STATISTICAL MODELLING Part IIC

Practical 1: Introduction to R IAC/Lent 2011

Comments and corrections to ioana@statslab.cam.ac.uk

A reminder before you start: **Please don't ever turn off the computers in this room when you log out.**

The following must be done at the beginning of each session:

1. Log in using your PWF password.
2. Create a directory called `Rwork` inside the `U:` drive. I.e.,
   (a) Open the `U:` drive by double–clicking on the icon at the top–left of your Desktop called `My Documents (Drive U)`.
   (b) Inside the window that opens up, right–click and select `New Folder`.
   (c) Name the new folder `Rwork`.
3. Start R. The launcher for R is located in the `Start` menu.
   `All Programs` → `Spreadsheets Mathematics and Statistics` → `R Statistical Data Analysis 2.11.1`

Statslab users on Unix machines need only type R from the command prompt in your home (or other) directory.

Note that R is free software. Precompiled binary distributions are available from `www.r-project.org` for Windows and Max OS X, and the nearest mirror site for downloading a copy is `http://www.stats.bris.ac.uk/R/`. Source code is available for other architectures. You are encouraged to install R on your home computer, and to download some of the R documentation. Follow the *Manuals* link in the left hand column under the heading *Documentation*. Download *An Introduction to R*. It provides a good reference at a level suitable for new users.

Many of these practicals are too long and involved to be completed in the 1–hour time–slot allotted. When this happens, you should feel obliged to complete the sheet, and exercises, at home or in the lab on your own time. Treat the exercises at the end of these sheets as mini example sheets, and bring them to your supervisions.

R can be used as a calculator:

```
> sqrt(14) * exp(-5) * choose(5, 2)/(log(4 * pi) + gamma(5))
```

```
[1] 0.009502494
```

In this example, the symbol > is the R prompt, and the [1] states that the answer is starting at the first element of a vector. Note that scalars are vectors containing only a single element. We have just used some of R's built-in functions: `sqrt`, `exp`, `choose`, `log`, and `gamma`. Help on any R function can be found by typing a question mark followed by the function, e.g.,

```
> ?choose
```

(The instance `choose`(5,2) returns $\binom{5}{2}$.) Alternatively, type at the prompt: `help(choose)`. For a feature specified by special characters and in a few other cases (one is "`function`"), the argument must must be enclosed in single or double quotes. For example, to get help using the arithmetic operator + use one of the following: `help("+")` or `?"+"`.

You will need to use this help facility extensively (and get used to skim-reading to find the relevant bit!). If using Unix, pressing `q` exits the help window. Sometimes there is more detailed information on functions available by typing `help.start()`, which has a search engine, amongst other things. Previous commands can be accessed by hitting the up arrow key. Note that R is case-sensitive.

Also note that R includes some useful constants, for instance `pi`. One that is a little bit tricky is e; use `exp(1)`.

In R, basic commands are generally either assignments or expressions. When an expression is given, it is evaluated, printed, then discarded. An assignment is evaluated and the value is stored as a variable, but is not automatically printed. The <- symbol is the assignment operator in R. For instance:

- We can assign the value 3 to the variable `x`, and then perform operations on `x`:

  ```
  > x <- 3
  > round(x^2 - log10(x), 3)
  ```

  ```
  [1] 8.523
  ```

```
> 37%/%x

[1] 12

> 37%%x

[1] 1
```

- The `c()` function combines values into a vector or list.

```
> x <- c(3, 6, 4, 2)
> x

[1] 3 6 4 2

> length(x)

[1] 4
```

  You can create a vector `y` with the same entries using `y <- scan()`. Enter one component per line and leave a blank line after the last. Try this! The function `scan` can also read data in from files. See the help file for details.

One of the first things to get used to is the way in which operations on vectors in R are performed component by component. This often avoids the need to write loops and can decrease the running time of algorithms. For example:

- scalar–vector multiplication and addition:

```
> 4 * x + 3

[1] 15 27 19 11
```

- creating sequences:

```
> 1:8

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

  `a:b` is a special case of `seq`; see also `rep`

- creating matrices:

```
> A <- matrix(1:8, nrow = 2, ncol = 4, byrow = FALSE)
> A

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

Can you use `matrix` to enter the terms by row instead?

- matrix–vector multiplication is done with `%*%`; component–wise multiplication with `*`; transposes are accomplished with `t()`.

```
> A %*% x

     [,1]
[1,]   55
[2,]   70

> A * A

     [,1] [,2] [,3] [,4]
[1,]    1    9   25   49
[2,]    4   16   36   64

> t(A)

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

- `solve(A)` returns the inverse of matrix `A`. What does `solve(A,b)` return?

```
> solve(A %*% t(A))

      [,1]  [,2]
[1,]  1.50 -1.25
[2,] -1.25  1.05
```

- means of each row of A:

```
> apply(A, 1, mean)
```

```
[1] 4 5
```

How would you find the mean of each column?

Components of a vector or matrix can be extracted:

- by giving indeces to specify the components

```
> x[2]

[1] 6

> A[2, 3]

[1] 6

> x[1:3]

[1] 3 6 4

> x[-2]

[1] 3 4 2
```

- by giving a vector of `True`/`False` indicating which component to be returned. The binary operator `<` performs component-wise comparison.

```
> x < 4

[1]  TRUE FALSE FALSE  TRUE

> x[x <= 4]

[1] 3 4 2

> x[x <= 5 & x > 2]

[1] 3 4
```

- The following use the convention `True = 1`, `False = 0`; note the `==` which is needed for logical comparisons. What is `!=`?

```
> sum(x == 4)
```

```
[1] 1

> sum(x != 4)

[1] 3
```

Vectors can be sorted via the function `sort`.

- The following sorts in increasing order. How about decreasing order?

```
> x <- c(3, 7, 2, 4, 1, 9)
> sort(x)

[1] 1 2 3 4 7 9
```

- What do the following functions do?

```
> sort.list(x)

[1] 5 3 1 4 2 6

> x[sort.list(x)]

[1] 1 2 3 4 7 9

> x[sort.list(-x)]

[1] 9 7 4 3 2 1
```

Lists collect together items of different types, and names can be specified for different items.

- For example

```
> Empl <- list(employee = "Eve", spouse = "Adam", children = 2,
+     child.ages = c(4, 7))
> Empl
```

```
$employee
[1] "Eve"

$spouse
[1] "Adam"

$children
[1] 2

$child.ages
[1] 4 7
```

Elements of a list need not be of the same length, but its components are numbered. Thus `Empl` is a list of length 4, and its components are referred to as `Empl[[1]]`, etc. Notice that `Empl[[4]]` is a vector, so `Empl[[4]][1]` is its first entry.

- Names of components can also be used to extract components.

```
> Empl$employee

[1] "Eve"

> Empl$child.ages[2]

[1] 7
```

Random numbers are generated with commands like `rnorm, runif, rbeta`, etc. The corresponding density, cumulative distribution and quantile functions are, e.g., `dnorm, pnorm, qnorm`. What do the functions `summary` and `sample` do?

```
> X <- rnorm(20, mean = 1, sd = 2)
> X


 [1]  4.9378785 -0.3023774 -0.9520653 -0.6871562 -2.2990575  1.4725558
 [7]  1.2549242 -1.1559552  1.2298580  1.1707465  2.6901323  1.4570274
[13]  0.9238759  0.8895438  2.9912317  4.3533328  1.1513152  2.8999520
[19] -2.1080259 -1.3379341


> summary(X)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.2990 -0.7534  1.1610  0.9290  1.7770  4.9380
```

```
> sample(X, 15, replace = T)
```

```
 [1] -1.1559552  4.3533328  4.3533328  1.2549242 -0.9520653  2.8999520
 [7]  2.6901323 -0.3023774  4.9378785  1.4570274  2.8999520 -1.3379341
[13] -1.3379341  1.2549242 -1.1559552
```

EXERCISES:

1. What is the upper 5% point of a χ^2_6 distribution?
2. Use `R` to solve

$$\begin{aligned} 3a + 4b - 2c + d &= 9 \\ 2a - b + 7c - 2d &= 13 \\ 6a + 2b - c + d &= 11 \\ a + 6b - 2c + 5d &= 27. \end{aligned}$$

3. Use `R` to estimate $\mathbb{E}(X^6)$ when $X \sim N(0,1)$.
4. With $n = 10$, generate $X_1, \ldots, X_n \overset{\text{iid}}{\sim} N(0,1)$. Now generate a bootstrap sample from $\{X_1, \ldots, X_n\}$; i.e., given the data $\{X_1, \ldots, X_n\}$, generate independent and identically distributed $X_1^*, \ldots, X_n^*$ such that each X_i^* has probability $1/n$ of being X_j, for $j = 1, \ldots, n$.

FINAL COMMENTS:

1. Escaping R: The two most common ways to get stuck in R are endless loops (or really long computations that you didn't intend) and unbalanced parenthesis. When you are in the endless loop situation, the prompt looks like this

   ```
   > while(TRUE) {}
   |
   ```

   In this situation, press `Esc` (in Windows or Mac OS X) or `Ctrl-C` (in Linux) to cancel the currently running command and return the command prompt. When you have unbalanced parenthesis, the prompt looks like this

```
> t <- sqrt(n) * (mean(x) - mu / std.dev(x)
 +
```

The way out of this is to type a bunch of right parenthesis, hit return (which generates a syntax error), then type the command again but with the parenthesis in the right places. Typing `Esc` (in Windows or Mac OS X) or `Ctrl-C` (in Linux) will also return you to the command prompt.

2. Exiting R: To exit R, type `q()`. You have the option of saving your commands and the variables you have created. Next time you run R, you can see which objects are in your workspace with `ls()`; they are deleted with `rm(x)`, for example, or `rm(list=ls())` to clear the entire workspace.