and Explorer windows. The purpose of the main windows is as follows.

SAS Editor window

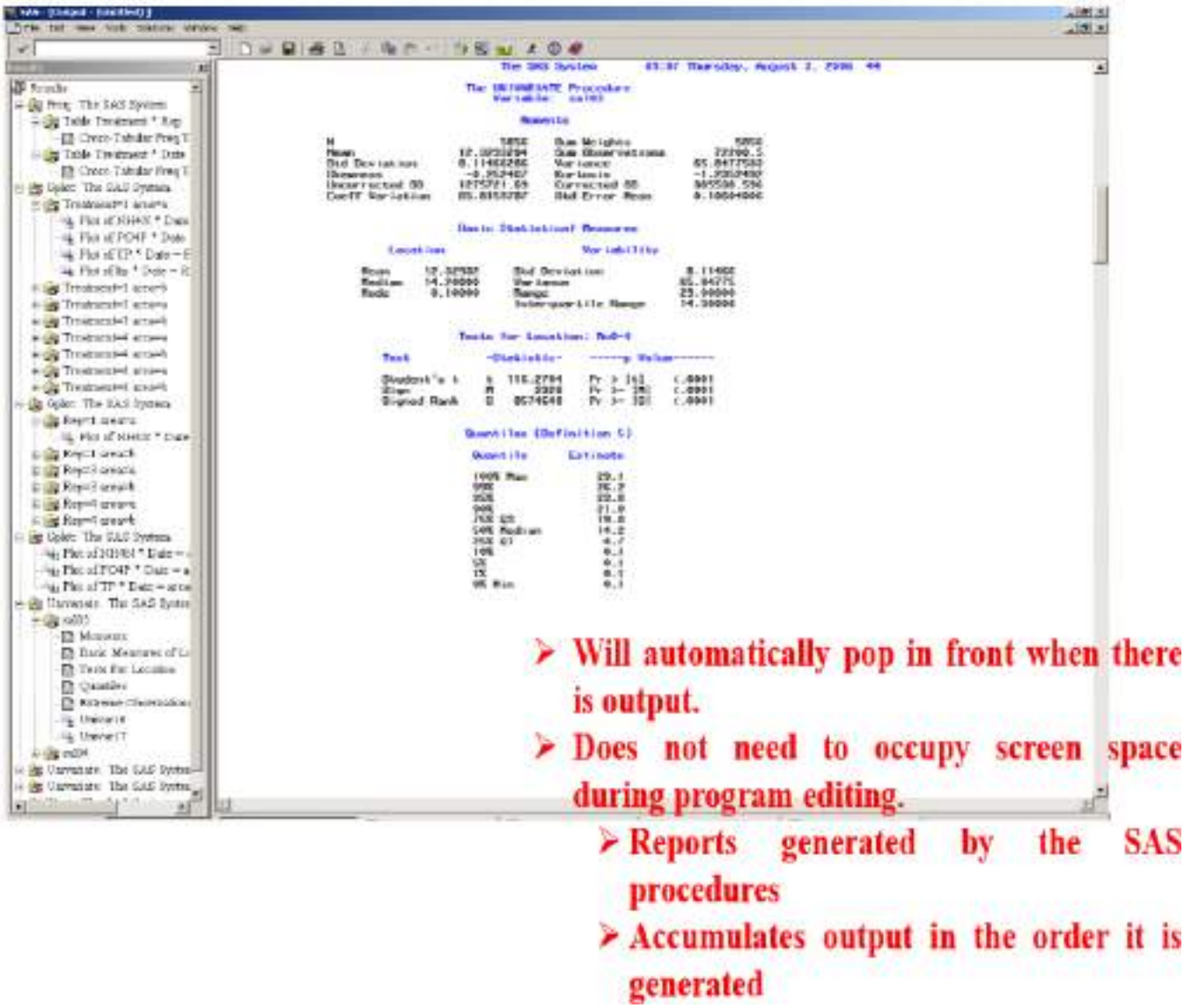


- Access and edit existing SAS programs
- Write new SAS programs
- Submitting SAS programs for execution
- Saving SAS programs

SAS Log Windows



SAS Output Windows



When graphical procedures are run, an additional window is opened to display the resulting graphs.

Managing the windows (e.g., moving between windows, resizing them, and rearranging them) can be done with the normal windows controls, including the Window menu. There is also a row of buttons and tabs at the bottom of the screen that can be used to select a window. If a window has been closed, it can be reopened using the View menu.

To simplify the process of learning to use the SAS interface, we concentrate on the Editor, Log, and Output windows and the most important and useful menu options, and recommend closing the Explorer and Results windows because these are not essential.

The editor windows

The editor is essentially a built-in text editor specifically tailored to the SAS language and with additional facilities for running SAS programs. Some aspects of the Editor window will be familiar as standard features of Windows applications. The File menu allows programs to be read from a file, saved to a file, or printed. The File menu also contains the command to exit from SAS. The Edit menu contains the usual options for cutting, copying, and pasting text and those for finding and replacing text.

The program currently in the Editor window can be run by choosing the Submit option from the Run menu. The Run menu is specific to the Editor window and will not be available if another window is the active window. Submitting a program may remove it from the Editor window. If so, it can be retrieved by choosing Recall Last Submit from the Run menu.

It is possible to run part of the program in the Editor window by selecting the text and then choosing Submit from the Run menu. With this method, the submitted text is not cleared from the Editor window. When running parts of programs in this way, make sure that a full step has been submitted. The easiest way to do this is to include a Run statement as the last statement.

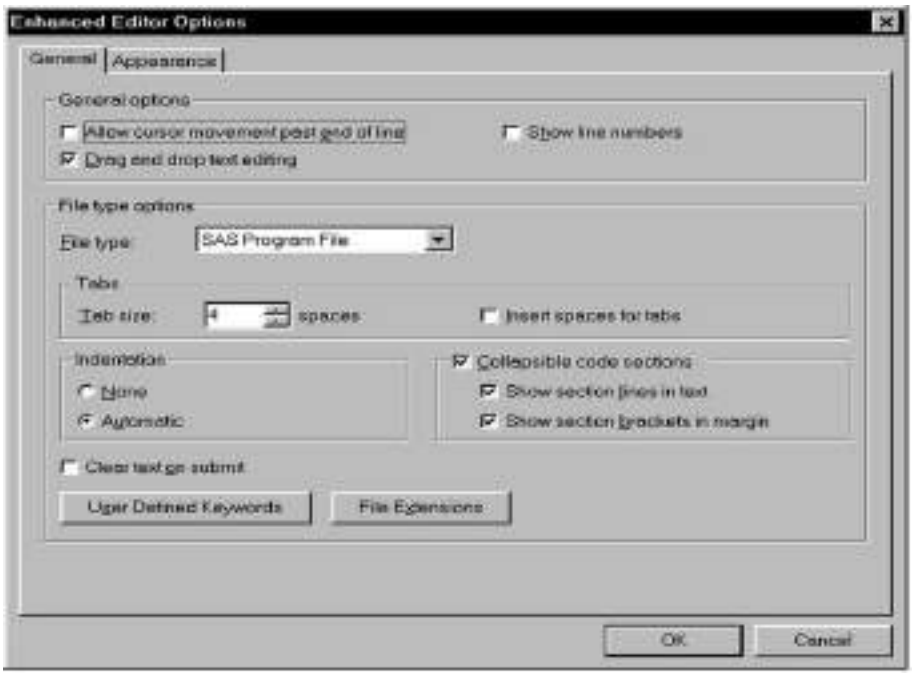
The Options submenu within Tools allows the editor to be configured. When the Enhanced Editor window is the active window (View, Enhanced Editor will ensure that it is), Tools; Options; Enhanced Editor Options will open a window.

The Log and Output Windows

The contents of the Log and Output windows cannot be edited; thus, several options of the File and Edit menus are disabled when these windows are active. The Clear all option in the Edit menu will empty either of these windows. This is useful for obtaining a “clean” printout if a program has been run several times as errors were being corrected.

Other Menus

The View menu is useful for reopening a window that has been closed. The Solutions menu allows access to built-in SAS applications but these are beyond the scope of this text.



The Help menu tends to become more useful as experience in SAS is gained, although there may be access to some tutorial materials if they have been licensed from SAS.

6.2. Overview of SAS

6.2.1. Accessing and Exiting SAS

The first thing you should do is find out how to access the SAS package for your course. This information will come from your instructor or system personnel, or from software documentation if you have purchased SAS to run on your own computer. If you are not running SAS on your own system, you will probably need a login name and a password to the computer system being used in your course. After you have logged on – provided a login and password to the computer system – you then access SAS. Perhaps the easiest method is to left-click or double left-click on the SAS icon on the desktop. Alternatively, you may use the Start button on the task bar in the lower left-hand corner of your screen.

To exit SAS, you can simply click on the close window symbol in the upper right-hand corner of the Display Manager window. You are then asked if you really want to terminate your SAS session. If you respond by clicking OK, then the Display Manager window closes and the SAS session ends. Of course, any programs, logs, or output produced during the SAS session are lost unless they have been saved. We subsequently discuss how to save such data so that it can be reused. Alternatively, you can use the menu command File → Exit or use Alt+F4 with the same results.

6.2.2. Getting Help

At times you may want more information about a command or some other aspect of SAS than this manual provides, or you may wish to remind yourself of some detail you have partially forgotten. SAS contains a convenient online help manual. There are several ways to access help. The simplest method is to click on the help button in the task bar.

6.2.3. SAS Programs

A SAS program typed into the Program Editor window is submitted for processing by clicking on the Submit button. A SAS program consists of SAS statements and sometimes data. Each statement must end in a semicolon. A SAS statement can be placed on more than one line. If a statement does not scan correctly as a valid SAS statement, then an error will occur and the program will not run. Broadly speaking the statements in a SAS program are organized in groups in two categories: data steps and procedures (hereafter referred to as proc's, as is standard when discussing SAS). Essentially the data steps are concerned with constructing SAS data sets that are then analyzed via various procedures. The data step typically involves inputting some data from a source, such as an external file, and then manipulating the data so that they are in a form suitable for analysis by a procedure. In an application we will have identified some SAS statistical procedures as providing answers to questions we have about the real-world phenomenon that gave rise to the data. There are many different SAS procedures, and they carry out a wide variety of statistical analyses. After a SAS procedure has analyzed the data, output is created. This may take the form of (a) actual output written in the Output window (b) data written to an external file, or (c) a temporary file holding a SAS data set.

Let us look at a very simple SAS program. Suppose we type the following commands in the Program Editor window:

```
data;
input x;
cards;
 1
 2
 3
 proc print;
var x;
```

```
run;
```

The data step consists of all the SAS statements starting with the line data; and ending with the line cards; the cards statement tells SAS that this data step is over. The word cards is a holdover from the days when programs were submitted to computers on punch cards. Alternatively, we can use the word data lines instead of cards. Immediately following the end of the data step are the data; in this case there are three observations, or cases, where each observation consists of one variable x. The value of x for the first observation is 1, for the second 2, and for the third 3. SAS constructs this data set and calls it work.data1. This is a default name until we learn how to name data sets. Immediately after the data comes the first procedure, in this case the proc print procedure. SAS knows that the actual data have ended when it sees a word like data or proc on a line. The procedure proc print consists of the line proc print; and the subsequent line var x;. The var statement tells proc print which variables to print in the just created SAS data set work.data. Since there is only one variable in this data set, there is really no need to include this statement. As we will see, we can also tell a proc which SAS data set to operate on; if we don't, the proc will operate on the most recently created data set by default. The run command tells SAS that everything above this line is to be executed. If no run command is given, the program does not execute although the program is still checked for errors and a listing produced in the Log window. After submitting the program, the Program Editor Window empties and a listing of the program together with comments is printed in the Log window displayed. We see from the Log window that the program ran, so we then check the Output window for the output from the program.

Suppose you submit your program and you have made an error. Because the Program Editor window empties after submission, you have to put the originally submitted program back into this window and correct it.

A SAS program can also contain comments. This is helpful when you have a long program and you want to be able to remind yourself later about what the program does and how it does it. Comments can be placed anywhere in a SAS program provided they start with /* and end with */. For example,

```
data; /* this is the data step */
input x;
cards;
1 /* this is the data */
```

```
2
2
proc print; /* this is a procedure */
 run;
```

6.2.4. SAS Data Steps

The first thing you have to know how to do is to get your data into the SAS program so that they can be analyzed. SAS uses data steps to construct SAS data sets. Several SAS data sets can be constructed in a single SAS program by including several data steps. These data sets can be combined to form new SAS data sets, and they can be permanently stored in computer files so that they can be accessed at a later date. The general form of the data step is

```
data name;
 statements
cards;
```

Where name corresponds to a name we give to the SAS data set being constructed and statements is a set of SAS statements. These statements typically involve reading data from observations from some source and perhaps performing various mathematical operations on the data to form new variables. SAS expects to read data from some source whenever there is an input statement included in statements. For example, the default method of supplying the data to an input statement is to include the data in the program immediately after the cards statement. If we are not going to use this method, and it is often inconvenient because it may involve a lot of error-prone typing, then we must tell SAS where to find the data. We don't have to supply name, but if we don't, a default name is assigned, with the first unnamed SAS data set in the program called work.data1, the second called work.data2, and so on. In general, it is better to name a data set with some evocative name so that you can remember what kind of data it contains. The value given to name must conform to certain rules, it must be a valid SAS name. A SAS name must begin with a letter or underscore _, can consist of letters, numbers, and the underscore _, and can be no longer than eight characters. For example, x1, ab_c, and lemon are all valid SAS names.

6.2.5. SAS Constants, Variables, and Expressions

There are two types of SAS constants. A numeric constant is simply a number that appears in a SAS statement. Numeric constants can use a decimal point, a minus sign, and scientific notation. For example, 1,1.23,01, −5,1.2E23,0.5E−10 are all valid constants that can appear in a SAS

program. A character constant consists of 1 to 200 characters enclosed in single quotes. For example,

```
data;
name='tom';
cards;
proc print;
```

creates a SAS data set with a single observation with one variable called name that takes the value tom.

SAS variables are given SAS names, as described in the above. When several variable names all have the same beginning but differ by a final number that increases by 1, such as in x1, x2, ... , x100, then the entire list may be referenced as a group, called a range list, as inx1-x100. This can be a significant convenience, as we avoid having to type in long lists of variables when entering a SAS program. SAS variables are either numeric or character in type. The type is determined by an input statement or an assignment statement. Numeric variables take real number values.

A SAS expression is a sequence of constants, variables, and operators that determines a value. For example, in the statement

```
x=y+z;
```

the numeric variable x is formed by adding the numeric variables y and z in the arithmetic expression y+z. The expression

```
x<y;
```

takes the value 1 when the variable x is less than y and the value 0 otherwise; i.e., it takes the value 1 when the logical expression x<y is true and the value 0 otherwise.

- SAS data set:
  - Specifically, structured file that contains data values.
  - File extension - .sas7bdat
  - Rows and Columns format – similar to Excel
    - Columns – variables in the table corresponding to fields of data
    - Rows – single record or observation
  - Located in SAS Data Libraries

Input

We now describe more fully how to input data using the input statement. We identify three

methods: from observations placed in the program, from external text files, and from other SAS data sets.

Data Input from the Program

Suppose each observation consists of three variables, y = weight, x1 = height, and x2 = average number of calories consumed daily. Suppose we have four observations given by

```
160 66 400
152 70 500
180 72 4500
240 68 7000
```

The statements

```
data example;
input y x1 x2;
cards;
160 66 400
152 70 500
180 72 4500
240 68 7000
;
Run;
```


create a SAS data set named example, containing four observations, each having three variables y, x1, and x2.

Data Input from a Text File

Suppose we have previously created a text file called C:\datafile.txt, where we have provided its full pathname, and suppose it contains the data

```
160 66 400
152 70 500
180 72 4500
240 68 7000
```

To access this file for input we use the infile statement. The program

```
data example;
infile ’C:\datafile’;
```

```
input y x1 x2;
cards;
```

reads these data into the SAS data set example and creates the SAS data set example.

Data Input from a SAS Data Set

Suppose we have a SAS program that creates a number of distinct data sets, and we want to combine the data sets to form a larger SAS data set. We do this using the set statement. For example, if the file C:\stuff.txt contains

```
1  2
3  4
```

then the SAS program

```
data one;
infile 'C:\stuff.txt';
input x y;
cards;
data two;
input x y;
cards;
5  6
data three;
set one two;
proc print data = three;
```

creates a new SAS data set called three that is the concatenation of one and two, i.e., the observations of one followed by the observations of two. The SAS data set three contains two variables x and y and three observations.

We can also use set to construct a data set consisting of a subset of the observations in a data set. The program

```
data one;
infile 'C:\stuff.txt';
input x y;
cards;
data two;
```

set one;

if y = 2;

Creates a data set two from one by selecting only those observations for which y=2. Therefore, two has only one observation. This is called a sub setting if statement.

Note: to tell SAS that a variable is character, add a dollar sign ($) after the name of the variable in the input statement.

Output

Output to the Output Window

After we have input data we commonly want to print it to make sure that we have input the data correctly. In fact, this is recommended. We have already seen that proc print can be used to print data records in the Output window. The output of most proc's are recorded in the Output window.

Output to the Log Window

The **put** statement can be used to write in the Log window or to an external file. To write in the Log window, the **put** statement occurs in a data step without a file statement. For example,

```
data example;
input x y;
z = x + y;
put 'x =' x 'y =' y 'z =' z;
cards;
1   2
3   4
```

writes

```
x=1 y=2 z=3
 x=3 y=4 z=7
```

In the Log window. Note that we can output characters by enclosing them in single quotes.

Control Statements

There are a number of statements that control the flow of execution of statements in the data step.

IF-THEN-ELSE

If-then-else statements are used to conditionally execute a SAS statement. They take the form if expression then statement1; else statement2; where statement1 is executed if expression is true, statement2 otherwise. The else part is optional, and if it is left out, control passes to the first

statement after the if-then statement when expression is false. For example,

```
data example;
input x $ y;
if x eq 'blue' then z = 1; else z = 0;
cards;
red 1
blue 2
```

Compares x to blue after input. A new variable, z, is set equal to 1 when x equals blue, otherwise z is set equal to 0. See Appendix A for a listing of all comparison and logical operators that can be used in such statements.

GOTO and RETURN

A goto statement tells SAS to jump immediately to another statement in the same data step and begin executing statements from that point. For example,

```
data info;
input x y;
if 1<=x then goto OK;
x = 3;
OK: return;
cards;
```


checks to see if the input value of x is greater than or equal to 1; if it is not, then x is set equal to 3; if it is then the SAS program jumps to the statement labelled OK. This is a return statement which tells SAS to begin processing a new observation.

STOP

The stop statement stops processing a SAS data step. The observation being processed when the stop statement is encountered is not added to the data set and processing resumes with the first statement after this data step. For example, in

```
data example;
input x y z;
if x = 2 then stop;
cards;
```

Stops building the data set when a value of x=2 is encountered.

SELECT-OTHERWISE

The select-otherwise statement replaces a sequence of if-then-else statements. The select statement takes the form:

```
select (expression);
when (expression1) statement1;
when (expression2) statement2;
.
.
.
otherwise statement;
end;
```

In this group of statements, SAS compares expression to expressioni. If they are equal, then statementi is executed. If none are equal, then statement is executed. The otherwise statement is optional. An end statement ends a select group.

6.2.6. Modifying SAS Data

As well as creating a SAS data set, the data step can also be used to modify the data in a variety of ways.

6.2.6.1. Creating and Modifying Variables

The assignment statement can be used both to create new variables and modify existing ones. SAS has the normal set of arithmetic operators: +, -, / (divide), * (multiply), and ** (exponentiation), plus various arithmetic, mathematical, and statistical functions. The result of an arithmetic operation performed on a missing value is itself a missing value. When this occurs, a warning message is printed in the log. Missing values for numeric variables are represented by a period (.) and a variable can be set to a missing value by an assignment statement such as: age =.;

To assign a value to a character variable, the text string must be enclosed in quotes; for example:

```
team='green';
```

A missing value can be assigned to a character variable as follows:

```
Team='';
```

To modify the value of a variable for some observations and not others, or to make different modifications for different groups of observations, the assignment statement can be used within an if then statement.

```
reward=0;
if weightloss > 10 then reward=1;
```

If the condition weigtloss > 10 is true, then the assignment statement reward=1 is executed; otherwise, the variable reward keeps its previously assigned value of 0. In cases like this, an else statement could be used in conjunction with the if then statement.

```
if weightloss > 10 then reward=1; else reward=0;
```

The condition in the if then statement can be a simple comparison of two values. The form of comparison can be one of the following:

Operator		*Meaning*	*Example*
EQ	=	Equal to	a = b
NE	~=	Not equal to	a ne b
LT	<	Less than	a < b
GT	>	Greater than	a gt b
GE	>=	Greater than or equalto	a >= b
LE	<=	Less than or equal to	a le b

Comparisons can be combined into a more complex condition using and (&), or (|), and not.

```
if team='blue' and weightloss gt 10 then reward=1;
```

In more complex cases, it may be advisable to make the logic explicit by grouping conditions together with parentheses. Some conditions involving a single variable can be simplified. For example, the following two statements are equivalent:

```
if age > 18 and age < 40 then agegroup = 1; if 18 < age < 40 then agegroup =
1;
```

and conditions of the form: x = 1 or x = 3 or x = 5 can

be abbreviated to x in(1, 3, 5) using the in operator.

If the data contain missing values, it is important to allow for this when recoding. In numeric comparisons, missing values are treated as smaller than any number. For example,

```
if age >= 18 then adult=1;
else adult=0;
```

Would assign the value 0 to adult if age was missing, whereas it may be more appropriate to assign a missing value. The missing function could be used do this, by following the else statement with:

```
if missing(age) then adult=.;
```

Care needs to be exercised when making comparisons involving character variables because these are case sensitive and sensitive to leading blanks. A group of statements can be executed conditionally by placing them between a do statement and an end statement:

```
If weightloss > 10 and weightnow < 140 then do; target=1;
reward=1; team =’blue’; end;
```

Every observation that satisfies the condition will have the values of target, reward, and team set as indicated. Otherwise, they will remain at their previous values.

Where the same operation is to be carried out on several variables, it is often convenient to use an array and an iterative do loop in combination. This is best illustrated with a simple example. Suppose we have 20 variables, q1 to q20, for which “not applicable” has been coded -1 and we wish to set those to missing values; we might do it as follows:

```
array qall {20} q1-q20; do i= 1 to 20;
if qall{i}=-1 then qall{i}=.;
end;
```

The array statement defines an array by specifying the name of the array, qall here, the number of variables to be included in braces, and the list of variables to be included. All the variables in the array must be of the same type, that is, all numeric or all character.

The iterative do loop repeats the statements between the do and the end a fixed number of times, with an index variable changing at each repetition. When used to process each of the variables in an array, the do loop should start with the index variable equal to 1 and end when it equals the number of variables in the array. The array is a shorthand way of referring to a group of variables. In effect, it provides aliases for them so that each variable can be referred to by using the name of the array and its position within the array in braces. For example, q12 could be referred to as qall{12} or when the variable i has the value 12 as qall{i}. However, the array only lasts for the duration of the data step in which it is defined.

6.2.6.2. Deleting Variables

Variables can be removed from the data set being created by using the drop or keep statements. The drop statement names a list of variables that are to be excluded from the data set, and the keep statement does the converse, that is, it names a list of variables that are to be the only ones retained

in the data set, all others being excluded. So the statement drop x y z; in a data step results in a data set that does not contain the variables x, y, and z, whereas keep x y z; results in a data set that contains only those three variables.

6.2.6.3. Deleting Observations

It may be necessary to delete observations from the data set, either because they contain errors or because the analysis is to be carried out on a subset of the data. Deleting erroneous observations is best done using the if then statement with the delete statement.

```
if weightloss > startweight then delete;
```

6.2.6.4. Subsetting Data Sets

If analysis of a subset of the data is needed, it is often convenient to create a new data set containing only the relevant observations. This can be achieved using either the subsetting if statement or the where statement. The subsetting if statement consists simply of the keyword if followed by a logical condition. Only observations for which the condition is true are included in the data set being created.

```
data men;
set survey;
if sex='M';
run;
```


The statement where sex='M'; has the same form and could be used to achieve the same effect. The difference between the subsetting if statement and the where statement will not concern most users, except that the where statement can also be used with proc steps, as discussed below. More complex conditions can be specified in either statement in the same way as for an if then statement.

6.2.6.5. Concatenating and Merging Data Sets

Two or more data sets can be combined into one by specifying them in a single set statement.

```
data survey;
set men women;
run;
```

This is also a simple way of adding new observations to an existing data set. First read the data for the new cases into a SAS data set and then combine this with the existing data set as follows.

```
data survey; set survey
```

```
newcases; run;
```

6.2.6.6. Merging Data Sets: Adding Variables

Data for a study can arise from more than one source, or at different times, and need to be combined. For example, demographic details from a questionnaire may need to be combined with the results of laboratory tests. To deal with this situation, the data are read into separate SAS data sets and then combined using a merge with a unique subject identifier as a key. Assuming the data have been read into two data sets, demographics and labtests, and that both data sets contain the subject identifier idnumber, they can be combined as follows:

```
proc sort data=demographics;
by idnumber;
proc sort
data=labtests;
by idnumber;
data combined;
merge demographics (in=indem) labtest (in=inlab);
by idnumber;
if indem and inlab;
run;
```


First, both data sets must be sorted by the matching variable idnumber. This variable should be of the same type, numeric or character, and same length in both data sets. The merge statement in the data step specifies the data sets to be merged. The option in parentheses after the name creates a temporary variable that indicates whether that data set provided an observation for the merged data set. The by statement specifies the matching variable. The subsetting if statement specifies that only observations having both the demographic data and the lab results should be included in the combined data set. Without this, the combined data set may contain incomplete observations, that is, those where there are demographic data but no lab results, or vice versa.

6.2.6.7. The Operation of the Data Step

In addition to learning the statements that can be used in a data step, it is useful to understand how the data step operates.

The statements that comprise the data step form a sequence according to the order in which they occur. The sequence begins with the data statement and finishes at the end of the data step and is

executed repeatedly until the source of data runs out. Starting from the data statement, a typical data step will read in some data with an input or set statement and use that data to construct an observation. The observation will then be used to execute the statements that follow. The data in the observation can be modified or added to in the process. At the end of the data step, the observation will be written to the data set being created. The sequence will begin again from the data statement, reading the data for the next observation, processing it, and writing it to the output data set. This continues until all the data have been read in and processed. The data step will then finish and the execution of the program will pass on to the next step.

In effect, then, the data step consists of a loop of instructions executed repeatedly until all the data is processed. The automatic SAS variable, _n_, records the iteration number but is not stored in the data set. Its use is illustrated in later chapters.

The point at which SAS adds an observation to the data set can be controlled using the output statement. When a data step includes one or more output statements an observation is added to the data set each time an output statement is executed, but not at the end of the data step. In this way, the data being read in can be used to construct several observations.

6.2.7. The proc Step

Once data have been read into a SAS data set, SAS procedures can be used to analyze the data. Roughly speaking, each SAS procedure performs a specific type of analysis. The proc step is a block of statements that specify the data set to be analyzed, the procedure to be used, and any further details of the analysis. The step begins with a proc statement and ends with a run statement or when the next data or proc step starts. We recommend including a run statement for every proc step.

The proc Statement

The proc statement names the procedure to be used and may also specify options for the analysis. The most important option is the data= option, which names the data set to be analyzed. If the option is omitted, the procedure uses the most recently created data set. Although this is usually what is intended, it is safer to explicitly specify the data set.

Many of the statements that follow particular proc statements are specific to individual procedures and are described in later chapters as they arise. A few, however, are more general and apply to a number of procedures.

The var Statement

The var statement specifies the variables that are to be processed by the proc step. For example:

```
proc print data=wghtclub;
var name team weightloss;
run;
```

restricts the printout to the three variables mentioned, whereas the default would be to print all variables.

The where Statement

The where statement selects the observations to be processed. The keyword where is followed by a logical condition and only those observations for which the condition is true are included in the analysis.

```
proc print data=wghtclub;
where weightloss > 0;
run;
```

The by Statement

The by statement is used to process the data in groups. The observations are grouped according to the values of the variable named in the by statement and a separate analysis is conducted for each group. To do this, the data set must first be sorted in the by variable.

```
proc sort data=wghtclub;
by team;
proc means;
var weightloss;
by team;
run;
```

The class Statement

The class statement is used with many procedures to name variables that are to be used as classification variables, or factors. The variables named can be character or numeric variables and will typically contain a relatively small range of discreet values. There may be additional options on the class statement, depending on the procedure.

6.2.8. SAS Procedures

There are a number of procedures to perform a specific task in SAS. Data can be read from one or more SAS data sets and uses them to build a new SAS data set by

6.2.8.1. SAS Summary Procedures

PROC PRINT:

- prints the observations in a SAS data set, using all or some variables
- Syntax/Expression

```
PROC PRINT data=...;
VAR…;
RUN;
```

- The var statement can be used to name the variables to be printed.

❖ If none is specified, then the last SAS data set created is printed. If no var statement is included, then all variables in the data set are printed, otherwise only those listed, and in the order in which they are listed are printed.

PROC SORT:

The procedure proc sort, sorts observations in a SAS data set by one or more variables, storing the resulting sorted observations in a new SAS data set or replacing the original.

- Syntax/Expression

```
PROC SORT DATA=... OUT=…;
BY…;
RUN;
```

- Specify the name of the data set to be sorted after the data= option.
- The OUT= option allows to put the newly sorted version of the data in a new data set. Without specifying this option, the original data set is replaced.

A by statement must be used with proc sort. Any number of variables can be specified in the by statement. The procedure proc sort first arranges the observations in the order of the first variable in the by statement, then it sorts the observations with a given value of the first variable by the second variable, and so on. By order, we mean increasing in value, or ascending order. If we want a by variable to be used in descending order, then the word descending must precede the name of the variable in the by list.

PROC CONTENTS:

- provides a description of a SAS dataset
- Syntax/Expression

```
PROC CONTENTS data=...;
RUN;
```

PROC UNIVARIATE

The proc univariate procedure is used to produce descriptive summary statistics for quantitative variables. proc univariate allows for many more descriptive statistics to be calculated. It includes mean, median, mode, standard deviation, skewness, kurtosis, quantiles, etc.

- Syntax/Expression

```
PROC UNIVARIATE DATA=…  <options>;
VAR variable1 variable2 variable3;
RUN;
```

- If the variable statement is not used summary statistics will be produced for all numeric variables in the input data set.

PROC MEANS

- It is similar to univariate procedure
- Syntax/Expression

```
PROC MEANS DATA=…  <options>;
<Optional SAS statements>;
RUN;
```

- With no options or optional SAS statements, the Means procedure will print out (for all numeric variables in the input data set)
  - ✓ the number of non-missing values,
  - ✓ mean,
  - ✓ standard deviation,
  - ✓ minimum, and maximum

PROC FREQ

The proc freq procedure produces frequency tables. A frequency table is table of counts of the values variables take. Frequency tables show the distribution of variable values and are primarily useful with variables where values are repeated in the data set.

- Syntax/Expression

PROC FREQ DATA=…;

TABLE variable1*variable2*variable3/<options>;

RUN;

- **Options**
  - LIST – prints cross tabulations in list format rather than grid
  - MISSING – specifies that missing values should be included in the tabulations
  - OUT=output_data_set – creates a data set containing frequencies, list format
  - NOPRINT – suppress printing in the output window
- Use BY statement to get percentages within each category of a variable
- **TABLE** statement options include:
  - **AGREE** – Tests and measures of classification agreement including McNemar's test, Bowker's test, Cochran's Q test, and Kappa statistics
  - **CHISQ** - Chi-square test of homogeneity and measures of association
  - **MEASURE** - Measures of association include Pearson and Spearman correlation, gamma, Kendall's Tau, Stuart's tau, Somer's D, lambda, odds ratios, risk ratios, and confidence intervals

Example, suppose the data set one contains ten observations of the categorical variable family corresponding to the number of members in a family. The program

data one;

input family;

cards;

2

3

1

5

3

2

4

6

1

2

```
proc freq data=one;
tables family;
run;
```

FAMILY	Frequency	Percent	Cumulative Frequency	Cumulative Percent
1	2	20.0	2	20.0
2	3	30.0	5	50.0
3	2	20.0	7	70.0
4	1	10.0	8	80.0
5	1	10.0	9	90.0
6	1	10.0	10	100.0

6.2.6.2. SAS Procedures for Data Analysis

PROC CORR

It is used to calculate the correlations between variables, Correlation coefficient measures the linear relationship between two variables, the Values Range from -1 to 1.

- Syntax/Expression

```
PROC CORR DATA=input_data_set <options>
VAR Variable1 Variable2;
With Variable3;
RUN;
```


PROC REG

PROC REG is used to fit linear regression models by least squares estimation. It is one of many SAS procedures that can perform regression analysis, Only continuous independent variables (Use GENMOD for categorical variables).

- Syntax/Expression

```
PROC REG DATA=…  <options>
MODEL dependent=independent1 independent2/<options>;
<optional statements>;
    RUN;
```

- PROC REG statement options includes
  - PCOMIT=m - performs principle component estimation with m principle components
  - CORR – displays correlation matrix for independent variables in the model MODEL statement options include

- SELECTION=Specifies a model selection procedure be conducted – FORWARD, BACKWARD, and STEPWISE
- ADJRSQ - Computes the Adjusted R-Square
- MSE – Computes the Mean Square Error
- CLB – computes confidence limits for parameter estimates
- ALPHA= Sets significance value for confidence and prediction intervals and tests

PROC GENMOD

PROC GENMOD is used to estimate linear models in which the response is not necessarily normal, Logistic and Poisson regression are examples of generalized linear models.

- Syntax/Expression

```
PROC GENMOD DATA=…;
CLASS independent1;
MODEL dependent = independent1 independent2/
 dist = <option>
 link=<option>;
run;
```

PROC ANOVA

The ANOVA procedure performs analysis of variance (ANOVA) for balanced data from a wide variety of experimental designs. In analysis of variance, a continuous response variable, known as a dependent variable, is measured under experimental conditions identified by classification variables, known as independent variables. The variation in the response is assumed to be due to effects in the classification, with random error accounting for the remaining variation. The ANOVA procedure is one of several procedures available in SAS/STAT software for analysis of variance. The ANOVA procedure is designed to handle balanced data (that is, data with equal numbers of observations for every combination of the classification factors), whereas the GLM procedure can analyze both balanced and unbalanced data. Because PROC ANOVA takes into account the special structure of a balanced design, it is faster and uses less storage than PROC GLM for balanced data. Use PROC ANOVA for the analysis of balanced data only, with the following exceptions: one-way analysis of variance, Latin square designs, certain partially balanced incomplete block designs, completely nested (hierarchical) designs, and designs with cell frequencies that are proportional to each other and are also proportional to the background

population. These exceptions have designs in which the factors are all orthogonal to each other. PROC ANOVA works for designs with block diagonal X'X matrices where the elements of each block all have the same value.

- Syntax/Expression

```
PROC ANOVA DATA=… <options>;
CLASS independent1 independent2;
MODEL dependent=independent1 independent2;
<optional statements>;
Run;
```

- Class statement must come before model statement, used to define classification variables.
- Useful PROC ANOVA statement option –Useful optional statement – MEANS independent1/<comparison type>
  - Used to perform multiple comparisons analysis
  - Set <comparison type> to:
    - TUKEY – Tukey's studentized range test
    - BON – Bonferroni t test
    - T – pairwise t tests
    - Duncan – Duncan's multiple-range test
    - Scheffe – Scheffe's multiple comparison procedure

6.2.9. SAS Graphical Procedures

One of the most informative ways of presenting data is via a graph. There are several methods for obtaining graphs in SAS.

Drawing Simple histogram;

```
PROC UNIVARIATE DATA=…;
VAR var1 var2;
 HISTOGRAM var1 var2;
run;
```

Drawing Simple boxplot;

```
PROC BOXPLOT DATA=…;
PLOT continus var*categorical var;
RUN;
```

Simple chart for categorical variables

PROC CHART

The procedure proc chart produces vertical and horizontal bar charts, and pie charts. These charts are useful for showing pictorially a variable's values or the relationships between two or more variables.

- Syntax/Expression

```
PROC CHART DATA=…;
VBAR var/ TYPE=percent;
HBAR var;
RUN;
```

PROC GCHART

The procedure proc gchart works just like proc chart but it plots to the Graph window, the plots have a more professional look, and there is much more control over the appearance of the plot.

- Syntax/Expression

```
PROC GCHART DATA=…;
VBAR var / TYPE=percent;
VBAR var;
PIE var / TYPE = percent;
 run;
```


PROC PLOT and PROC GPLOT

A scatterplot of two quantitative variables is a very useful technique when looking for a relationship between two variables. By a scatterplot we mean a plot of one variable on the y axis against the other variable on the x axis.

PROC PLOT

The proc plot procedure graphs one variable against another, producing a scatterplot. This procedure takes the values that occur for each observation in an input SAS data set on two variables, say x and y, and plots the values of (x, y), one for each observation.

- Syntax/expression;

```
PROC PLOT DATA=…;
PLOT var1*var2;
RUN;
```

The plot statement lists the plots to be produced. You may include many plot statements and specify many plot requests on one plot statement. The general form of the plot request is vertical * horizontal; first you name the variable to be plotted on the y axis, then a *, and then the variable to be plotted on the x axis. When a point on the plot represents a single observation, the letter A is used to represent this point. When a point represents two observations, the letter B is used, and so on. When a value of a variable is missing, that point is not included in the plot. Another form of the plot request is vertical*horizontal='character'. With this form, you are naming the variables to be plotted on the y and x axes and also specifying a character (inside single quotation marks) to be used to mark each point on the plot. A further form of the plot request is vertical*horizontal=variable, where the value of variable is now printed to mark each point. If you want to plot all combinations of one set of variables with another, you can use a grouping specification.

PROC GPLOT

The procedure proc gplot produces higher-resolution graphics. We have much more control over the appearance of the plot with this procedure.

- Syntax to Scatter plot with options;

```
TITLE1 "Plot Height versus Weight";
SYMBOL1 VALUE=dot;
PROC GPLOT DATA=…;
PLOT var1*var2=var3; *var3 should be categorical;
RUN;
```

The default plotting symbol is +. The symbol statement is used to define alternative plotting characters and control other characteristics such as whether or not we want to join the points. More than one symbol statement can appear (up to 99, labeled symbol1, symbol2, and so on) When more than one plot is requested, the appropriate symbol statement is referenced by using vertical*horizontal=n where n refers to the n^{th} symbol generated in the program (not necessarily the one labeled this). Actually a full description of the use of the symbol statement is reasonably complicated.

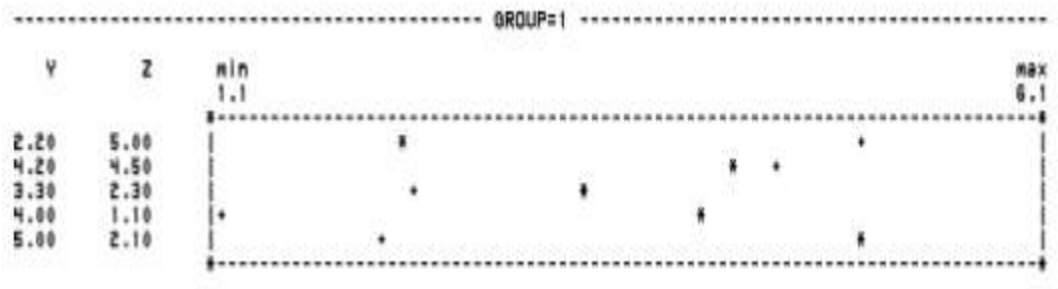
PROC TIMEPLOT

The procedure proc timeplot is used to plot time series. For example, the program

data one;

```
input group y z;
cards;
1 2.2 5.0
1 4.2 4.5
1 3.3 2.3
1 4.0 1.1
1 5.0 2.1
2 2.6 5.1
2 3.3 4.2
2 5.3 3.2
2 6.1 2.4
2 1.5 3.2
proc timeplot data=one uniform;
plot y =’*’ z =’+’ /overlay;
by group;
run;
```

creates two plots, one for each by group as specified by the value of the variable group. The uniform option to proc timeplot specifies that the horizontal scale, which is the scale for the variables, is the same for each plot and a time plot is given for variables y and z with these plots overlaid as specified by the overlay option to the plot statement. The time plot group=1 is given below. Notice that the vertical axis is time, and of course we are assuming here that the order in each group corresponds to the time order in which each observation was collected. Also the values of the variables are printed by the time axis. In the plot statement we list all the variables for which time plots are requested and here specify the plotting symbols, as in y =’*’ z =’+’, which indicates that each value of y is plotted with * and each value of z is plotted with +. As with proc means, a class statement can also be used with proc timeplot.



Lab Exercise 6

1) The following data refers to the seasonal wheat yield per acre at eight different locations, all having roughly the same quality soil. The relate the wheat yield at each location to the seasonal amount of rainfall and the amount of fertilizer used per acre.

RF(inches)	15.4	18.2	17.6	18.4	24	25.2	30.3	31
Fertilizer amount(pound/ acre)	100	85	95	140	150	100	120	80
Wheat yield	46.6	45.7	50.4	66.5	82.1	63.7	75.8	58.9

Write and save the above data as csv file (wheat.csv) in excel. Read these data using the INFILE statement to create a SAS data set called "wheat" also print out the resulting SAS data set?

2) Consider the following data on the gender and satisfaction of 20 randomly selected mekdela amba second year statistics department students with their gender f, m, m, f, m, f, m, f, m, f, m, m, f, m, f, f, m, m, f, f and their corresponding satisfaction level are: Y, Y, Y, N, Y, Y, Y, N, N, N, Y, Y, Y, Y, Y, Y, N, Y, N, Y, where f=female, m=male, Y=is yes for satisfied and N is no for unsatisfied students. Then
    a) Create SAS data set from the above variable
    b) Describe data using appropriate summary statistics
    c) Is there any association between gender of students and satisfaction on the department at 5% level of significance?
3) The weight of adults in a certain city has a mean of 60kg, with unknown standard Deviation. A random sample of 20 adults living in one of the Kebeles of the city was taken as data from sample given below:

Weight	45	60	68	67	50	52	51	53	63	47	89	52	41	52	56	58	57	63	69	61

Does this mean the residents of the kebele weight significantly less than 60kg at 1% level of significance? (Assume normal population for the weight of adults)

4) Consider the following data set on the strength of materials manufactured two branches of Company-Z:

Branch-1	1.26	0.34	0.70	1.75	0.57	1.55	0.08	0.42	0.50	3.20	0.15	0.49	0.95	0.24	1.37
Branch-2	2.37	2.16	14.82	1.73	41.04	0.23	1.32	2.91	39.41	0.11	27.44	4.51	0.51	4.50	0.18

a) Summarize the data by computing the appropriate summery statistics?

b) Construct comparative box-plot of strength of materials by branches and interpret it?

5) A researcher is interested in the effect of an approach to teaching graduate statistics on statistics anxiety. The statistics course offered by the Educational Psychology department is a lecture based course and a computer based course with no lectures. The content of both courses is exactly the same. There are twelve students in each class. At the end of the course students were asked to fill out the Statistics Anxiety Questionnaire. The results are presented below:

Lecture based approach	Computer based approach
10	27
23	24
11	15
17	19
7	17
4	21
18	26
11	17
11	20
14	29
10	27
19	22

Please enter the data into SAS and, Test the null hypothesis that the difference between the mean anxiety score of the students taking the lecture based course and the mean anxiety score of the students taking the computer based course is zero.

6) A real state exports wanted to find the relationship between the sell prices of houses and various characteristics of houses. She collected data on four variable recorded in the tables for 13 houses that were sold recently. The four variables are:

Price = sale price of house in thousands of

Lot size = size of lot in arcs

Living area = living area in square feet

Age = age of a house in year

Price	455	278	463	327	105	264	445	346	487	289	434	411	223
Lot size	1.4	0.9	1.8	0.7	2.6	1.2	2.1	1.1	2.8	1.6	3.2	1.7	0.5
Area	2500	2250	2900	1800	3200	2400	2700	2050	2850	2400	2600	2300	1700
Age	8	12	5	9	4	28	9	13	7	16	5	8	19

a) Indicate where you expect positive or negative relationship between response and each explanatory variable?

b) Write estimate require equation?

c) Explain the meaning of the estimated regression coefficients of all explanatory variables?

d) What are the value of standard error of the estimate, R^2 and adjusted R^2?

e) The correlation between response and explanatory variable.

Reference

This manual is based on the following manuals published by R and SAS Institute Inc

Dalgaard, P. (2002). Introductory Statistics with R. Springer-Verlag, New York

Rizzo M. L (2008). Statistical Computing with R. Chapman & Hall/CRC.

Maindonald, J. and Braum, J. (2003). Data Analysis and Graphics using R: An Example based Approach.Cambridge University Press.

Fox, J. (2002). An R and S-PLUS Companion to Applied Regression. Sage Books.

Becker R. A., Chambers J. M. and. Wilks A. R (1988). The New S Language. Chapman& Hall, New York.

Chambers J. M. and Hastie T. J. eds. (1992). Statistical Models in S. Chapman & Hall, New York. 8.

Chambers J. M. (1998). Programming with Data. A Guide to the S Language.

R Project, "What is R?" (http://www.r-project.org/about.html).

Base SAS 9.1.3 Procedures Guide (2nd Edition). Volumes 1-4, SAS publishing.

SAS Procedures Guide, Version 6, Third Edition.

SAS/STAT User's Guide Volume 1, Version 6, Fourth Edition.

SAS/STAT User's Guide Volume 2, Version 6, Fourth Edition.

SAS/GRAPH Software, Volume 1, Version 6, First Edition.

Appendix

A: Answers for selected exercise

Exercise 1

1.

a) [1] 32

b) [1] 84

c) [1] 116

2. ____

3.

a) [1] Inf

b) [1] 64

c) [1] 3.367573e+82

4. > X
[1] 20 21 34 56 23

> Z
[1] "m" "m" "p" "p" "o"

5. _____

6. > M1
[,1] [,2] [,3]

[1,] 15 14 -13

[2,] 13 17 12

[3,] 12 10 18

a) [1] 2492

b) [1] 3 3

c) [,1] [,2] [,3]

[1,] 15 13 12

[2,] 14 17 10

[3,] -13 12 18

d) [,1] [,2] [,3]
[1,] 0.07463884 -0.153290530 0.15609952

[2,] -0.03611557 0.170947030 -0.14004815

[3,] -0.02969502 0.007223114 0.02929374

e) Eigen values
[1] 29.846925 14.324350 5.828724

Eigen vectors
[,1] [,2] [,3]

[1,] -0.07795179 -0.6259299 0.7335896

[2,] -0.72151837 0.5466771 -0.6535590

[3,] -0.68799329 0.5561977 -0.1862980

7. > studentdata

	CGPA	Studyhour	Sex
1	2.40	5	M
2	2.60	4	M
3	2.83	6	F
4	3.60	8	F
5	3.25	7	M
6	2.75	3	M

Exercise 2

1. **> sample**

[1] 9 6 6 5 5

2.

a) [1] 0.006209665

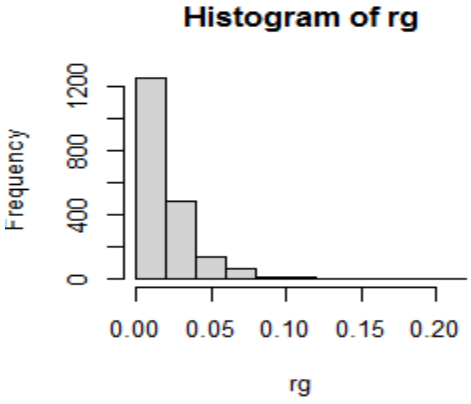
b) [1] 0.9937903

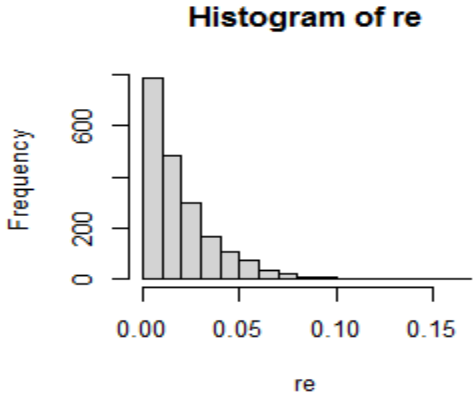
3. **> sample=rnorm(2000,20, 4)**

> sample

4.

> set.seed(12345)
> rg=rgamma(2000,shap=1,rate=50)
> re=rexp(2000,rate=50)
> par(mfrow=c(2,2))
> hist(rg)
> hist(re)





5. [1] 0.9998802

Exercise 3

```
1) > meanx=function(x)
   + {m=sum(x)/length(x)
   + return(m)
   + }
   > data=rnorm(100)
   > mean(data)
[1] 0.1296838
2)      > F1=function(x,y)
          + {x+y}
          > F1(3,4)
          [1] 7
3)  > twosam=function(y1,y2)
     {
     n1=length(y1);n2=length(y2)
     yb1=mean(y1);yb2=mean(y2)
      s1=var(y1);s2=var(y2)
      s=((n1-1)*s1+(n2-1)*s2)/(n1+n2-2)
      tst=(yb1-yb2)/sqrt(s*(1/n1+1/n2))
      return(c(n1,n2,yb1,yb2,s1,s2,s,tst))
     }
4)  > yz=function(a,b)
      {z=a*b
      y=b^a
      return(c(z,y))
      }
     > a=3
     > b=5
     > yz(a,b)
     [1] 15 125
5)  > MatrixA(y)
     [[1]]
        [,1] [,2] [,3]
     [1,]   10   15   69
     [2,]   11   18   78
     [3,]   12   25   89
     [[2]]
     [1] -54
     [[3]]
             [, 1]       [, 2]       [, 3]
     [1,] 6.444444 0.7962963 -1.0925926
     [2,] -7.222222 -1.1481481 1.2962963
     [3,] 1.333333 0.3888889 -0.2777778
```

Exercise 4

1. #display one way table for blood type

> table1

A AB B O

2 3 1 5

#display two way table for blood type by sex

> table2

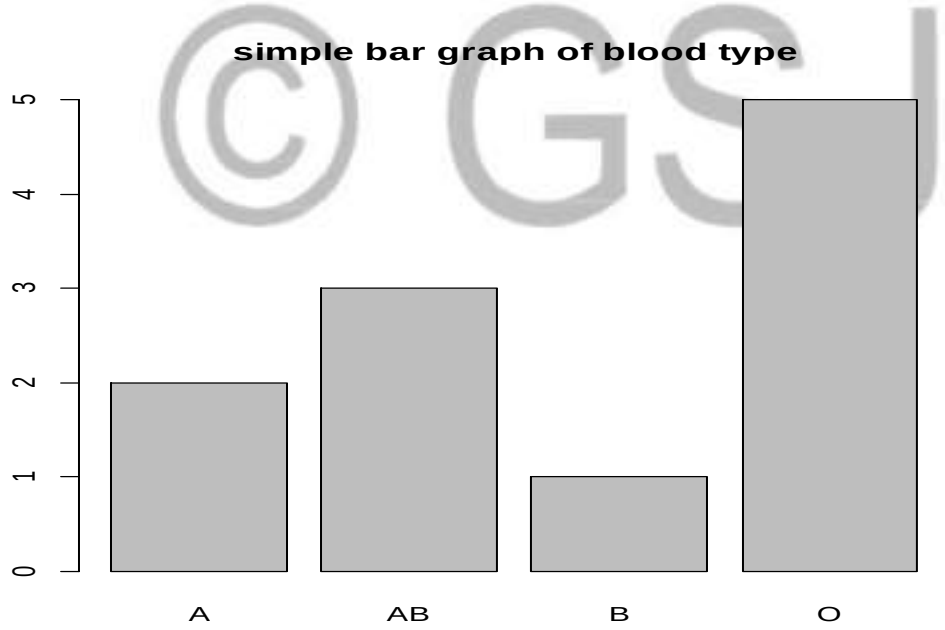
F M

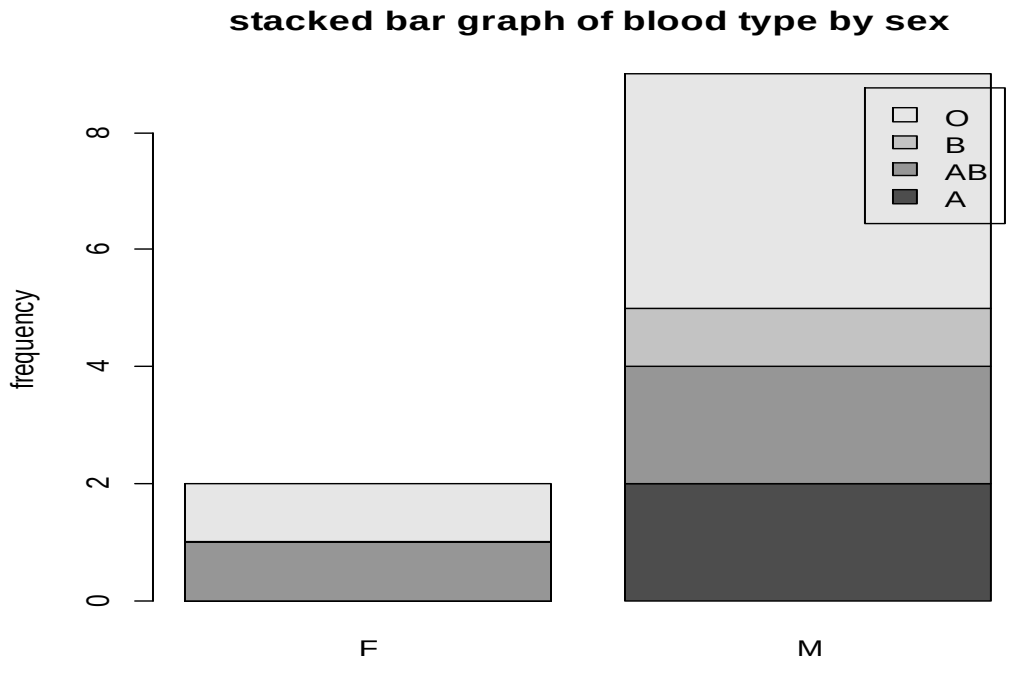
A 0 2

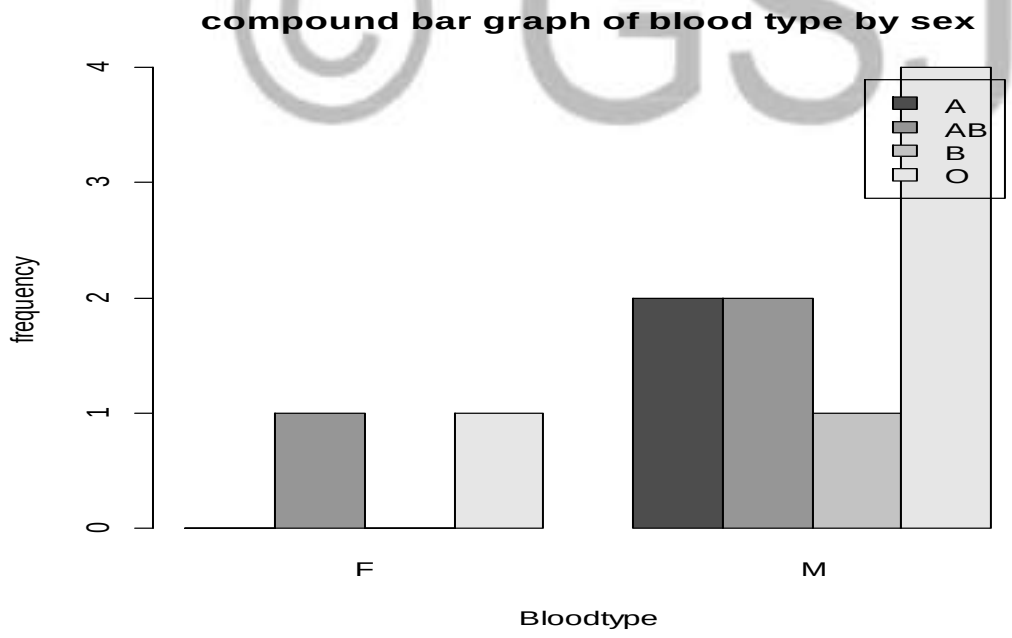
AB 1 2

B 0 1

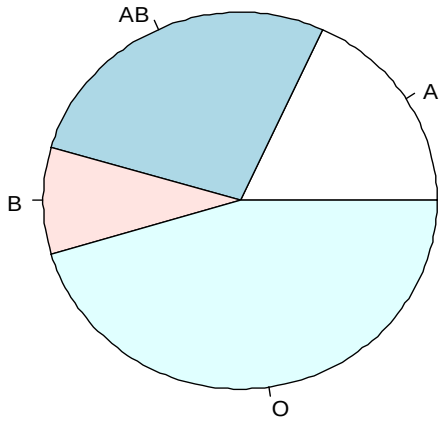
O 1 4



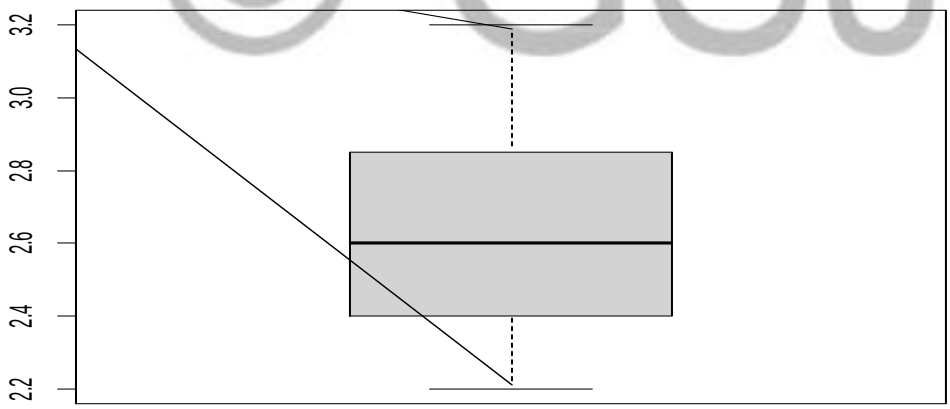

stacked bar graph of blood type by sex
frequency
0
2
4
6
8
O
B
AB
A
F
M
Bloodtype

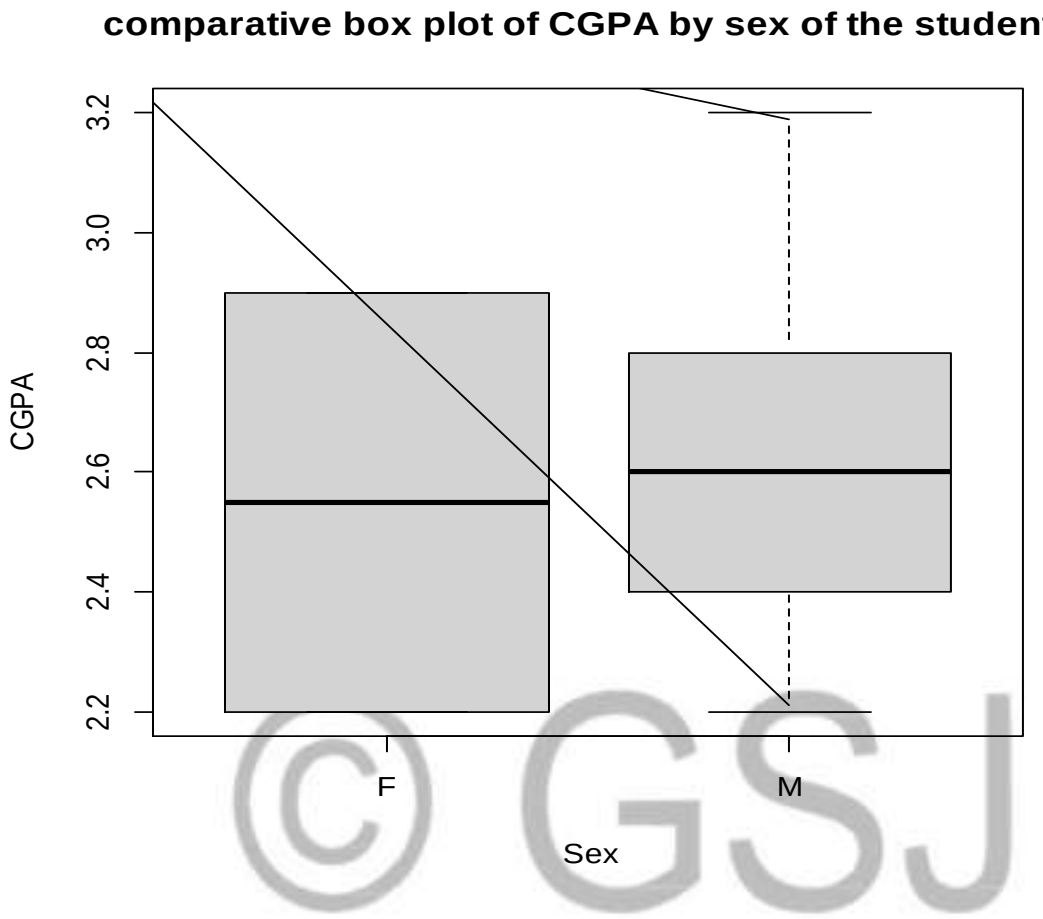

compound bar graph of blood type by sex
frequency
0
1
2
3
4
A
AB
B
O
F
M
Bloodtype

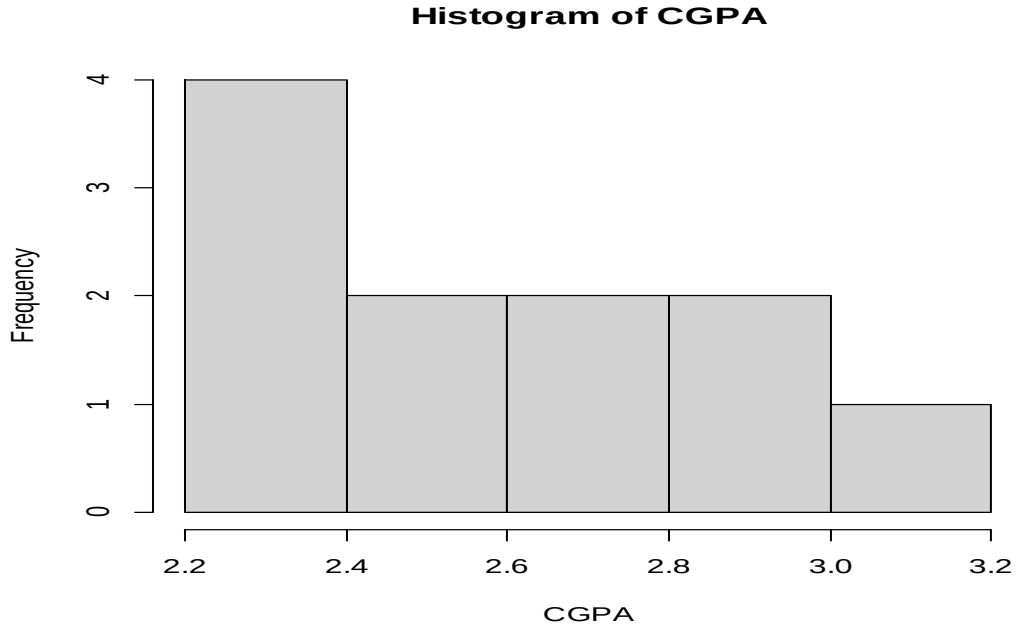
pie-chart of blood type



 # Box plot of students CGPA




comparative box plot of CGPA by sex of the students
CGPA
2.2
2.4
2.6
2.8
3.0
3.2
F
M
Sex


Histogram of CGPA
Frequency
0
1
2
3
4
2.2
2.4
2.6
2.8
3.0
3.2
CGPA

Exercise 5

1.

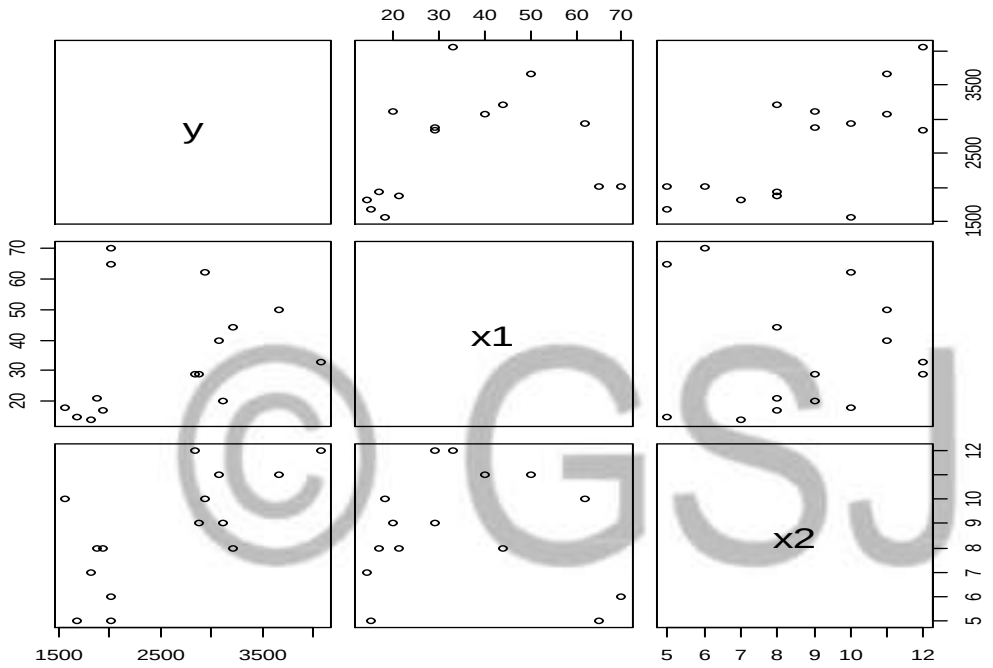
a) y x1 x2

y 1.0000000 0.26396212 0.69054069

x1 0.2639621 1.00000000 -0.08618751

x2 0.6905407 -0.08618751 1.00000000

Matrix scatter plot



b) Call:

lm(formula = y ~ x1 + x2, data = data)

Residuals:

Min	1Q	Median	3Q	Max
-1105.1	-322.1	-61.0	331.9	721.2

Coefficients:

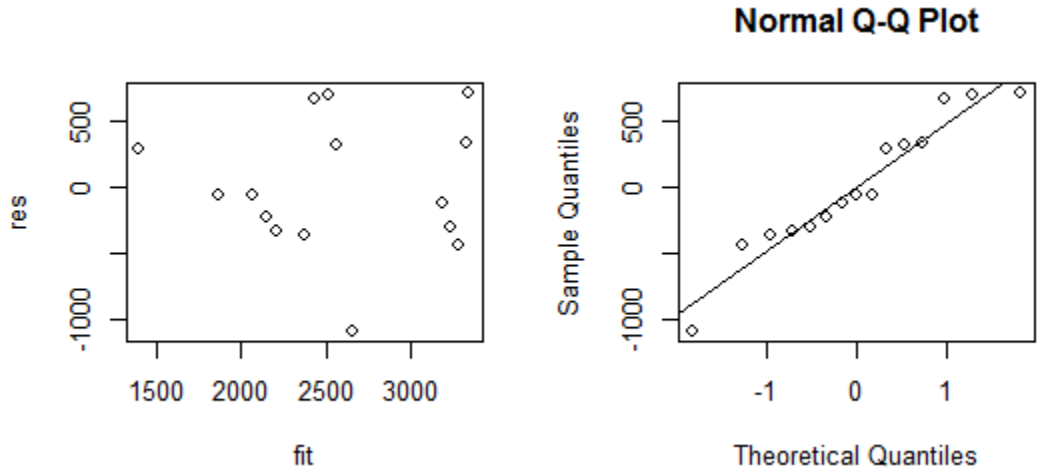
	Estimate	Std. Error	t value	Pr(>|t|)	
(Intercept)	-20.352	652.745	-0.031	0.97564	
x1	13.350	7.672	1.740	0.10738	
x2	243.714	63.512	3.837	0.00236	**

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 547.7 on 12 degrees of freedom

Multiple R-squared: 0.5823, Adjusted R-squared: 0.5126
F-statistic: 8.363 on 2 and 12 DF, p-value: 0.005314

c)



d) _____
e) _____
f) **1675.43**
2.

a) Call:
aov(formula = combin$patient ~ as.factor(combin$drugtype))

Terms:

	as.factor(combin$drugtype)	Residuals
Sum of Squares	816.1229	91.8171
Deg. of Freedom	2	18

Residual standard error: 2.258529

Estimated effects may be unbalanced

> summary(oneway)

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
as.factor(combin$drugtype)	2	816.1	408.1	80	1.11e-09 ***
Residuals	18	91.8	5.1		

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

b) TukeyHSD(oneway)
Tukey multiple comparisons of means
95% family-wise confidence level
Fit: aov(formula = combin$patient ~ as.factor(combin$drugtype))
$`as.factor(combin$drugtype)`

	diff	lwr	upr	p adj

2-1 11.814286 8.7332265 14.895345 0.000000
3-1 14.285714 11.2046551 17.366774 0.000000
3-2 2.471429 -0.6096306 5.552488 0.129646

3. Pearson's Chi-squared test
data: performance
X-squared = 38.151, df = 4, p-value = 1.043e-07

4. ____

5. > #coding number for A B C D is equal to(1,2,3,4) respectively

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
as.factor(locationtype)	3	133.5	44.5	178.0	2.99e-06	***
as.factor(car)	3	2850.5	950.2	3800.7	3.18e-10	***
as.factor(tirestype)	3	645.5	215.2	860.7	2.72e-08	***
Residuals	6	1.5	0.2			

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

Exercise 6

1. ____

2.

a) ___

b)

The FREQ Procedure

Frequency Percent Row Pct Col Pct	Table of gender by satisfaction		
	satisfaction		
gender	n	y	Total
f	4 20.00 40.00 66.67	6 30.00 60.00 42.86	10 50.00
m	2 10.00 20.00 33.33	8 40.00 80.00 57.14	10 50.00
Total	6 30.00	14 70.00	20 100.00

c)

Statistics for Table of gender by satisfaction

Statistic	DF	Value	Prob
Chi-Square	1	0.9524	0.3291
Likelihood Ratio Chi-Square	1	0.9663	0.3256
Continuity Adj. Chi-Square	1	0.2381	0.6256
Mantel-Haenszel Chi-Square	1	0.9048	0.3415
Phi Coefficient		0.2182	
Contingency Coefficient		0.2132	
Cramer's V		0.2182	
WARNING: 50% of the cells have expected counts less than 5. Chi-Square may not be a valid test.			

Fisher's Exact Test	
Cell (1,1) Frequency (F)	4
Left-sided Pr <= F	0.9296
Right-sided Pr >= F	0.3142
Table Probability (P)	0.2438
Two-sided Pr <= P	0.6285

3.

The TTEST Procedure

Variable: weight

N	Mean	Std Dev	Std Err	Minimum	Maximum
20	57.7000	10.6480	2.3810	41.0000	89.0000

Mean	99% CL Mean		Std Dev	99% CL Std Dev	
57.7000	50.8882	64.5118	10.6480	7.4722	17.7414

DF	t Value	Pr > |t|
19	-0.97	0.3462

4.

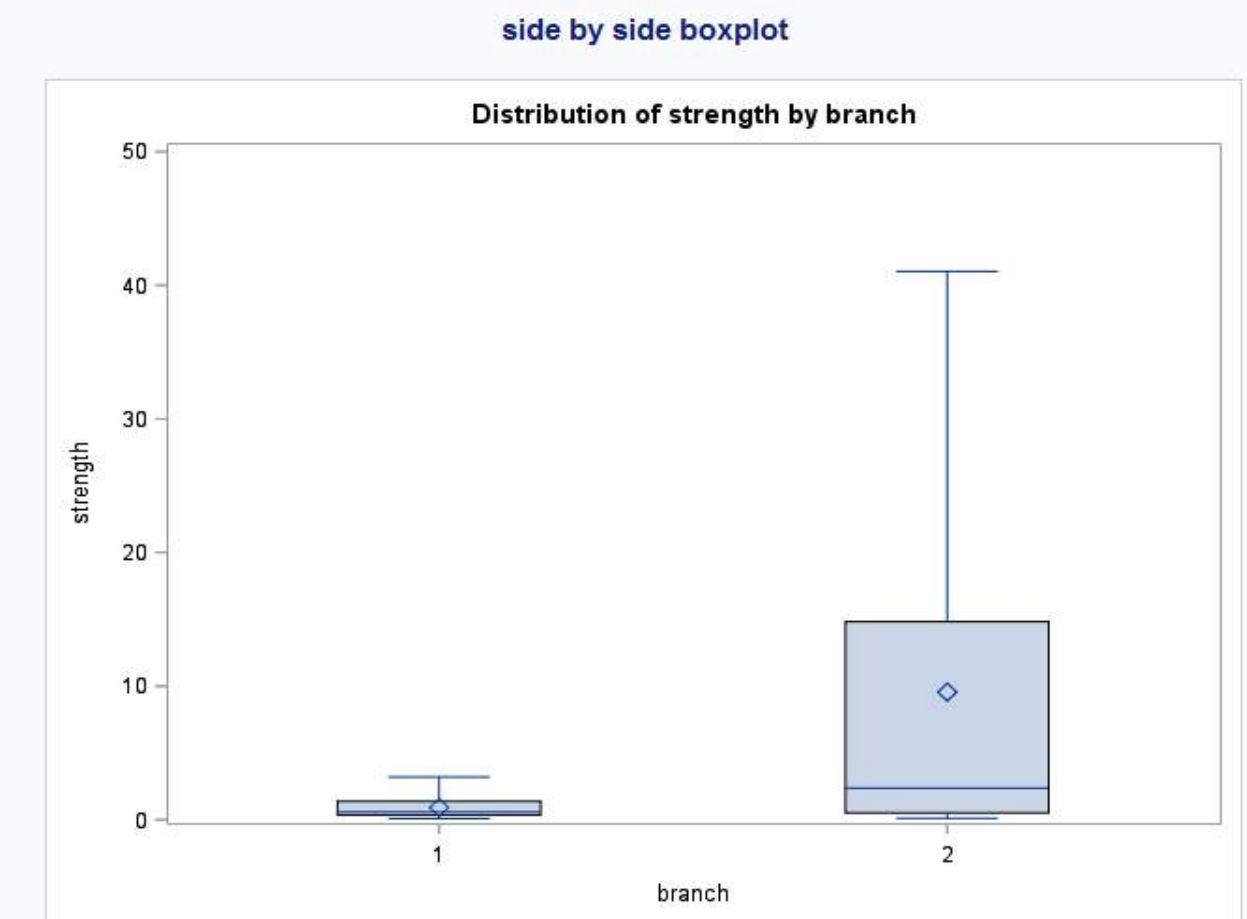
a)

The MEANS Procedure

Analysis Variable : strength

N	Mean	Std Dev	Minimum	Maximum
30	5.2270000	10.9490337	0.0800000	41.0400000

b)



5.

The GLM Procedure

Dependent Variable: anxiety

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	495.041667	495.041667	19.56	0.0002
Error	22	556.916667	25.314394		
Corrected Total	23	1051.958333			

R-Square	Coeff Var	Root MSE	anxiety Mean
0.470591	28.81914	5.031341	17.45833

Source	DF	Type I SS	Mean Square	F Value	Pr > F
approach	1	495.0416667	495.0416667	19.56	0.0002

Source	DF	Type III SS	Mean Square	F Value	Pr > F
approach	1	495.0416667	495.0416667	19.56	0.0002

Parameter	Estimate		Standard Error	t Value	Pr > |t|
Intercept	22.00000000	B	1.45242309	15.15	<.0001
approach 1	-9.08333333	B	2.05403643	-4.42	0.0002
approach 2	0.00000000	B	.	.	

6.

The REG Procedure
Model: MODEL1
Dependent Variable: price

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	3	30841	10280	0.74	0.5528
Error	9	124484	13832		
Corrected Total	12	155324			

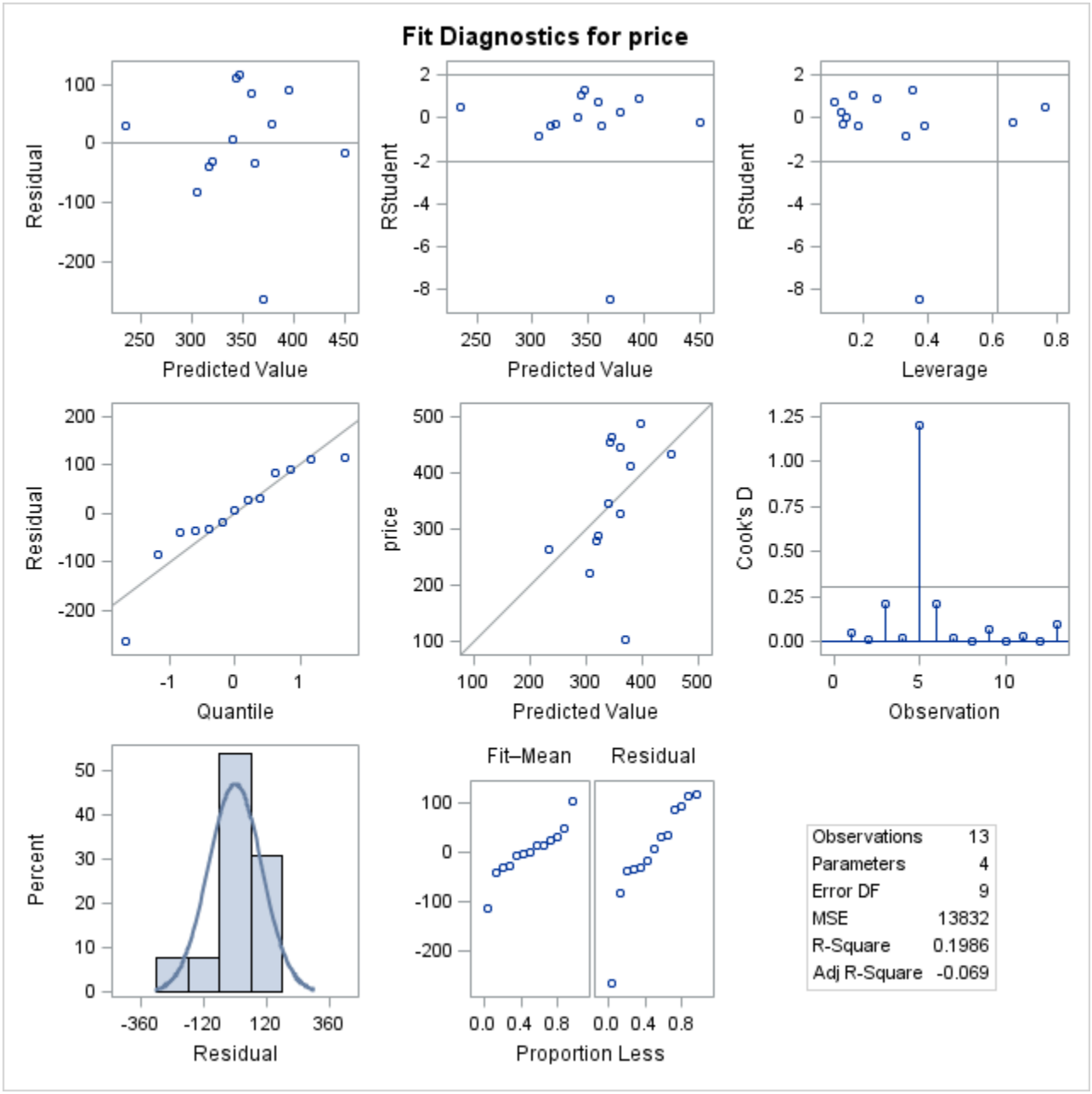
Root MSE	117.60752	**R-Square**	0.1986
Dependent Mean	348.23077	**Adj R-Sq**	-0.0686
Coeff Var	33.77287		

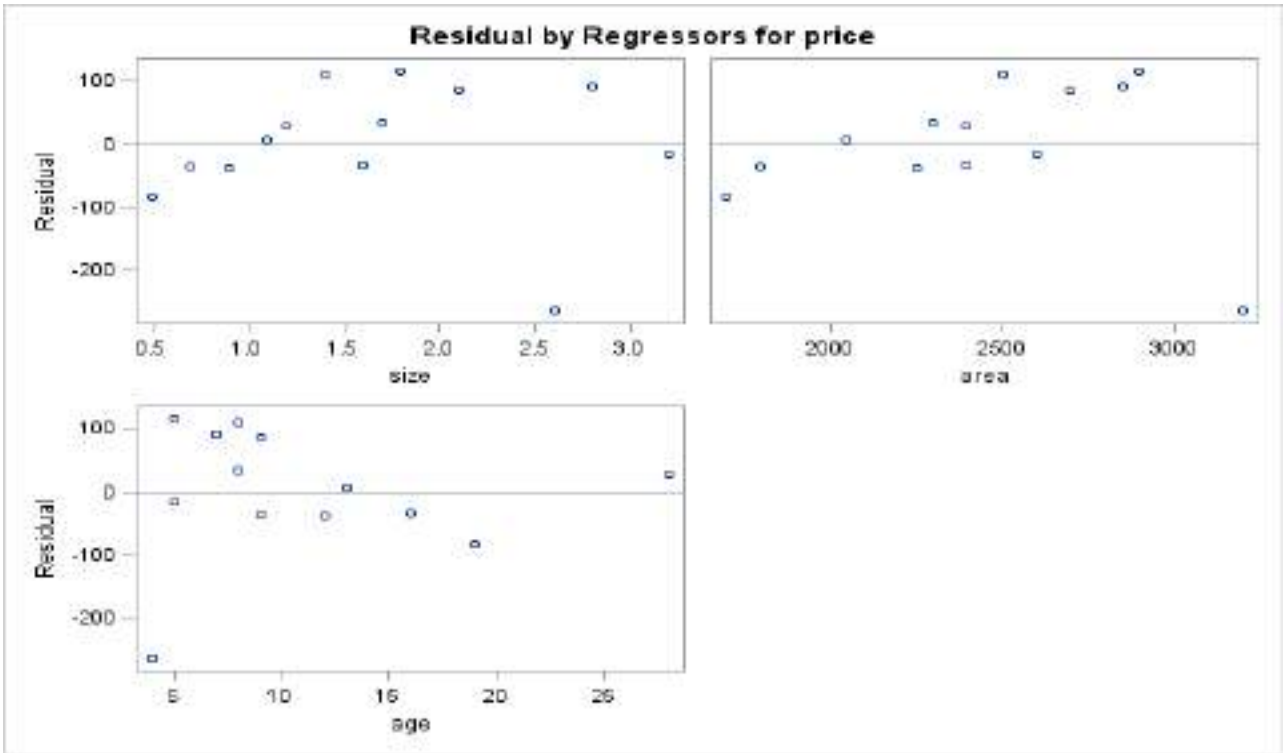
Parameter Estimates

Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	**1**	529.73329	277.61802	1.91	0.0887
size	**1**	55.32860	73.80388	0.75	0.4726
area	**1**	-0.08821	0.13360	-0.66	0.5256

Parameter Estimates

Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|
age	**1**	-5.33426	6.19114	-0.86	0.4113





The CORR Procedure

4 Variables:	price size area age

Simple Statistics

Variable	N	Mean	Std Dev	Sum	Minimum	Maximum
price	13	348.23077	113.77035	4527	105.00000	487.00000
size	13	1.66154	0.82718	21.60000	0.50000	3.20000
area	13	2435	430.30252	31650	1700	3200
age	13	11.00000	6.74537	143.00000	4.00000	28.00000

Pearson Correlation Coefficients, N = 13
Prob > |r| under H0: Rho=0

	price	size	area	age
price	1.00000	0.31667 0.2918	0.14902 0.6270	-0.38223 0.1974
size	0.31667 0.2918	1.00000	0.80592 0.0009	-0.57949 0.0379
area	0.14902 0.6270	0.80592 0.0009	1.00000	-0.50100 0.0812
age	-0.38223 0.1974	-0.57949 0.0379	-0.50100 0.0812	1.00000