

# Introduction to R

R is a high level language especially designed for statistical calculations. R is free. You can get it at:

<http://www.cran.r-project.org/>

There are versions for Unix, Linux, Windows and Mac. There is a similar program called Splus. The commands in the two languages are virtually identical. Splus has more stuff in it but R is free and it is faster. If you want to use Splus, you can purchase a copy from Insightful at <http://www.splus.mathsoft.com/>.

## 1 Getting Started

In Unix or Linux, you start R by typing: R. In windows, click on the R icon. You can now use R interactively. Just start typing commands.

You can also use R in Batch mode. To do this, store your R commands in a file, say, file.r. In R type: `source("file.r")` which will execute the commands in file.r. In Unix (or Linux), you can also do the following:

```
R BATCH file.r file.out &
```

which will execute the commands and store them in file.out.

**NOTE:** Use the command: `q()` to quit from R.

Use `help(xxxx)` to get help on command xxxx. Better yet, type `help.start()` to open up a help window.

## 2 Basics

Here is a simple R session. The `#` symbol means “comment.” R ignores any command after `#`. I have added lots of comments below to explain what is going on. You do not need to type the comments.

```
x = 5          ### assign x the value 5
x              ### print x
print(x)      ### another way to print x
x <- 5        ### you can also use <- to make assignments
y = "Hello there"
y
y = sqrt(10)
z = x + y
```

```
z
q()          ### Use this to quit
```

Scalars are treated by S-plus as vectors of length 1. That is why they print with a leading “[1]” indicating that we are at the first element of a vector.

Vectors can be created using the `c()` command. `c()` stands for concatenate. Square brackets are used to get subsets of a vector. The colon is used for sequences. Start up R again then do this:

```
x = 1:5          ### the vector (1,2,3,4,5)
print(x)
x = seq(1,5,length=5)  ### same thing
print(x)
x = seq(0,10,length=101)  ### 0.0, 0.1, ..., 10.0
print(x)
x = 1:5
x[1] = 17
print(x)
x[1] = 1
x[3:5] = 0
print(x)
w = x[-3]        ### everything except the third element of x
print(w)
y = c(1,5,2,4,7)
y
y[2]
y[-3]
y[c(1,4,5)]
i = (1:3)
z = c(9,10,11)
y[i] = z
print(y)

y = y^2
print(y)
y = 1:10
y = log(y)
y
y = exp(y)
y
x = c(5,4,3,2,1,5,4,3,2,1)
z = x + y
z          ### R carries out operations on
```

```
### vectors, element by element.
```

If you add vectors of different lengths then R automatically repeats the smaller vector to make it bigger. This generates a warning if the length of the smaller vector is not the same length as the longer vector.

```
x = 1
y = 1:10
x + y
```

```
x = 1:3
y = 1:4
x + y
```

```
x = 1:10
y = c(5,4,3,2,1,5,4,3,2,1)
x == 2          ### This is a logical vector.
z = (x == 2)
print(z)
z = (x<5); print(z)    ### You can put two commands
                        ### on a line if you use a semi-colon.
x[x<5] = y[x<5]      ### Do you see what this is doing?
print(x)
sort(y)
rank(y)
order(y)
o = order(y)
y[o]
```

Two expressions can be written on the same line if separated by a semicolon. One expression can be written over several lines as long as a valid expression does not end a line.

### 3 Matrices and Lists

To create a matrix, use the `matrix()` function as follows:

```
junk = c(1, 2, 3, 4, 5, 0.5, 2, 6, 0, 1, 1, 0)
m     = matrix(junk,ncol=3)
print(m)
m = matrix(junk,ncol=3,byrow=T)
print(m)          ### see the difference?
```

```

dim(m)
y = m[,1]      ### y is column 1 of m
y
x = m[2,]      ### x is row 2 of m
x
z = m[1,2]
print(z)
zz = t(z)      ### take the transpose
zz
new = matrix( 1:9, 3 , 3)
print(new)
hello = z + new
print(hello)
m[1,3]
subm = m[2:3, 2:4]
m[1,]
m[2,3] = 7
m[,c(2,3)]
m[-2,]

x1 = 1:3
x2 = c(7,6,6)
x3 = c(12,19,21)
A = cbind(x1,x2,x3)  ### Bind vectors x1, x2, and x3 into a matrix.
                    ### Treats each as a column.

A = rbind(x1,x2,x3)  ### Bind vectors x1, x2, and x3 into a matrix.
                    ### Treats each as a row.

x = 1:20
A = matrix(x,4,5)    ### Change vector x
                    ### into a 4 by 5 matrix.

dim(A)              ### get the dimensions of a matrix
nrow(A)             ### number of rows
ncol(A)             ### number of columns
apply(A,1,sum)      ### apply the sum function to the rows of A
apply(A,2,sum)      ### apply the sum function to the columns of A

B = matrix(rnorm(30),5,6)
A %*% B            ### multiply matrices
t(A)               ### transpose of A

```

```

x = 1:3
A = outer(x,x,FUN="*")  ### outer product
print(A)
sum(diag(A))  ### trace of A
A = diag(1:3)
print(A)
solve(A)      ### inverse of A
det(A)        ### determinant of A

```

Lists are used to combine data of various types.

```

who = list(name="Joe", age=45, married=T)
print(who)
print(who$name)
print(who[[1]])
print(who$age)
print(who[[2]])
print(who$married)
print(who[[3]])
names(who)
who$name = c("Joe","Steve","Mary")
who$age = c(45,23)
who$married = c(T,F,T)
who

```

## 4 For Loops etc.

A for loop is done as follows.

```

for(i in 1:10){
  print(i+1)
}

x = 101:200
y = 1:100
z = rep(0,100)          ### rep means repeat
help(rep)
for(i in 1:100){
  z[i] = x[i] + y[i]
}

w = x + y

```

```
print(w-z)
```

```
### As this example shows, we can often avoid using loops since  
### R works directly with vectors.  
### Loops can be slow so avoid them if possible.
```

```
for(i in 1:10){  
  for(j in 1:5){  
    print(i+j)  
  }  
}
```

```
### if statements
```

```
for(i in 1:10){  
  if( i == 4)print(i)  
}
```

```
for(i in 1:10){  
  if( i != 4)print(i)      ### != means ‘‘not equal to’’  
}
```

```
for(i in 1:10){  
  if( i < 4)print(i)  
}
```

```
for(i in 1:10){  
  if( i <= 4)print(i)  
}
```

```
for(i in 1:10){  
  if( i >= 4)print(i)  
}
```

You can also use while loops.

```
i = 1  
while(i < 10){  
  print(i)  
  i = i + 1  
}
```

## 5 Functions

You can create your own functions in R. Here is an example.

```
my.fun = function(x,y){
```

```
##### This function takes x and y as input.
##### It returns the mean of x minus the mean of y
a = mean(x)-mean(y)
return(a)
}
```

```
x = runif(50,0,1)
y = runif(50,0,3)
output = my.fun(x,y)
print(output)
```

I like to call give functions names like xxxx.fun but this is not necessary. You can call them anything you like. You can return more than one thing in a function. If you put more than one thing in the return statement, the function returns a list. In the retrun statement, you can attach names to the items in the list.

```
my.fun = function(x,y){
  mx = mean(x)
  my = mean(y)
  d = mx-my
  return(meanx=mx,meany=my,difference=d)
}
```

```
x = runif(50,0,1)
y = runif(50,0,3)
output = my.fun(x,y)
print(output)
names(output)
output$difference
output[[3]]
```

```
### The following function will compute the square root of A:
sqrt.fun = function(A){
  e      = eigen(A,symmetric=TRUE)
  sqrt.A = e$vectors %*% diag(sqrt(e$values)) %*% t(e$vectors)
  return(sqrt.A)
}
```

```
A = diag(1:3)
B = sqrt.fun(A)
print(B)
B %*% B
```

## 6 Statistics

```
x = runif(100,0,1)    ### generate 100 numbers randomly between 0 and 1
y = rnorm(10,0,1)    ### 10 random Normals, mean 0, standard deviation 1
mean(y)
median(y)
range(y)
max(y)
min(y)
sqrt(var(y))
summary(y)

y = rpois(500,4)      ### 500 random Poisson(4)
pnorm(2,0,1)          ### P(Z < 2) where Z ~ N(0,1)
pnorm(2,1,4)          ### P(Z < 2) where Z ~ N(1,4^2)
qnorm(.3,0,1)         ### find x such that P(Z < x)=.3 where Z ~ N(0,1)
pchisq(3,6)           ### P(X < 3) where X ~ chi-squared with 6 degrees
                      ### of freedom
```

## 7 Plots

There are many options related to plotting. You control them with the `par` command, which stands for “plotting parameters.” Type `help(par)`.

```
x = 1:10
y = 1 + x + rnorm(10,0,1)
plot(x,y)
plot(x,y,type="h")
plot(x,y,type="l")
plot(x,y,type="l",lwd=3)
plot(x,y,type="l",lwd=3,col=6)
plot(x,y,type="l",lwd=3,col=6,xlab="x",ylab="y")
plot(1:20,1:20,pch=1:20)
plot(1:20,1:20,pch=20)

par(mfrow=c(3,2))      ### put 6 plots per page, in a 3 by 2 configuration
for(i in 1:6){
  plot(x,y+i,type="l",lwd=3,col=6,xlab="x",ylab="y")
}
```



```

postscript("plot.ps")      ### put the plots into a postscript file
                           ### you have to do this if you use BATCH
plot(x,y,type="l",lwd=3,col=6,xlab="x",ylab="y")
dev.off()                  ### This turns the printing device off.
                           ### This will close the postscript file so you
                           ### can print it.
                           ### Now you can print the file our view it with
                           ### a previewer such as ghostview.

par(mfrow=c(1,1))         ### return to 1 plot per page
y = rpois(500,4)           ### 500 random Poisson(4)
hist(y)                    ### histogram
hist(y,nclass=50)
x = seq(-3,3,length=1000)
f = dnorm(x,0,1)           ### normal density
plot(x,f,type="l",lwd=3,col=4)

x = rnorm(1000)
boxplot(x)

```

## 8 Data Frames and Reading Data From Files

To read in commands or functions from a file rather than typing them in, use `source()`. Put some R commands into a file called `hello`. Try `source("hello")`.

If you have data in a file, you can read it into R using the `read.table` command. Suppose `file.txt` looks like this:

```

2 4 17.2
3 8 12
3 3.4 19
2 52 101.2
1 1 3

```

Read the data as follows.

```
a = read.table("file.txt")
```

This places the data into a data frame. A data frame is like a matrix but is more general. Each column can be a different type of data (character, numeric etc.) Read the help file on `data.frame` and `read.table` for more information.

You can also read data into a vector using the `scan` command:

```

a = scan("file.txt")  ### a is a vector
a = matrix(a,ncol=3,byrow=T)
print(a)

```

## 9 Regression

Here is how to do linear regression in R. First, you should read the help files on the commands `lm` (linear models) and `step` (stepwise regression):

```
help(lm)
help(step)
```

Suppose you have three vectors `y`, `x1` and `x2` and you want to fit the model:

$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \epsilon$$

```
x1 = seq(1,10,length=25)
x2 = runif(25,3,7)
y = 4 + 2*x1 + 7*x2 + rnorm(25,0,1)
mydata = data.frame(y=y,x1=x1,x2=x2)
out = lm(y ~ x1 + x2, data = mydata)
names(out)
extractAIC(out)
s = summary(out)
print(s)
names(s)
par(mfrow=c(2,2))
plot(out,ask=F)
```

Another way to do linear regression is as follows:

```
X = cbind(x1,x2)
temp = lsfit(X,y)
ls.print(temp)
names(temp)
```

To do stepwise regression:

```
out = lm(y ~ x1 + x2,data = mydata)
forward = step(out,direction="forward")
backward = step(out,direction="backward")
summary(forward)
summary(backward)
```

Here are some more regression examples.

```

### Cat example
### heartweight versus brainweight.
library(MASS) ### This is the library from Modern Applied
               ### Statistics in S (Venables and Ripley)

attach(cats)
names(cats)
summary(cats)
postscript("cat.ps",horizontal=F)
par(mfrow=c(2,2))
boxplot(cats[,2:3])
plot(Bwt,Hwt)
out = lm(Hwt ~ Bwt,data = cats)
summary(out)
abline(out,lwd=3)
names(out)
r = out$residuals
plot(Bwt,r,pch=19)
lines(Bwt,rep(0,length(Bwt)),lty=3,col=2,lwd=3)
qqnorm(r)
dev.off()

```

Now have a look at the file cats.ps.

```

### Rats example
postscript("rats.ps",horizontal=F)
par(mfrow=c(2,2))

data = c(176,6.5,.88,.42,
         176,9.5,.88,.25,
         190,9.0,1.00,.56,
         176,8.9,.88,.23,
         200,7.2,1.00,.23,
         167,8.9,.83,.32,
         188,8.0,.94,.37,
         195,10.0,.98,.41,
         176,8.0,.88,.33,
         165,7.9,.84,.38,
         158,6.9,.80,.27,
         148,7.3,.74,.36,
         149,5.2,.75,.21,
         163,8.4,.81,.28,
         170,7.2,.85,.34,
         186,6.8,.94,.28,

```

```

146,7.3,.73,.30,
181,9.0,.90,.37,
149,6.4,.75,.46)

data = matrix(data,ncol=4,byrow=T)
bwt = data[,1]
lwt = data[,2]
dose = data[,3]
y = data[,4]
n = length(y)

out = lm(y ~ bwt + lwt + dose)
summary(out)
plot(out)
infl = lm.influence(out) ### influence statistics
hii = infl$hat
delta.beta = round(infl$coef,3)
st.res = infl$wt.res ### residuals
for(i in 1:3){
  plot(1:n,infl$coef[,i],pch=19,type="h")
  lines(1:n,rep(0,n),lty=3,col=2)
}
plot(1:n,st.res,type="h")
lines(1:n,rep(0,n),lty=3,col=2)
print(data[3,])
par(mfrow=c(1,1))

### remove third case
y = y[-3]
bwt = bwt[-3]
lwt = lwt[-3]
dose = dose[-3]
out = lm(y ~ bwt + lwt + dose)
summary(out)
dev.off()

```

## 10 C functions in R

In Unix and Linux, you can include a C function (or Fortran function) into R as follows (the procedure in Windows is a bit different):

**STEP (1):** Write a C program. Here is an example:

```
#include "stdio.h"
#include "math.h"
#include "stdlib.h"

#define PI 3.14159
#define NMAX 100

double add(double *x, double *y, long *nn, double *out)
{
    long n = *nn;
    int i;
    for(i=0;i<n;i++) out[i] = x[i] + y[i];
}
```

Note 1: All arguments must be pointers.

Note 2: Any variable that is integer in R must be long in C.

**STEP (2):** compile it. Assuming the file is called add.c, the compilation is done as follows:

```
R CMD COMPILE add.c
R CMD SHLIB add.o
```

**STEP (3):** Go into R and type:

```
dyn.load("add.so")
is.loaded("add")
```

**STEP (4):** Write an R function as follows:

```
add.fun = function(x,y){
  n = length(x)
  out = as.double(rep(0,n))
  z = .C("add",as.double(x),as.double(y),as.integer(n),
        out=as.double(out))
  z
}
```

Note: It is best to use `as.double` and `as.integer` to make sure that the variables have the correct attributes.

Note: To return something, you must set aside a variable. For example, the variable `out` is for that purpose. Make sure `out` is the right length.

Now you can use this function just like any other R function. It is also possible to call R functions from C.