Chapter 1

1. Introduction to R

Objectives

After completing this unit, students will be able to:

- Understand the very basics of the R languages
- Understand various objects in R
- Identify Various data types
- How to search for help.
- Perform matrix algebra in R,
- Enter, import, export data using R

1.1. The R environment

R is an integrated suite of software facilities for data manipulation, calculation and graphical display. Among other things it has

- an effective data handling and storage facility,
- a suite of operators for calculations on arrays, in particular matrices,
- a large, coherent, integrated collection of intermediate tools for data analysis,
- graphical facilities for data analysis and display either directly at the computer or on hard-copy, and
- a well-developed, simple and effective programming language (called 'S') which includes conditionals, loops, user defined recursive functions and input and output facilities.

Why R?

- It's free, open-source software
- It runs on many operating systems: Windows, Unix and (Mac) OS X.
- It provides an supreme platform for programming new statistical methods in an easy and straightforward manner.
- It has advanced graphics capabilities.
- It can work on objects of unlimited size and complexity with a consistent, logical expression language;

- It is supported by comprehensive technical documentation and user contributed tutorials.
- There are also several good textbooks on statistical methods that use R for illustration.
- It stimulates critical thinking about problem-solving rather than a "push the button" mentality.

1.2. Getting started with R and the online help system

R Overview

You can enter commands one at a time at the command prompt **(>)** or run a set of commands from a source file. There is a wide variety of data types, including vectors (numerical, character, logical), matrices, data frames, and lists.

To quit R, use **q( )**

Most functionality is provided through built-in and user-created functions and all data objects are kept in memory during an interactive session.

Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed

Using R

There are several ways to work with R:

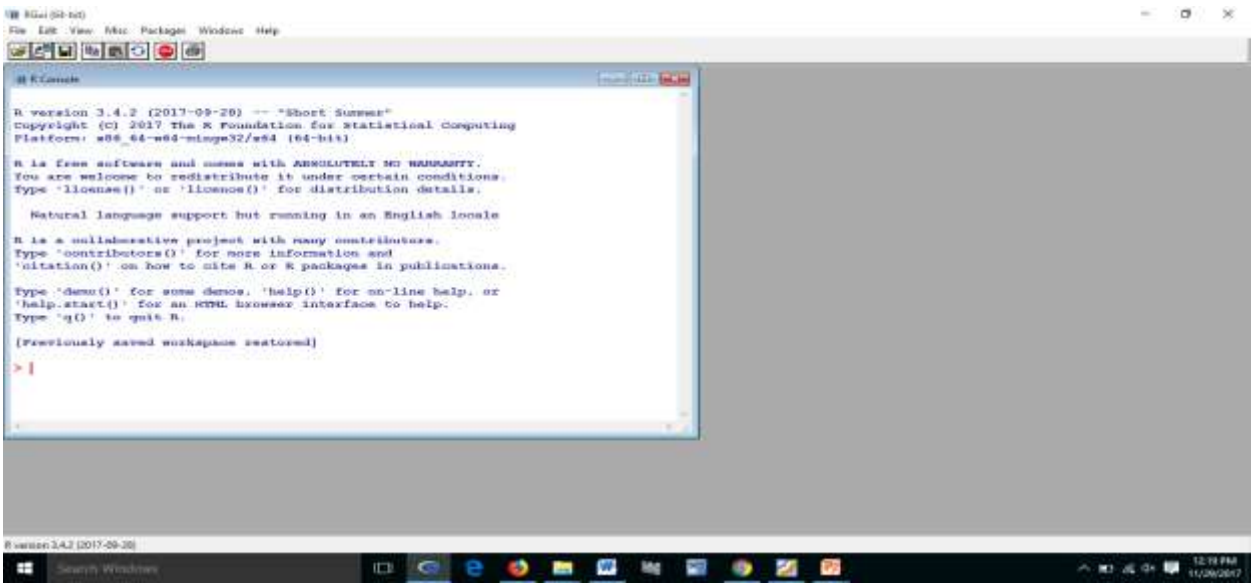
- with the R console GUI ;
- with the RStudio IDE ;

Of these, RStudio is for most R users the best choice; it contains an R command line interface but with a code editor, help text, a workspace browser, and graphic output.

R interface

Start the R system, the main window (**RGui**) with a sub window (**R Console**) will appear. In the **'Console'** window the cursor is waiting for you to type in some R commands.

The First R Session



Getting help

Once R is installed, there is a comprehensive built-in help system. So R has an inbuilt help facility. At the program's command prompt you can use any of the following:

help.start() # general help

To get more information on any specific named function, for example ***t.test***, the command is

help(*t.test*) # help about function *t.test*

?t.test # same thing

apropos("*t.test*") # list all functions containing string t.test

example(*t.test*) # show an example of function *t.test*

For a feature specified by special characters, the argument must be enclosed in double or single quotes, making it a "character string": This is also necessary for a few words with syntactic meaning including **if**, **for** and function.

Example, to get more information on any special characters, like **'%*%'**

```
help("%*%")
```

1.3. Commands and execution

Technically R is an expression language with a very simple syntax.

– Commands are separated either by a semi-colon ('**;**'), or by **a newline**.

- *Comments* can be put almost anywhere, starting with a hashmark (‘**#**’)
- If a command is not complete at the end of a line, R will give a different prompt, by default **+**

Results of calculations can be stored in **objects** using the assignment operators:

- An arrow **(<-)** formed by a less than character and a hyphen without a space!
- The equal character **(=)**.

These objects can then be used in other calculations. To print the object just enter the name of the object.

There are some restrictions when giving an object a name:

- Object names cannot contain `strange' symbols like !, +, -, #.
- A dot (.) and an underscore (_) are allowed, also a name starting with a dot.
- Object names can contain a number but cannot start with a number.

R is case sensitive, **X** and **x** are two different objects

Examples: Assign values to variables, various data types, and, very importantly, how to search for help.

First create a new object called 'x‘

```
x<-5
x=5
assign("x",5)
5->x # or
```

To list the objects that you have in your current R session use **ls()** or **objects().** If you assign a value to an object that already exists then the contents of the object will be overwritten with the new value (without a warning!). Use the function **rm** to remove one or more objects from your session.

Let’s create two small vectors

```
x <- c(1,2,3,4,5,6)
y <- c(6,8,3,5,7,1)
```

R Workspace

Objects that you create during an R session are hold in memory, the collection of objects that you currently have is called the **workspace.**

When you close the **RGui** or the R console window, the system will ask if you want to save the workspace image. If you select to **save** the workspace image then all the objects in your current R session are saved in a file .RData.

This is a binary file located in the working directory of R, which is by default the installation directory of R.

During your R session you can also explicitly save the workspace image. Go to the `File‘ menu and then select `Save Workspace...', or use the save.image function.

Save to the current working directory

```
save.image()
```

Just checking what the current working directory is

```
getwd()
```

Commands are entered interactively at the **R** user prompt. **Up** and **down arrow keys** scroll through your command history.

R gets confused if you use a path in your code like

```
c:\mydocuments\myfile.txt
```

This is because R sees "\" as an escape character. Instead, use

```
c:\\my documents\\myfile.txt      #or
c:/mydocuments/myfile.txt
getwd()           # print the current working directory
ls()              # list the objects in the current workspace
setwd(mydirectory)    # change to mydirectory
setwd("C:/Users/Likuta/Desktop")
```

R packages

One of the strengths of R is that the system can easily be extended. The system allows you to write new functions and package those functions in a so called `R package' (or `R library').

The R package may also contain other R objects, for example data sets or documentation. When you download R, already a number (around 30) of packages are downloaded as well. Other

packages can be installed from CRAN. There are now more than **13,000** R packages available for download

To use a function in an R package, that package has to be attached to the system. When you start R not all of the downloaded packages are attached, only seven packages are attached to the system by default. You can use **`search()`** to see a list of packages that are currently attached to the system

To attach another package to the system you can use the **menu** or the **library** function.

- Via the menu: Packages → Load package…
- Via the library function: library(*packagename*)

Examples

```
o library()        # list all available packages
o library(lib.loc = .Library) #list all packages in the
  default library
o library(help = splines) #documentation on package 'splines'
o library(splines)        #attach package 'splines'
o require(splines)         # the same
o search()                 # "splines", too
o detach("package:splines")
```

1.4. Simple manipulation

Data Input

Data Type: R has a wide variety of data types including scalars, vectors (numerical, character, logical), matrices, data frames, and lists.

1.4.1. Vectors; Generating sequences

Vectors

Numeric vector is a single entity consisting of an ordered collection of numbers. To set up a vector named **`a`**, say, consisting of five numbers, namely 10.4, 5.6, 3.1, 6.4 and 21.7, use the R command

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

Character vector,

```
b <- c("one","two","three")
```

Logical vector,

```
c <- c(TRUE,TRUE,TRUE,FALSE,TRUE,FALSE)
```

Refer to elements of a vector using subscripts.

```
a[c(2,4)] # 2nd and 4th elements of vector
```

Generating regular sequences

R has a number of facilities for generating commonly used sequences of numbers.

i. The **colon** operator has high priority within an expression

Example

- `1:30` is the vector c(1, 2, ..., 29, 30).
- `2*1:15` is the vector c(2, 4, ..., 28, 30).
- 30:1 is used to generate a sequence backwards.

ii. The function **`seq()`** is a more general facility for generating sequences.

Arguments to `seq()`,

`seq(from = value, to = value, by = value)`

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.

`seq(length = value, from = value, by = value)`

Example:

```
1. seq(from=1, to=30)
   seq(1, 30) # the same
2. seq(from =-5, to=5, by=0.2)
   seq(-5, 5, 0.2) # the same
   seq(length=51, from=-5, by=.2) # the same
```

Self-test exercise

1. What command would you use to create the following variables:
   a) `c(-3, -2, -1, 0, 1, 2, 3)`
   b) `c(0.5, 0.8, 1.1, 1.4, 1.7, 2.0)`
2. `x <- c(8, 1.4, 3, -5.4, 2.2)`
   a) What happens if we execute the command `x[4] <- 0.3`?
   b) And what for the command `x[6] <- 2.9`?
   c) What is the command `x[-1]` doing?
3. Put `n <- 10` and compare the sequences `1: n-1` and `1:(n-1)`.

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.

Usage

```
array(vector, dim = length(data))
```

Example - if the dimension vector for an array, say B, is c(3,4,2) then there are 3*4*2 = 24 entries in B 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].

```
B <- array(1:24, dim=c(3,4,2)) # Generate a 3 by 4 of 2 arrays.
```

We can use `array()` to generate matrix, 2 dimensional array

```
E <- array(1:12, dim =c(3,4))
```

Matrices

A matrix in mathematics is just a two-dimensional array of numbers. All columns in a matrix must have the same mode (numeric, character, etc.) and the same length.

The general format is

`mymatrix <- matrix(vector, nrow=r, ncol=c, byrow=FALSE)`

`byrow=TRUE` indicates that the matrix should be filled by rows. `byrow=FALSE` indicates that the matrix should be filled by columns (the default).

The rows and columns of matrices can be named using the `rownames()` or `colnames()`. For example, we can create names for A as follows.

Example: Generates 4 x 4 numeric matrix

```
A<-matrix(1:16, nrow=4,ncol=4)
A
```

We can name the rows and columns of matrices

```
rownames(A) <- letters[1:4]
colnames(A) <-c("FR", "SO", "JR", "SR")
```

One can identify rows, columns or elements using subscripts.

```
A[ ,4]      # 4th column of matrix
A[3, ]      # 3rd row of matrix
A[2:4,1:3] # rows 2,3,4 of columns 1,2,3
```

Matrix facilities

In the following table, A and B are matrices and x and b are a vectors.

Operator or Function	Description
A * B	Element-wise multiplication
A %*% B	Matrix multiplication
A %o% B	Outer product. AB'
crossprod(A,B) crossprod(A)	A'B and A'A respectively.
t(A)	Transpose
diag(x)	Creates diagonal matrix with elements of x in the principal diagonal
diag(A)	Returns a vector containing the elements of the principal diagonal
diag(k)	If k is a scalar, this creates a k x k identity matrix. Go figure.
solve(A, b)	Returns vector x in the equation b = Ax (i.e., $A^{-1}b$)
solve(A)	Inverse of A where A is a square matrix.
ginv(A)	Moore-Penrose Generalized Inverse of A. ginv(A) requires loading the MASS package.
y<-eigen(A)	y$val are the eigenvalues of A y$vec are the eigenvectors of A

y<-svd(A)	Single value decomposition of A. y$d = vector containing the singular values of A y$u = matrix with columns contain the left singular vectors of A y$v = matrix with columns contain the right singular vectors of A
cbind(A,B,...)	Combine matrices(vectors) horizontally. Returns a matrix.
rbind(A,B,...)	Combine matrices(vectors) vertically. Returns a matrix.
rowMeans(A)	Returns vector of row means.
rowSums(A)	Returns vector of row sums.
colMeans(A)	Returns vector of column means.
colSums(A)	Returns vector of column sums.

Self-test exercise

Given a matrix

$$D = \begin{bmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 5 & 3 & -3 \end{bmatrix}$$

a) Write a command to display a matrix above by using `matrix` and `rbind` functions.

b) Write a command that display only eigenvalues of matrix D and find the eigenvalues and eigenvectors of matrix D.

Data frames

A data frame is more general than a matrix, in that different columns can have different modes (numeric, character, factor, etc.).

Example

```
age <- c(25, 30, 56)
gender <- c("male", "female", "male")
weight <- c(160, 110, 220)
mydata <- data.frame(age, gender, weight)
```

There are a variety of ways to identify the elements of a dataframe .

`mydata[2:3]` # columns 2 &3 of data frame

`mydata[c("age","weight")]` # columns age and weight from data frame

`mydata$age` # variable age in the data frame

Lists

is an ordered collection of objects (components). A list allows you to gather a variety of (possibly unrelated) objects under one name.

Example of a list with 4 components

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

Identify elements of a list using the [[ ]] convention.

```
Lst[[2]] # 2nd component of the list
```

So in the simple example given above:

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

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

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

Factors

Factors in R are used to represent categorical data. Factors can be ordered or unordered and are an important class for statistical analysis and for plotting. Factors are stored as integers, and have labels associated with these unique integers. Once created, factors can only contain pre-defined set values, known as *levels*. By default, R always sorts *levels* in alphabetical order.

Example - create a numerical vector pain, encoding the pain level of five patients

```
> pain <- c(0,3,2,2,1)
> fpain <- factor(pain, levels=0:3) #
> levels(fpain) <- c("none","mild","medium","severe")
> fpain
[1] none severe medium medium mild
Levels: none mild medium severe
> as.numeric(fpain)
[1]  1 4 3 3 2
> levels(fpain)
[1] "none" "mild" "medium" "severe"
```

The function **as.numeric** extracts the numerical coding as numbers 1–4 and levels extracts the

names of the levels.

Notice that the original input coding in terms of numbers 0–3 has disappeared; the internal representation of a factor always uses numbers starting at 1.

Reading data from files

Importing Data

1. **From A Comma Delimited Text File (csv files)**

Use / instead of \ on windows systems

i. Using the **`read.table()`** function

If the data is located in working directory, we use

```
mydata <- read.table("mydata.csv", header=TRUE, sep=",")
```

ii. Using **`read.csv()`** function

```
mydata <- read.csv("mydata.csv")
```

In some systems you can use **file.choose()** to get the full path to a file. In particular this works well on **R GUI** for Windows or OS X. **For example:**

```
mydat <-read.table(file.choose(), header=TRUE, sep=",")
mydat <- read.csv(file.choose())
```

it will bring up a window for you to choose the file from.

Example –just practice

```
mydel<-read.table("deliv2.csv", header=TRUE, sep=",")
mydel<-read.csv("deliv2.csv")
mydel
```

2. **From Excel**

The best way to read an Excel file is to export it to a comma delimited file and import it using the method above.

3. **From SPSS File**

For the data files in SPSS format, it can be opened with the r**ead.spss** function from the **foreign** package. There is a "**to.data.frame**" option for choosing whether a data frame is to be returned.

```
>library(foreign)  # load the foreign package
>help(read.spss)  # documentation
>mydata <- read.spss("myfile", to.data.frame=TRUE)
>mydata<-read.spss("C:/Users/Likuta/p027.sav",
```

```
to.data.frame=TRUE)
  >mydata
```

Exporting Data

To A Comma Delimited Text File

```
>write.csv(deliv2,"C:/Users/Likuta/Desktop/mydelivery1.csv")
>write.csv(deliv2,"C:/Users/Likuta/Desktop/mydelivery1.csv",
row.names=FALSE)
>write.table(deliv2,
"C:/Users/Likuta/Desktop/mydelivery_1.csv", sep=",", row.names
= FALSE)
```