A Brief Introduction of R

G.J. LI

R is a statistical open source software freely available on `http://www.r-project.org/`. You can download R and many of its packages for free and install them onto your own computer. Open source means you can find the source codes of many functions implemented by R and many statistical methods are no longer black boxes. Apart from conveniently using statistical packages written by others, you can also write your own programs and do calculations, which would be hard or awkward to implement in a calculator or any point-and-click software. In the second semester, you will learn Matlab. In fact the two languages are very similar, see Hiebler (2011). This note is to introduce many basic concepts and operations in R. For more details, please refer to Lam (2010) and Venables et.al. (2012). In the following discussion, the following font type is used to indicate an R command or R message and any words after # serve as comments of explanation.

`command` #explanation

If you want to find more details of a command, you can type "?" followed by the name of the command and press enter.

1 Basic Objects

After opening R, you will find a cursor after a command prompt ">", from where you can type your commands and press enter to tell R what to do. For example,

`>ls()`

is command to see what is there in the workspace (the computer memory of R).[1] Since we have just opened R, there is nothing in the workspace. So we see `character(0)` on the screen. Now if we type

`>a=2` # = can be replaced by < − (less than and minus),

and type `ls()` again, we will see an object called a is in the workspace. If we type the name of the object, the content of a, which is 2, will be printed on the screen. a is called numeric object in R. There are other kinds of objects in R. For example, the commands

`>b="this is a string of characters"` #can use either single or double quotation

`>l=1>0`

will create *b* as a character object and *l* as a logical object, which has only two values, TRUE (1) or FALSE (0). The command to check the object kind is `class(x)`, in which you replace x by the name of the object.

2 Vector and Matrix

If we concatenate a few numbers and store them into an object, we could call it an numeric vector.

[1] Strictly speaking, it is an R function.

```
>nv=c(6,5,4,3,2,1) #c() is a system function
```

We can also concatenate a few character elements and store them into an object.

```
>cv=c("a string of characters","also a string of characters")
```

We can use the function `length()` to check how many elements a vector contains. To index a vector, we will use squared bracket. The commands

```
>nv[3]
>cv[2]
```

will print the third and the second element of *nv* and *cv* respectively.

To create a matrix in R, we basically need to organize the elements of a vector into box form. The command

```
>m1=matrix(nv,nrow=3,ncol=2)
```

will create a 3×2 matrix. If we check the length of $m1$, we can see that it has the same number of elements as in nv. After printing $m1$, we know that $m1$ just vertically separates nv into two columns. If we type

```
>class(nv)
>class(m1)
```

we will see that nv and $m1$ belong to different classes. Furthermore, if we type

```
>attributes(nv)
>attributes(m1)
```

we can see that while nv has no attributes, $m1$ has an attribute called *dim* (dimension) with values 3 and 2. Another command to check the dimension attribute of a matrix is `dim()`. We can say that in R, matrix is a vector with dimension attribute.

Many generic functions in R will have different methods to handle objects of different classes. For example, the `plot()` function will plot a vector and a matrix in a different way.

The following are some matrix functions for you to try out. For more other functions of calculation, check Hiebler (2011).

>`length(nv)` #show the number of elements

>`min(nv)` #minimum value of *nv*

>`which.min(nv)` #the index of the minimum value of *nv*

>`max(nv)` #maximum value of *nv*

>`which.max(nv)` #the index of the maximum value of *nv*

>`sort(nv,decreasing=TRUE)` #list *nv*'s elements of nv decreasingly,decreasing is an optional argument with default value FALSE

>`nv[2:5]` #print 2-5 elements of nv

>`m1[3,2]` #print the $(3,2)$ element of m1

>`m1[,2]` #print the 2nd column of m1

>`a*m1` #scalar multiplication

>`diag(2)` #identity matrix of dimension 2

>`rep(1,2)` #replicate 1 twice

>`cbind(rep(1,6),nv)` #put two vectors vertically together to make a matrix

>`m3=m1%*%t(m1)` #`t()` is to transpose a matrix and `%*%` is for matrix multiplication

>`qr(m3)$rank` #rank of $m3$

>`det(m3)` #determinant of $m3$

>`sum(diag(m3))` #trace of $m3$

`>solve(m3)` #inverse of $m3$

3 Infinite Values and Missing Data

There are limitations in R as to the range of numbers it can handle and the precision of calculations. It would be helpful to know about the data format in R. Usually a number is stored as a double precision floating number in R. If we type

```
>typeof(a)
```

we will see that a is stored in double precision format. We do not have to store a using double precision. For example, we can store a in integer format such that it will occupy less computer memory by typing

```
>ai=as.integer(a)
```

Usually, the maximum integer in R is $2^{31} - 1$ on a 32-bit computer[2] Try

```
>as.integer(2^31 - 1)
>as.integer(2^31)
```

The first command can produce numeric output while the second command gives NA (not available). In later discussion, we will talk about how to import data into R. If there are any missing observations in the data, R will treat them as NA. We can check whether an object is NA by using `is.na()` function. For example,

```
>is.na(log(-1))
```

The maximum floating point number in R can be much larger than the maximum integer and is usually 2^{1023}. If you type

[2]It depends on machines. Check out the command `.Machine`.

`>2^1024`

you can see R will produce `Inf` (infinite). For large numbers, we would better work with their logs.

4 Probability Related Functions

Probability functions in R usually starts with letter r (generate random number), d (probability density function), q (quantile function) and p (probability mass function or cumulative distribution function). The following are some examples.

>`runif(n,a,b)` #generate n random numbers uniformly from (a,b)

>`rnorm(n,mu,sigma)` #generate n normally distributed random numbers with mean mu and standard deviation sigma

>`rt(n,df)` #generate n random numbers from student t distribution with df degrees of freedom

>`dnorm(x)` #return the density function value of standard normal distribution evaluated at x

>`pnorm(q)` #return cumulative distribution function of standard normal distribution evaluated at q

>`qnorm(p)` #return the pth quantile of standard normal distribution ($0 \leq p \leq 1$)

5 Other Useful Objects

5.1 Factor Object

Factor object is useful to organize the data points into different categories. For example, the following command creates a factor object called gender with the first three observations as male while the last two observations as female.

```
>gender= factor(c("male","male","male","female","female"))
```

To check how many categories there are in a factor object, use the command below.

```
>levels(gender)
```

5.2 Time Series Object

A time series object is created by `ts()` and it associates every element in a vector with a time index. For example, the command below creates a monthly time series object with April 1987 as the starting point.

```
>myts1=ts(data=rnorm(100),start=c(1987,4),freq = 12)
```

The function `tsp()` returns the start and end time, and also the frequency without printing the complete data of the time-series.

5.3 List Object

We can use list objects to organize data of different classes. For example, the following list object contains the name, student number and marks of a student.

>`lobj <- list(name='Jack',stid=123,score=100)`#you may need to retype the single quotation when doing copy-and-paste

There are two ways to access the elements in a list object. The first way is to type $ followed by the name of the element after the name of the list object. The command

```
>lobj$name
```

will return the name of the student. You can check what other components are there in a list object by pressing the Tab key after $. Or you can use the command `names(lobj)`. We can also use double-squared-bracket index to access the list's elements. The command

```
>lobj[[3]]
```

will return the score of the student.

Each element in a list object can be a list. Try the commands below.

>`lobj2=vector('list',2)` #initialize an empty list object with two elements

>`lobj2[[1]] <- list(name='Jack',stid=123,score=100)` #the first student

>`lobj2[[2]]<-list(name='Mike',stid=456,score=60)` #the second student

>`lobj2[[1]]$stid` #the first student's ID number

>`lobj2[[2]][[3]]` #the second student's score

5.4 Dataframe Object

We can import external data files into R, which usually uses dataframe objects to handle the data. Download london.csv from Learning Central and put it under some folder. To read the file, we first need to set R's working directory, which contains the data file. The command

```
>setwd("filepath") #inside the quotation marks, type the folder location
```

tells R where to look for the data file. Alternatively, you can use the Windows file menu to set the working directory. Next, the command

```
>data=read.csv('london.csv')
```

reads the file into data. Check the class of data by `class(data)` and its type by `typeof(data)`. If we print its contents, we can see that dataframe is just like matrix with names for every column. Instead of printing all the contents of a dataframe object, sometimes we may just want to have some rough ideas about what the object is like. We can use the command `str(data)` to print a short description of the object. To access `data`'s column, we can use `data$columnname` (the type of `data` is list) or use matrix index. Unlike matrix, different columns in dataframe can belong to different classes. For example, the command

```
>data[,1]=as.character(data[,1])
```

changes the first column in data to be characters.

6 Some Statistical Functions

The following are some useful statistical functions.

```
>sum(data$income) #sum of the income of all observations
>mean(data$income) #sample mean of income
>var(data$income) #sample variance of income
>sd(data$income) #sample standard deviation of income
```

>`quantile(data$income)` #return some sample quantiles of income

>`hist(data$income,breaks=100,freq=FALSE)` #plot the empirical density with 100 cells

>`which(data$income>140)` #return the indices for observations with income higher than 140.

>`cov(data$income,data$totexp)` #sample covariance between income and total expenditure

>`cor(data$income,data$totexp)` #sample correlation between income and total expenditure

>`cor(data[,2:11])` #sample correlation matrix for the 2-11 variables in data

>`summary(data[,2:11])` #some statistics for the 2-11 variables in data

>`colSums(data[,2:11])` #return the column sums of the 2-11 variables in data

Question: Can you find out which variables have missing observation(s)? In what positions?

7 Least Squares Regression

You can perform OLS for some variables in a dataframe object. For example, the following command is to regress the budget share of food on the log of total expenditure and income using `data`.

```
>lmobj=lm(wfood~log(totexp)+log(income),data)
```

Strictly speaking, the term `wfood~log(totexp)+log(income)` is a formula object. For more details, see Lam (2010), pp. 142-144. We can check the type and class of `lmobj` by `typeof(lmobj)` and `class(lmobj)`. We can see that the type of `lmobj` is list. Hence any operations related to list can be applied to `lmobj`. On the other hand, the class of `lmobj` is `lm`. Some functions below are specially tailored for this class. For more details of these functions, see Kleiber and Zeileis (2008), Chapter 3.

>`summary(lmobj)` #summary of the fitted model

>`coef(lmobj)` #estimated model parameters ($\hat{\beta}_{OLS}$)

>`resid(lmobj)` #estimated residuals ($M_X y$)

>`fitted(lmobj)` #fitted values of the model ($P_X y$)

>`deviance(lmobj)` #the residual sum of squares (SSR or $y' M_X y$)

>`confint(lmobj)` #confidence interval

>`logLik(lmobj)` #value of the log likelihood function (assume normal error)

>`AIC(lmobj,k)` #information criterion, $k = 2$ for AIC and $k = log(N)$ for BIC (assume normal error, to be covered later)

>`vcov(lmobj)` #$\hat{\beta}_{OLS}$'s estimated variance-covariance matrix ($\widehat{\sigma^2}(X'X)^{-1}$)

>`anova(lmobj)` #returns an anova table

>`predict(lmobj,data.frame(totexp=1,income=1))` #returns predictions

>`plot(lmobj)` #create diagnostic plots

Question: Can you use matrix algebra in R to reproduce the outputs of the following functions: `coef(lmobj)`, `resid(lmobj)`, `fitted(lmobj)`, `deviance(lmobj)` and `vcov(lmobj)`?

8 Writing Functions

An R function is a procedure which takes in some or no inputs and returns some outputs. In addition to using some built-in functions in R, it is possible to write your own functions. By copying and pasting the following codes into R:

```
fr=function(x,a=100,b=1) {   ## Rosenbrock Banana function
    x1 <- x[1]
    x2 <- x[2]
    return(a * (x2 - x1 * x1)^2 + (b - x1)^2)
}
```

we have created a function with name fr with x as the variable and a and b as parameters (or data), which have default values equal to 100 and 1 respectively. Note that x is a 2×1 vector. In other words fr is function of two variables:

$$fr(x_1, x_2) = a(x_2 - x_1^2)^2 + (b - x_1)^2.$$

If we type `fr(c(0,0))`, R will return the result 1. We can use an R built-in function `optim()` to minimize or maximize a function. For example, the command

`>frres=optim(c(0,0),fr,a=101,b=2,hessian=TRUE)`

will minimize the fr function with $a = 101$ and $b = 2$ (different from the default values) and return the results as a list called `frres`. If we add the option `control=list(fnscale=-1)` into the `optim` function, we will do maximization instead of minimization.

9 Control the Flow of Execution

When we write a function, we may want the function to return different outputs based on the input. Consider the following function:

```
fif=function(x){
 if ((x%%2)==0){
  y="you have entered an even number."
 }else if((x%%2)==1){
  y="you have entered an odd number."
 }else{
  y="what you entered is not an integer."
 }
 return(y)
}
```

Sometimes we may want to do something a certain number of times in a function, called looping.

```
ffor=function(x){
 if (((x%%2)!=0&&(x%%2)!=1)||x<1){
  stop("Please enter an integer greater than 0.")
 }
 for (i1 in 1:x){
  print(paste("hi",as.character(i1)))
 }
}
```

When we use `for`, we know exactly how many loops (iterations) we want to do. We can use the `while` loop when we are not sure about the number of iterations.

```
fwhile=function(x){
 if (x<10){
  stop("Please enter an number greater than or equal to 10.")
 }
 y=0
 while(x>0.1){
  x=x/2
  y=y+1
 }
 return(y)
}
```

10 Install Package

Many functions in R are associated with packages. For example, we may want to test whether the residuals from the linear regression are normally distributed by Jarque-Bera (JB) test. We cannot find a function for this purpose if we have just installed R (base). If we do a bit of Google search, we can find that the package "tseries" contains a function called `jarque.bera.test()`, which can implement the JB test. To use this function, we first need to install the package by typing

>`install.packages(‘‘tseries’’)` #your computer must be connected to

the internet

After installation finishes, we have to load the package into the computer memory in order to use `jarque.bera.test()`. Type

```
>library(tseries)
```

Now the function `jarque.bera.test()` is at our disposal. We just need to put a vector containing the residuals into the brackets to implement the test.

Strictlty speaking, an R package is a collection of R functions, data, and compiled code in a well-defined format. So if we load a package, we can have access to not only functions, but also data in a package. We can check what packages are currently installed by `library()` and check what packages are currently loaded in the memory by `search()`. The website `http://cran.at.r-project.org/web/packages/` contains all the packages available for installation by the `install.packages()` command, while `http://cran.at.r-project.org/web/views/` organizes the packages according to different topics. A good thing about R is that all packages are well documented. Some even provide you with hands-on tutorials. All such information is easily obtained by simple googling. Moreover, since R is open sourced, any one can contribute to it. That is why R and its packages are developing so fast.

References

[1] D. Hiebler, *Matlab/R Reference*, `http://www.math.umaine.edu/~hiebeler/comp/matlabR.pdf`, 2011.

[2] C. Kleiber and A. Zeileis, *Applied Econometrics with R*, Springer, 2008.

[3] L. Lam, *An Introduction to R*, `cran.r-project.org/doc/contrib/Lam-IntroductionToR_LHL.pdf`, 2010.

[4] W.N. Venables, D.M. Smith and the R Core Team, *An Introduction to R*, ver.2.15.1, `cran.r-project.org/doc/manuals/R-intro.pdf`, 2012.