Introductory Guide to *R*

For UVM Statistics Students

Contents

1 Introduction and Preliminaries

1.1 What is R ?

R is a statistical analysis system created by Ross Ihaka & Robert Gentleman (1996, J. Comput. Graph. Stat., 5: 299-314). As a programming language R is considered to be a dialect of the language S created by the AT& T Bell Laboratories. S is available as the software S-PLUS commercialized by MathSoft (see http://www.splus.mathsoft.com/ for more information).

R is freely distributed from the Comprehensive R Archive Network (CRAN), at http://cran.r-project.org/ or http://www.r-project.org/, under the terms of the GNU Public Licence of the Free Software Foundation (for more information: http://www.gnu.org/). Its development and distribution are carried on by several statisticians known as the R Development Core Team.

1.2 Starting R

After you start the R program it issues a prompt when it is ready for input commands. The default prompt is "`>`"

To quit the R program the command is

```
> q()
```

or, select *File: Exit*.

1.3 Getting help with functions and features

R has an inbuilt help facility for many commands. To get more information on any specific named function, for example `solve` the command is

```
> help(solve)
```

Alternatively, you can use features under the Help menu (*Help: Html help*.

1.4 R commands. Case sensitivity.

Technically R is a *function language* with a very simple syntax. It is *case sensitive*, so `A` and `a` are different variables.

Commands are separated either by a semi-colon, `;`, or by a newline. If a command is not complete at the end of a line, R will give a different prompt, for example

```
+
```

on second and subsequent lines and continue to read input until the command is syntactically complete.

1.5 Executing commands from (or diverting output to) a file

The easiest way to do this is to cut and paste commands from a text editor (e.g., NotePad or MS Word) into the "R Console" window. Similarly, output and/or graphs can be cut and pasted into Word.

Alternatively, if commands are stored in an external file `commands.R` in the current working directory they may be executed at any time in an R session with the command

```
> source("commands.R")
```

Similarly

```
> sink("out.txt")
```

will divert all subsequent output from the terminal to an external file, `out.txt`, in the current directory. The command

```
> sink()
```

restores it to the terminal once again.

The current working directory can be identified by typing `getwd()`, which is short for "get working directory".

The working directory can be changed to, say `"C:\Documents and Settings\Rich\My Documents\R-work"`, by typing

```
> setwd("C:/Documents and Settings/Rich/My Documents/R-work")
```

Note: the slashes in the directory path are em forward slashes, not em back slashes. Alternatively, double back-slashes (e.g., \\) can be used instead of forward slashes as separators. This feature represents the UNIX origins of the R language.

The directory can also be changed by selecting *File: Change dir*

1.6 UVM-specific functions and data

There is a set of functions and data for the course located at "http://www.uvm.edu/∼rsingle/Rdata". At this location there is a file, called scripts.R, containing several functions defined for the course. The entire file can be cut and pasted into the "R Console" window.

Alternatively, if you are connected to the internet, you can type the command below to load these functions.

```
> source("http://www.uvm.edu/~rsingle/Rdata/scripts.R")
```

Further details about these functions and data are given below in Section 4.3.

2 Simple manipulations; numbers and vectors

2.1 Objects in R

R works with objects which all have two intrinsic attributes: mode and length. The mode is the kind of elements of an object; there are four modes: numeric, character, complex, and logical. The length is the total number of elements of the object.

A vector is a collection of items of the same mode. A factor is a categorical variable/vector. An array is a table with k dimensions, a matrix being a particular case of array with k = 2. Note that the elements of an array or of a matrix are all of the same mode. A data.frame is a table composed with several vectors all of the same length but possibly of different modes.

2.2 Vectors and Assignments

It is useful to know that R discriminates, for the names of the objects, the upper-case characters from the lower-case ones (i.e., it is case-sensitive), so that x and X can be used to name distinct objects (even under Windows).

The typical assignment operator consists of the two characters `<` (less than) and `-` (minus) side-by-side and it 'points' to the object receiving the value of the expression. (The equal sign, "`=`", has recently become an allowable synonym for the left pointing assignment operator (`<-`), but since it is used for other purposes in a different manner you can avoid potential confusion by using the `<-` operator.)

```
> a <- 1; A <- 10
> ls()
```

[1] "A" "a"

```
> A
```

[1] 10

```
> a
```

[1] 1

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.

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` remains 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.

The function `ls()` lists simply the objects in memory: only the names of the objects are displayed.

```
> name <- "Felix"; n1 <- 10; n2 <- 100; m <- 0.5
> ls()
```

[1] "m" "n1" "n2" "name"

Note the use of the semi-colon ";" to separate distinct commands on the same line. If there are a lot of objects in memory, it may be useful to list those which contain given character in their name: this can be done with the option pattern (which can be abbreviated with pat) :

```
> ls(pat="m")
```

[1] "m" "name"

If we want to restrict the list of objects whose names start with this character:

```
> ls(pat="^m")
```

[1] "m"

To delete objects in memory, we use the function `rm()`. `rm(x)` deletes the object x, `rm(x,y)` both objects x and y, `rm(list=ls())` deletes all objects in memory; the same options mentioned for the function `ls()` can then be used to delete selectively some objects: `rm(list=ls(pat="m"))`.

2.3 Vector arithmetic

Vectors can be used in arithmetic expressions, in which case the operations are performed element by element.

The elementary arithmetic operators are the usual `+`, `-`, `*`, `/` and `^` for raising to a power. In addition all of the common arithmetic functions are available. `log`, `exp`, `sin`, `cos`, `tan`, `sqrt`, and so on, all have their usual meaning. `max` and `min` select the largest and smallest elements of an vector respectively. `range` is a function whose value is a vector of length two, namely `c(min(x), max(x))`. `length(x)` is the number of elements in `x`, `sum(x)` gives the total of the elements in `x` and `prod(x)` their product.

Two statistical functions are `mean(x)` which calculates the sample mean, which is the same as `sum(x)/length(x)`, and `var(x)` which gives

```
sum((x-mean(x))^2)/(length(x)-1)
```

or sample variance.

`sort(x)` returns a vector of the same size as `x` with the elements arranged in increasing order; however there are other more flexible sorting facilities available (see `order()` or `sort.list()` which produce a permutation to do the sorting).

`rnorm(x)` is a function which generates a vector (or more generally an array) of pseudo-random standard normal deviates, of the same size as `x`.

2.4 Generating regular sequences

```
> 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`.

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`.

2.5 Logical vectors

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, `c1 | c2` is their union and `! c1` is the negation of `c1`.

Logical vectors may be used in ordinary arithmetic, in which case they are *coerced* into numeric vectors, `F` becoming `0` and `T` becoming `1`. However there are situations where logical vectors and their coerced numeric counterparts are not equivalent, for example see the next subsection.

2.6 Character vectors

Character quantities and character vectors are used frequently in R , for example as plot labels. Where needed they are denoted by a sequence of characters delimited by the double quote character. E. g. `"x-values"`, `"New iteration results"`.

Character vectors may be concatenated into a vector by the `c()` function; examples of their use will emerge frequently.

The `paste()` function takes an arbitrary number of arguments and concatenates them into a single character string. Any numbers given among the arguments are coerced into character strings in the evident way, that is, in the same way they would be if they were printed. The arguments are by default separated in the result by a single blank character, but this can be changed by the named parameter, `sep=`*string*, which changes it to *string*, possibly empty.

For example

```
> labs <- paste(c("X","Y"), 1:10, sep="")
```

makes `labs` into the character vector

```
("X1", "Y2", "X3", "Y4", "X5", "Y6", "X7", "Y8", "X9", "Y10")
```

Note particularly that recycling of short lists takes place here too; thus `c("X", "Y")` is repeated 5 times to match the sequence `1:10`.

2.7 Index vectors. Selecting and modifying subsets of a data set

Subsets of the elements of a vector may be selected by appending to the name of the vector an *index vector* in square brackets. More generally any expression that evaluates to a vector may have subsets of its elements similarly selected by appending an index vector in square brackets immediately after the expression.

For example `x[6]` is the sixth component of `x` and

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

selects the first 10 elements of `x`, (assuming `length(x)` $\geq$ `10`).

```
> y <- x[-(1:5)]
```

gives `y` all but the first five elements of `x`.

For vectors, matrices and arrays, it is possible to access the values of an element with a comparison expression as index:

```
> x <- 1:10
> x[x >= 5] <- 20
> x
[1] 1 2 3 4 20 20 20 20 20 20
> x[x == 1] <- 25
> x
[1] 25 2 3 4 20 20 20 20 20 20
```

3 Matrices

3.1 Matrix facilities. Multiplication, inversion and solving linear equations.

As noted above, a matrix is just an array with two subscripts. However it is such an important special case it needs a separate discussion. R contains many operators and functions that are available only for matrices. For example `t(X)` is the matrix transpose function, as noted above. The functions `nrow(A)` and `ncol(A)` give the number of rows and columns in the matrix `A` respectively.

The operator `%*%` is used for matrix multiplication. An $n \times 1$ or $1 \times n$ matrix may of course be used as an $n-$vector if in the context such is appropriate. Conversely vectors which occur in matrix multiplication expressions are automatically promoted either to row or column vectors, whichever is multiplicatively coherent, if possible, (although this is not always unambiguously possible, as we see later).

If, for example, `A` and `B` are square matrices of the same size, then

```
> A * B
```

is the matrix of element by element products and

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

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

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

is a quadratic form.[1]

The function `crossprod()` forms "crossproducts", meaning that

`> crossprod(X, y)` is the same as `t(X) %*% y`

but the operation is more efficient. If the second argument to `crossprod()` is omitted it is taken to be the same as the first.

Other important matrix functions include `solve(A, b)` for solving equations, `solve(A)` for the matrix inverse, `svd()` for the singular value decomposition, `qr()` for QR decomposition and `eigen()` for eigenvalues and eigenvectors of symmetric matrices.

3.2 Forming partitioned matrices. `cbind()` and `rbind()`.

As we have already seen informally, matrices can be built up from other vectors and matrices by the functions `cbind()` and `rbind()`. Roughly `cbind()` forms matrices by binding together matrices horizontally, or column-wise, and `rbind()` vertically, or row-wise.

Suppose `X1` and `X2` have the same number of rows. To combine these by columns into a matrix `X`, together with an initial column of `1`s we can use

```
> X <- cbind(1, X1, X2)
```

The result of `rbind()` or `cbind()` always has matrix status. So, `cbind(x)` and `rbind(x)` are possibly the simplest ways explicitly to allow the vector `x` to be treated as a column or row matrix respectively.

4 Reading Data

4.1 Entering Data Interactively

The assignment operator [e.g., `x <- c(1,2,3)`] is fine for entering small data sets. For larger data sets it is much easier to read them in from a space-, tab-, or comma- delimited text file.

The `data.entry` function is a passable graphical interface for entering small data sets. The object must exist before you can open it for editing using `data.entry`. One way to accomplish this is to assign an intitial empty object (e.g., `x <- character(0)` or `x <- numeric(0)`). After typing `data.entry(x)`, you can enter appropriate values according to the type (character or numeric). You can create additional objects by clicking on the gray "var2" box to give it a name and type. When you are finished, click on the `x` in the upper right corner of the "Variable Editor" window. After entering the data, click on the `x` in the upper right corner of the "Data Editor" window. This will close the object(s) and save the changes. The command `data.entry(x,y)` will bring up the objects `x` and `y` in the "Data Editor" window. Type `help(data.entry)` for further details.

[1] Note that `x %*% x` is ambiguous, as it could mean either $\mathbf{x}'\mathbf{x}$ or $\mathbf{x}\mathbf{x}'$, where $\mathbf{x}$ is the column form. In such cases the smaller matrix seems implicitly to be the interpretation adopted, so the scalar $\mathbf{x}'\mathbf{x}$ is in this case the result. The matrix $\mathbf{x}\mathbf{x}'$ may be calculated either by `cbind(x) %*% x` or `x %*% rbind(x)` since the result of `rbind()` or `cbind()` is always a matrix.

4.2 The `scan()` function

To read a matrix (5×3) from the file `input.dat` as in Figure 1, use

```
> X <- matrix(scan("input.dat"), ncol=3, byrow=T)
```

```
52.00    111.0      830
54.75    128.0      710
57.50    101.0     1000
57.50    131.0      690
59.75     93.0      900
```

Figure 1: Input file form for a matrix

4.3 Reading Data from Files

There are three major sources of data for the class. These can be accessed using the `data()`, `bookdata()`, and `otherdata()` commands described below.

The command `data()` lists the data sets that come with the R software and some that I have added.

```
> data(state); ls()
```

[1] "state.abb" "state.area" "state.center" "state.division"

[5] "state.name" "state.region" "state.x77"

Exercises from the textbook can be accessed using the `bookdata()` command. For example the data for exercise #1 in chapter 5 can be assigned to the object `dat` using the command `dat <- bookdata("ex0501.dat")`. Note that the syntax requires quotes around the name of the exercise.

The syntax for the `otherdata()` command is similar to that of the `bookdata()` command. For example,

```
> dat <- otherdata("example.dat")
```

5 Writing Functions

5.1 A simple example

There is a built in function for computing the standard deviation named `sd`. This function takes into consideration the type of data object and computes the standard deviation appropriately.

You can define your own standard deviation function `stdev` with the line below.

```
> stdev <- function(x){ return(sqrt(var(x))) }
```

`sd(x)` and `stdev(x)` should return the same result as long as the object `x` is a simple vector of numeric values.

6 Graphical procedures

6.1 High-level plotting commands

High-level plotting functions are designed to generate a complete plot of the data passed as arguments to the function. Where appropriate, axes, labels and titles are automatically generated (unless you request otherwise.) High-level plotting commands always start a new plot, erasing the current plot if necessary.

6.1.1 The `plot()` function

One of the most frequently used plotting functions in R is the `plot()` function. This is a *generic* function: the type of plot produced is dependent on the type or *class* of the first argument.

`plot(x,y)`	If `x` and `y` are vectors, `plot(x,y)` produces a scatterplot of `x` against `y`.
`plot(x)`	Produces a time series plot if `x` is a numeric vector or time series object, or an Argand diagram if `x` is a complex vector.

6.1.2 Displaying multivariate data

R provides two very useful functions for representing multivariate data. If `X` is a numeric matrix or data frame, the command

```
> pairs(X)
```

produces a pairwise scatterplot matrix of the variables defined by the columns of `X`, that is, every column of `X` is plotted against every other column of `X` and the resulting $n(n-1)$ plots are arranged in a matrix with plot scales constant over the rows and columns of the matrix.

6.1.3 Display graphics

Other high-level graphics functions produce different types of plots. Some examples are:

`qqnorm(x)` `qqplot(x,y)`	Distribution-comparison plots. The first form plots the quantiles for the vector `x` against the expected Normal quantiles. The second form plots the quantiles of `x` against those of `y` to compare their respective distributions.
`hist(x)`	Produces a histogram of the numeric vector `x`. A sensible number of classes is usually chosen, but a recommendation can be given with the `nclass=` argument. Alternatively, the breakpoints can be specified exactly with the `breaks=` argument. If the `probability=T` argument is given, the bars represent relative frequencies instead of counts.

`pie(slices,`
`names,`
`explode=...)`
	Make a pie diagram, including the possibility of some pieces displaced or *exploded* out from the centre. (Pie diagrams are especially good for showing to administrators and bosses, but not much else, in my opinion.)

6.2 Low-level plotting commands

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

Some of the more useful low-level plotting functions are:

`points(x,y)` `lines(x,y)`	Adds points or connected lines to the current plot. `plot()`'s `type=` argument can also be passed to these functions (and defaults to `"p"` for `points()` and `"l"` for `lines()`.)
`text(x, y,` `  labels, ...)`	Add text to a plot at points given by `x, y`. Normally `labels` is an integer or character vector in which case `labels[i]` is plotted at point `(x[i], y[i])`. The default is `1:length(x)`. Note: This function is often used in the sequence `> plot(x, y, type="n"); text(x, y, names)` The graphics parameter `type="n"` suppresses the points but sets up the axes, and the `text()` function supplies special characters, as specified by the character vector `names` for the points.

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

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

6.2.1 Multiple figure environment

R allows you to create an $n \times m$ array of figures on a single page. Each figure has its own margins, and the array of figures is optionally surrounded by an *outer margin*.

The graphical parameters relating to multiple figures are as follows:

`par(mfrow=c(3,2))`	Set size of multiple figure array. The first value is the number of rows; the second is the number of columns.
`par(mfrow=c(1,1))`	Returns to one figure per plot (the default)