A (very) brief introduction to R

You typically start R at the command line prompt in a command line interface (CLI) mode. It is not a graphical user interface (GUI) although there are some efforts to produce one for R. CLI is a bit daunting for some GUI point-and-click users, but it is powerful and fits the programming environment that R is designed for. So, we have to live with it!

From the Unix command prompt $, we enter into the R environment which has its own command prompt, ">".

```
$ R
R version 2.10.1 (2009-12-14)
Copyright (C) 2009 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
>
```

To quit, you can press Control-D keys together, or enter

```
> q()
$
```

This gets you back to the Unix command prompt, "$". Essentially **R** to get in, and **q()** to quit.

Inside the R environment, we can do everything we want R to do by typing in specific commands. Since you are interested in parallel computing in R, we believe you want to create an R script to run non-interactively. You can store all your commands in a script with a shebang "#!" like as follows:

```
#!/usr/bin/Rscript
# ... Insert R commands hereafter...
```

Or simply run:

```
$ R --vanilla < your.script.R
```

Interactive R environment

R handles data very well. One basic data structure is the **vector**. **Numeric** vectors are like an entity or a variable which is named using a series of alphabets, and each consisting of an ordered collection of numbers. If we want to have variable x containing numbers from 1 to 6, here's how we assign it.

```
> x <- c(1,2,3,4,5,6)        # individually
> x <- 1:6                   # also does the trick, faster!
>
```

After typing the above, and hitting ENTER, nothing seems to happen. That is good! No mistakes or errors. To check that variable x contains the values 1 to 6

```
> x
[1] 1 2 3 4 5 6
>
```

This means the vector x comprises values 1 2 3 4 5 6, exactly as we intended. The symbol "[1]" shows the index of the first element of that display-row.
We can assign $y = x^2$ easily, which is called vectorized calculation.

```
> y <- x^2
>
```

Check y is has values of x^2

```
> y
[1] 1 4 9 16 25 36
>
```

 Indeed, we now have two vectors, x and y, where x is from 1 to 6 and y is x^2 from 1, 4, 9...36. We can also think of x and y as two columns of numbers and say we want to **plot** them, (x,y): (1,1), (2,4), (3,9), (4,16), (5,25) and (6,36).

```
> plot(x,y)
>
```

And a scatter plot of these six points will appear in another window with default, circles for each point and default axes x (from 1 to 6) and y (from 1 to 36).

Now we can introduce another data structure called the **data frame**.

NOTE: R is case sensitive, so be careful to distinguish between upper and lowercase!

```
> z <- data.frame(x, y)
> z
  x  y
1 1  1
2 2  4
3 3  9
4 4 16
5 5 25
6 6 36
> plot(z)
```

Another very useful data structure in R is **list**.

```
> ls <- list(x, y)
> ls
[[1]]
[1] 1 2 3 4 5 6                  # this is x

[[2]]
[1]  1  4  9 16 25 36            # this is y

> lt <- list(a=x, b=y)          # assign labels to each element of the
list
> lt
$a
[1] 1 2 3 4 5 6                  # this is x

$b
[1]  1  4  9 16 25 36            # this is y
```

We can therefore store elements with different data structures into one list and the data structure of each element is retained, and so even if y is a data.frame, the structure of y as a data frame will be retained in the list.

Back to the variable x and y, we can also do some simple statistics too on these values, e.g. mean, standard deviation (**sd**) and variance (**var**).

```
> mean(x)
[1] 3.5
> mean(y)
[1] 15.16667
> sd(x); sd(y)               # we can do two commands on the same line with a
";"
[1] 1.870829                 # the output gets catenated one after the other
[1] 13.37784
> var(x); var(y)
```

```
[1] 3.5
[1] 178.9667
```

Since we have stored both x and y into a list lt, we can use one of the apply functions called **lapply** to operate an R function on each element of the list, and each element, in this case, lt$a and lt$b, containing a numeric vector each, of which we will calculate the mean and the sd as follows:

```
> lapply(lt, mean)
$a
[1] 3.5

$b
[1] 15.16667

> lapply(lt, sd)
$a
[1] 1.870829

$b
[1] 13.37784
```

lapply if applied to a numeric vector will thus simply apply the mean (or sd) to all the numbers.

Alternatively, we can also pass another customised function, to lapply, which we can define as function acting on a dummy variable x, which calculates the mean of x and the sd of x, and assigns them to mymean and mysd respectively:

```
> lapply(lt, function(x) c(myamean=sum(x)/length(x),
                           mygmean=prod(x)^(1/length(x))))
$a
   myamean          mygmean
   3.500000       2.993795
$b
   myamean          mygmean
  15.16667        8.96281
```

Some of these functions are already pre-built. For example, we can generate a summary of statistics.

```
> summary(x)
   Min. 1st Qu.  Median   Mean 3rd Qu.   Max.
   1.00    2.25    3.50    3.50    4.75    6.00
```

Fit a linear regression model on x and y, and plot out the diagnostic pots of the regression model, all four diagnostic plots on a single window two plots by two.

```
> summary(lm(y ~ x))
Call:
lm(formula = y ~ x)
Residuals:
      1       2       3       4       5       6
3.3333 -0.6667 -2.6667 -2.6667 -0.6667  3.3333
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -9.3333     2.8441  -3.282 0.030453 *
x             7.0000     0.7303   9.585 0.000662 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 3.055 on 4 degrees of freedom
Multiple R-squared: 0.9583,     Adjusted R-squared: 0.9478
F-statistic: 91.87 on 1 and 4 DF,  p-value: 0.000662
> par(mfrow=c(2, 2))
> plot (lm(y ~ x))
```

We can transform the values in x and y by addition, substraction, multiplication or division very easily, as follows:

```
> z <- (x + y)
> a <- (y - x)
> b <- (y / x)
> c <- (x * y)
> x; y; z; a; b; c
[1]  1 2 3 4 5 6
[1]  1   4   9 16 25 36
[1]  2   6 12 20 30 42
[1]  0   2   6 12 20 30
[1]  1 2 3 4 5 6
[1]  1   8  27  64 125 216
```

We can easily invoke high level plot functions to plot the values:

```
> plot(a,b,type='p')       # plot points, b against a (default - points)
> plot(a,b,type='l')       # plot line,   b against a
> plot(a,b,type='b')       # plot both line and points

> plot(x,z,type='b',pch=1)    # plot character (pch=1, 2, ...)
> plot(a,c,type='b',lty=1)    # line type (lty=1, 2, ...)
> plot(x,c,type='b',lwd=3)    # line thickness (lwd= 1, 2,...)
> plot(x,a,type='b',col="red") # color (col="red", "blue",...)
> colors()                    # will show you a list of colors
```

Conceptually, there are three basic plotting functions in R: high-level plots, low-level plots, and the layout command **par**. Basically, a high-level plot function creates a complete plot and a low-level plot function adds to an existing plot, that is, one created by a high-level plot command.

After a high level function creates a plot, we can add to it low level plot commands (see a R tutorial manual for details).

The par command controls the layout of the graphics device. The option you will use most often will
probably be mfrow (multi-figure, by row), or mfcol. For example, to have a 3x2 layout where the plots
are added by row, set

```
> par(mfrow=c(3,2))
```

This setting will exist throughout the life of the graphics device unless you change it back to the default mfrow=c(1,1).

this new device will have the default layout setting. To see your devices

```
> dev.list()
```

Or your current device

```
> dev.cur()
```

To plot a graph and save into JPG format, find out your current working directory as to where the file will be saved. Next, invoke the jpeg command to define the basic parameters of the jpg file. Then, invoke the high command level command to plot (x,y). You can add more lines using the low level commands, lines and title. Finally you must switch off the device to make sure the file gets formed. Otherwise, it is still waiting for more commands.

```
> getwd()
/root/Desktop
> jpeg('test.jpg', quality = 100, bg = "white", res = 200, width = 7,
height = 7, units = "in")
> plot(x,y,type="b")
> lines(x,z,type="b",col="red"
> lines(x,a,type="b",col="blue"
> title("Test Plot")
> dev.off()
```

Generally, we plot the graphs to the default device (in this case X11 window) until we are satisfied, and then we send it to the output. In R, we can do the same by using the dev.copy() command.

```
> plot(x,z,type="b",col="blue")        # initiate the main plot
> lines(x,y,type="b",col="red")        # add one line
> lines(x,a,type="b",col="green")       # add another line
> lines(x,b,type="b",col="cyan")       # and another
> title("Test Plot")                    # add the title
> dev.copy(jpeg,"test2.jpg")            # copy to output jpeg format
> dev.off()                            # switch off device and save
> dev.copy(pdf,"test2.pdf")             # copy to output in pdf format
> dev.off()                            # switch off device and save
```

Formats which can be used:

Format	Driver	Notes
JPG	jpeg	Can be used anywhere, but doesn't resize
PNG	png	Can be used anywhere, but doesn't resize
WMF	win.metafile	Windows only; best choice with Word; easily resizable
PDF	pdf	Best choice with pdflatex; easily resizable
Postscript	postscript	Best choice with latex and Open Office; easily resizable

Instead of typing in your values, you can generate them, for example using a random number generator, or import them from an existing file or a spreadsheet.

Generating Random numbers

```
> k=runif(1000)    # generate 1000 random uniform numbers
> hist(k)          # plot and see the random number histogram
> n=rnorm(1000)    # generate 1000 random numbers in normal distribution
> hist(n)          # plot and see the normal distribution of the
randomnumbers
> e=rexp(2000)     # generate 2000 random numbers in exponent distribution
> hist(e)          # plot the exponential distribution
> p=rpois(3000,3) # generate 3000 random numbers and lambda 3 in poisson
> hist(p)          # plot the poisson distribution
```

Doing statistical testing

```
> n=rnorm(1000)    # generate 1000 random numbers mean about 0 normal
distribution
> n                # lists all 1,000 numbers to the screen
   [1]  0.7832942667  0.5827743341 -0.2830421642 -0.2913130854
-0.2787417952
   [6]  1.0217393633 -1.4070364216 -2.0348665593 -1.1887303759
0.0196533509
  [11]  1.9770762525  0.2536302575  0.2851324087  0.8429003716
-1.3687178855
...
> t.test(n)
```

```
        One Sample t-test
data:  n
t = -0.3196, df = 999, p-value = 0.7493
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
-0.06865001  0.04941785
sample estimates:
   mean of x
-0.009616076
```

If we change the mean for example:

```
> n2=rnorm(1000,mean=5)         # generate the normal distribution mean=5
> hist(n2)                       # plots the histogram; check mean about 5
> t.test(n2)                    # default mean is 0
         One Sample t-test
data:  n2
t = 161.4026, df = 999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
4.91280 5.03373
sample estimates:
mean of x
4.973265


>  t.test(n2,mu=5)              # mean is 5
        One Sample t-test
data:  n2
t = -0.8677, df = 999, p-value = 0.3858
alternative hypothesis: true mean is not equal to 5
95 percent confidence interval:
4.91280 5.03373
sample estimates:
mean of x
4.973265


> n3=rnorm(1000,mean=3.9999)  # generate the normal distribution close to 4
> n4=rnorm(1000,mean=4.0000)  # generate the normal distribution mean=4
> t.test(n3,n4)
        Welch Two Sample t-test
data:  n3 and n4
t = -0.271, df = 1996.99, p-value = 0.7864
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-0.10070931  0.07625618
sample estimates:
mean of x mean of y
4.006663  4.018890
```

```
# Very likely that the two means are the same – p value not significant



> t.test(n,n4)
        Welch Two Sample t-test
data:  n and n4
t = -91.3284, df = 1988.331, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
-4.115013 -3.941999
sample estimates:
   mean of x    mean of y
-0.009616076  4.018889728
# Very likely that the two means are not the same – p value is very
significant
```

Importing and exporting

```
> getwd()                              # find out your current working
directory
> setwd("directory")                # set your working directory
> save.image("my.RData")             # save in current directory my.RData
> load("my.RData")                   # load the image back
> sink("myfile", append=FALSE, split=FALSE)  # direct text output to file
> sink()                             # redirect text output back to
terminal
> pdf("my.pdf")                      # redirect graph to pdf
> jpeg("mygraph.jpg")                # redirect graph to jpg file , bmp or
postscript
(don't forget to dev.off to save the file)
> source("myRfile")                  # to load script and run
```

Reading from a data file e.g. an csv file

```
> mydata <- read.table("PATH-TO/mydata.csv", header=TRUE, sep=",",
row.names="id")
> mydata <- read.csv("PATH-TO/mydata.csv")      # a shortcut for CSV files
```

Writing data out to a file

```
> write.table(mydata, "mydata.txt", sep="\t")
```

For more information about howtos in R,

```
vignette()
vignette("Bioconductor")         # don't forget the quotes
vignette("matchPattern")         # read about matching patterns in
"Biostrings"
```

```
help.start()                    # call for help messages
?blahblah                       # help messages for blahblah
??blahblah                      # search help messages for blahblah
apropos("string")                # where string is the word or regular
                                # expression searched for
example("t.test")               # examples of how to use t.test
```

Install packages

```
> install.packages("package.name")
```

In **R**, missing values are represented by the symbol **NA** (not available).
R is a case-sensitive, interpreted programming language.
And don't forget to add the quotes.

For more information about R and statistical computations, see excellent tutorial:
http://www.statmethods.net/stats/

Explore!

The concept here is that R comes with loads of pre-installed packages

```
> library()                   # to see what packages are installed in the
library
> install.packages()          # choose from a list of packages to install
> install.packages("new")     # install a package called "new"
```