R: An Econometrician's Guide

Mirza Trokić

Department of Economics
McGill University
Montreal, QC, Canada
H3A 2T7
mirza.trokic@mail.mcgill.ca

Latest revision: October 13, 2011
Created: September 2011

1 Introduction

R, or more appropriately the R programming language, is a very powerful platform used to conduct a wide array of statistical analyses. It is light, fast, and most importantly, it is free. This latter point has very importantly implications in terms of versatility and adaptability - something we will talk about more when we discuss R packages.

Once opened, the interface of R may intimidate users who are used to nice point and click interface of the Microsoft kind. There are no nice icons and the menu items do not offer much in terms of usability. Nevertheless, this is precisely what one wants if one is looking for efficiency - a property any serious econometrician does not take for granted.

1.1 R Syntax

Let us now turn to the console window and use R to perform some very basic functions. This is done by interacting with the software at the command prompt `>`. Consider the following command:

```
> 2+2
```

This produces the result:

```
[1] 4
```

The point of this exercise is to demonstrate how one interacts with R. In other words, the user enters a command at the command prompt telling R what to do, then, R executes the command and displays or stores the result. Having said this, it is important to notice that R is a programming language and thus, it is only as intelligent as the user speaking said language. Consider the following example. Suppose I want to compute the logarithm of 10 with base 10. If I execute the command:

```
> log(10)
```

This yields:

```
[1] 2.302585
```

The correct response however is 1. What happened? Surely for such an important piece of software such an embarrassing mistake is not befitting! Well, the reason for the error is that R did not make the mistake. The user did! By executing the log function `log(10)`, R understood this to mean take the natural logarithm of 10. And indeed, the answer above is correct. However, because we wanted the logarithm with base 10, we should have specified this in R through the following:

```
> log10(10)
```

This indeed does yield the answer we desired, namely:

```
[1] 1
```

As exemplified by the above, it is crucial to realize that in R, as in all other programming languages, the output is only as good as the input! In fact, consider another very common mistake. Suppose we define the variable `x` to be equal to two through the following command:

```
> x = 2
```

And suppose now we want to compute twice the value of `x`. If we were to follow the convention we often use when writing down mathematical operations, we would be inclined to execute the following:

```
> 2x
```

Unfortunately, this would yield the output:

```
Error: unexpected symbol in "2x"
```

This is because R does not understand what `2x` means. Had we been a little more precise and used the multiplication operator `*`, we would have obtained the correct response through the following:

```
> 2*x
[1] 4
```

On a related note, it is important to keep in mind that mathematical operators are not all executed with the same priority. For example, recall that exponentiation is executed before multiplication or division which are executed before addition or subtraction. Thus, if we wanted to compute half the value of twice more the square of `x`, executing something like:

```
> 2+x^2/2
```

would yield:

```
[1] 4
```

The answer we were looking for however is 3. The reason for the discrepancy is that we were not careful with what we were executing. What we required was the following input:

```
> (2+x^2)/2
```

which gives the desired answer:

```
[1] 3
```

1.2 R Scripts

We mentioned earlier that R is a very efficient software. One way this efficiency is manifested is through something known in the programming community as a script. A script is a set of instructions akin to what we were doing earlier which tells the software what to do. In the simplest terms, this means that we can write as many instructions as we wish and tell the computer to execute them in the order we wish without the need for live interaction. For instance, say we wanted to obtain the results above at some later time without having to write all of the functions from scratch. In this case, we can tell R to execute all of our functions automatically by writing the script in any text editing program including the standard Windows notepad. Once written, the file should be saved with extension `.r`. Then, the script can be executed. We will return to this at a later time.

1.3 The Question Mark

Whether you are a master programmer or just starting out today, your best friend in R is the `help()` function! Suppose we know the exact name of a function in R that we need help with. For instance, consider the `log` function. If we want to see what help R can offer on this function, we execute the following command:

```
> help("log")
```

This takes us to a page which allows to browse all the information about the said function. This includes information such as how the function should be invoked, what arguments it takes, what outputs it prints, etc. Here, we can scroll through the information using the "up" and "down" keys, and return to the console window by pressing the "q" key.

Suppose on the other hand that we are not sure what the exact name of the function is. In this case, we can search for keywords instead. Let us search for the keyword "logarithm" via the following command:

```
> help.search("logarithm")
```

The latter command brings us to a list of all functions in R that match this keyword. Moreover, note that the output is split into two columns. The first column displays the functions associated with this keyword, while the second displays the description. More importantly,

take a closer look at the first column. Here, the output is organized as `library::function`, or in our particular case, we have:

```
base::log
nlme::logDet
```

What this says is that our search has found two functions in R. The first is the function `log` which belongs to the "base" package. The second function is the `logDet` function which belongs to the "nlme" package. From here, we can press the "q" key to return to the main console window and then proceed to read the help file on the `log` via the `help` function discussed above.

Now, before proceeding to discuss packages, note that the functions `help` and `help.search` can also be invoked via `?` and `??` respectively. That is `help("log")` is the same as `?"log"` and invoking `help.search("logarithm")` is equivalent to executing `??"logarithm"`.

1.4 Packages

One of the things which makes R so powerful and versatile is its ability to use so called packages. Packages here can be thought of as sets of functions and scripts aimed at automating operations which would otherwise be very difficult to perform if we had to write them out every time we needed them. For example, suppose that what we want to obtain requires invoking 10 different functions in a specific order, on a set of arguments. If we had to invoke these 10 functions every time we needed this result we would quickly become very frustrated. Instead, suppose we write a script which combines these 10 functions into a single function. Then, all we have to do is invoke our one function and specify only the arguments. R then takes care of the rest and we obtain our desired result.

A package then is a collection of functions and scripts aimed at performing specialized statistical analyses. For instance, there are packages that deal only with operations used in time series. Others only deal with cross-sectional data. Yet others deal with graphing, etc. All in all, there are thousands of packages available for use. They can all be downloaded and they are all free. However, before one goes and downloads the first package one finds, it is of importance to note that packages can be written by anyone, from Ph.D. students to prominent world-class authors, and so it's important to use packages which will do the job properly!

To see the list of packages which are already installed with R, we invoke the following command:

```
> library()
```

This brings up a page which lists all the packages installed in R and their description. Suppose we are interested in the package called "stats". To use this package we must loaded it

into the current R session. We do so as follows:

```
> library(stats)
```

Now that we have loaded the package what we are really after are the functions that come bundled in it. To see what these functions are, we have to read the documentation of the package. This is done as follows:

```
> library(help = "stats")
```

The above command produces a list of the functions and scripts in the package "stats" along with short descriptions. Suppose that what we are seeking is a function which provide quantiles and percentiles from the Normal distribution. It seems only natural to consider the `Normal` script. Because we loaded the package to which this script belongs earlier, we can read the documentation on it as follows:

```
> help("Normal")
```

Reading the said documentation, we see a section at the top titled "Usage", which tells us what functions can be invoked. The section immediately below this one, titled "Arguments", tells us what arguments need to be specified when we invoke the desired functions. Suppose we are interested in the quantile which characterizes the right-most 2.5% of the standard Normal distribution. What we want to use then is the `qnorm` function as follows:

```
> qnorm(.975)
[1] 1.959964
```

This answer should be very familiar. Suppose however that we require the same quantile but from the Normal distribution with mean 1 and variance 2, or equivalently, the Normal distribution with mean 1 and standard deviation $\sqrt{2}$. Consulting the documentation we realize that this can be done as follows:

```
> qnormal(.975, mean = 1, sd = sqrt(2))
[1] 3.771808
```

Consider now that while browsing the for R functions and packages, we realize that the function we want is part of package which we do not have installed in R. Luckily, downloading packages is a trivially easy task in R. Assuming that we have a working internet connection, a package can be installed by invoking the following command:

```
> install.packages("nameofpackage")
```

As an example we will download the package called "fortunes". This package is guaranteed to eat away at your productivity, but it's a must have for anyone looking for yet another way to procrastinate! Without further delay, let's download this time waster:

```
> install.packages("fortunes")
```

Once executed, R will ask you to choose a server from which you wish to download the package. Here you want to choose the server which is closest to your physical location so as to have the fastest download rate. Once the download is complete we will load the package as follows:

```
> library("fortunes")
```

Now, a quick look at the documentation reveals that the package "fortunes" has only one function, namely `fortune`. So let us invoke this function a few times and have some fun:

```
> fortune()
```

It is now clear what the function does. So anyone looking to waste some time while giving the illusion to others that you are working, invoke the R fortunes function.

1.5 Setting Working Directory and Exiting R

It is often useful, not to mention cleaner to have a dedicated directory whenever working on a particular project. This directory will serve to store your data files, your saved files, your outputs, etc. Telling R what this directory is the subject of this section. Suppose we have created a folder called "learningR" in the root directory "C:". Then, setting said directory as the working directory is done as follows:

```
> setwd("C:/learningR/")
```

That's it. This directory is now stored into memory and any time R is opened it will be working out of said directory. In this regard, suppose down the road we forget where our files are stored and we wish to be reminded. This can be done as follows:

```
> getwd()
[1] "C:/learningR/"
```

Now that we have the basics of R, let us proceed to quit the program. This is done by invoking the following command:

```
> q()
```

Upon invoking the quit function, R will ask:

```
Save workspace image? [y/n/c]
```

Then, pressing "y", "n", or "c", corresponds to "yes", "n", or "cancel", respectively. If we press "y" R will exit having saved our variables. Next time we open R those same variables will be loaded automatically making the process very efficient.

2 Data Input

Entering data into R can be accomplished through several means. The most laborious of these methods is entering each data point individually. The first step in this direction is defining variables with a single point. In this regard, let us create the variables labelled "a", "b", "c" which taking on the values 1,2, and 3, respectively as follows:

```
> a = 1
> b <- 2
> c = 3
```

The first thing to note above is that the operators `=` and `<-` are equivalent and so either one can be used. The second point to realize is that when we define variables, R does not display no output in the console. Does this mean that nothing happened? No! To see that we did in fact succeed in creating said variables, proceed by typing the variable name at the command prompt and executing:

```
> a
[1] 1
> b
[1] 2
> c
[1] 3
```

Another way to see that we did indeed succeed is to take a look at the list of all the variables created. This is done through the so called `ls` function as follows:

```
> ls()
[1] "a" "b" "c"
```

2.1 Multiple Data Points

In the previous subsection we demonstrated how to create a variable with a single data point. Here, we would like to extend this principle and create a variable with multiple data points.

This is achieved with the "concatenation" function.

Table 1: Car Data

	MPG	EngineSize	Horsepower	Weight
1	25	1.80	140	2705
2	18	3.20	200	3560
3	20	2.80	172	3375
4	19	2.80	172	3405
5	22	3.50	208	3640
6	22	2.20	110	2880
7	19	3.80	170	3470
8	16	5.70	180	4105

Suppose we have data describing various characteristics of car models such as the mpg in the city, engine size, horsepower, and weight, as summarized in Table 1. To enter the data from into R we need to create five variables each containing eight data points. We do this as follows:

```
> mpg = c(25, 18, 20, 19, 22, 22, 19, 16)
> es = c(1.8, 3.2, 2.8, 2.8, 3.5, 2.2, 3.8, 5.7)
> hp = c(140, 200, 172, 172, 208, 110, 170, 180)
> wt = c(2705, 3560, 3375, 3405, 3640, 2880, 3470, 4105)
```

Let's take a look at our "mpg" variable and see what comes up.

```
> mpg
[1] 25 18 20 19 22 22 19 16
```

What was created was an 1-by-8 vector labelled "mpg" containing the eight points we entered earlier. The fact that our variables are vectors allows us to view, edit, or delete any range of values we wish. For example, suppose we wish to know the value of the third data point in the vector "es". We can do this using square brackets as follows:

```
> es[3]
[1] 2.8
```

On the other hand, we can look at points three through 6 of the vector "hp" using the colon notation like so:

```
> hp[3:6]
[1] 172 172 208 110
```

Perhaps we want to look at every point except the fifth in the vector "mpg". This can be done as follows:

```
> mpg[-5]
[1] 25 18 20 19 22 19 16
```

Or more complicated still, perhaps we want to look at every point except the third and the fifth. This can be done using the `c()` function as shown below:

```
> mpg[c(-3, -5)]
[1] 25 18 19 22 19 16
```

Before moving on let us take a look at how to edit points as well. If we we made a mistake in entering the second point of the "mpg" vector by typing 18 instead of 19, we can easily rectify this and verify that we made the correct change:

```
> mpg[2] = 19
> mpg
[1] 25 19 20 19 22 22 19 16
```

Finally, note that similar changes can be made on more than one data point using the concatenation operator.

2.2 Stacking Variables

Now that we've seen how to create variables into R, let's take a look how to stack them. Stacking variables allows us to combine two vectors for example into a matrix, or, to append vectors to existing matrices thereby increasing their dimension.

The fist thing we need to notice here is that `c()` must not be used carelessly. For example, if we take the four eight-by-one variables we created earlier and combine them using the concatenation function we do not obtain an 8-by-4 matrix, rather, we obtain a 32-by-1 vector as can be seen from the following:

```
> combo = c(mpg, es, hp, wt)
[1] 25.0 18.0 20.0 19.0 22.0 22.0 19.0 16.0 1.8 3.2
[11] 2.8 2.8 3.5 2.2 3.8 5.7 140.0 200.0 172.0 172.0
[21] 208.0 110.0 170.0 180.0 2705.0 3560.0 3375.0 3405.0 3640.0 2880.0
```

```
[31] 3470.0 4105.0
```

So why would anyone want to do this then? Well, as we will see later on in the course, we can use indicator functions to extract the values we want from large vectors. This type of analysis usually comes in handy when we deal with seasonality. But more on this at a later time. It suffices to say that when a specific value of the indicator function is called, the value we want is extracted. We now focus on creating such an indicator function. In essence, we want a variable defined as follows:

```
> id = c(1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3,
3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4)
[1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4
```

So, using the "id" variable, we can map the values from the "combo" variable as follows. Whenever a 1 is called, we obtain the corresponding value in the "mpg" variable, when a 2 is called, we get the corresponding value from the "es" variable, and so on. It goes without say that this procedure is very inefficient for larger data sets. Luckily, R has a very convenient solution through the `rep()` function. Consider the following:

```
> id = rep(c(1,2,3,4), each=8)
[1] 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4
```

Obviously the two "id" sequences are identical. However, as long as we are on the subject of sequences, let us look at the "seq()" function. The latter is used to create sequences of various sorts. Suppose, we wanted to create a sequence of 1 to 4, by increments of 1. That is, we want a sequence reading 1, 2, 3, 4. This can bedone as follows:

```
> s4 = seq(1,4, by = 1)
[1] 1 2 3 4
```

It is easy to see how this can be extended to various other combinations. Suppose for example we wanted a sequence of 1 to 16 which moves in increments of three. This is done as follows:

```
> s16.3 = seq(1,16, by = 3)
[1] 1 4 7 10 13 16
```

It is also important to note that we do not necessarily have to restrict ourselves to numbers when creating variables or sequences. That is, suppose we don't like a numerical way of identifying our variables, but we want to keep track of the names. This too can be done as follows:

```
> idnames = c('mpg', 'es', 'hp', 'wt')
```

```
[1] "mpg" "es" "hp" "wt"
```

Then, we can create a 1-by-32 vector of names as follows:

```
> idnames2 = rep(idnames, each = 8)
```

Also, a very important point here is that the argument "each" is essential. If it is omitted, we will get something very different from what we had intended. To see this, note:

```
> idnames3 = rep(idnames, 8)
[1] "mpg" "es" "hp" "wt"
[5] "mpg" "es" "hp" "wt"
[9] "mpg" "es" "hp" "wt"
[13] "mpg" "es" "hp" "wt"
[17] "mpg" "es" "hp" "wt"
[21] "mpg" "es" "hp" "wt"
[25] "mpg" "es" "hp" "wt"
[29] "mpg" "es" "hp" "wt"
```

But looking at the above structure, we see that our original data is in precisely the above form. The problem is, we have four 8-by-1 vectors and our data is in matrix form. So how do we go about stacking our data to resemble Table 1 above? The answer lies in the "column bind" function. The latter allows us to stack vectors horizontally. To see it in action, observe the following:

```
> table = cbind(mpg, es, hp, wt)
> table
```

Executing the above expresses the data in the form of Table 1. In other words, the output is no longer a vector, but an 8-by-4 matrix. As such, we should learn a few commands which allow us access and manipulate various points of matrices. For example, if we want the element found in row 3, column 4 of the matrix "table", we issue the command:

```
> table[3,4]
wt
3375
```

On the other hand, suppose we want all of the points in the second column, that is, we want to extract the values of the variable "es". This can be done like this:

```
> table[,2]
[1] 1.8 3.2 2.8 2.8 3.5 2.2 3.8 5.7
```

Similarly, we can do something similar if we wanted to obtain all the values in row 7 as follows:

```
> table[7,]
 mpg es hp wt
19.0 3.8 170.0 3470.0
```

And finally, if we want to find out the dimension of any variable, we use the `dim()` function as folllows:

```
> dim(table)
[1] 8 4
```

Now, let us look at the row binding function `rbind()`. This function is analogous to the `cbind()` function but instead of stacking variables horizontally, it stacks them vertically. Let us repeat recreate the table above but with the variable names indicating the rows rather than columns. This is done as follows:

```
> table2 = rbind(mpg, es, hp, wt)
table2
```

This type of operation is known as a transpose in mathematics and luckily, there is a very quick way of doing it using the `t()`. Consider the above operation expressed as follows:

```
> table = cbind(mpg, es, hp, wt)
table
table = t(table)
table
```

Finally, note that all this time, we've been labelling our variables in a manner which may confuse some. That is, we've been using the variable name as both the name and the argument. In this case, note that the naming occurs AFTER the action on the argument. Thus, looking at the third command above, we see that the existing variable "table" is transposed first, and then it is named "table" again. This type of naming scheme clearly overwrites existing variables which have the same name.

2.3 Basic Matrix Operations

The `matrix()` function allows one to create a matrix of any dimension and fill it with whatever data we please. Consider first the idea of creating an empty matrix of dimension 8-by-4.

```
> mat = matrix(nrow = 8, ncol =4)
> mat
```

As can be seen by executing the above, the matrix we created is precisely an 8-by-4 matrix of empty values which R labels as"NA". We can now proceed to fill in the values by using one of the methods of element manipulation discussed earlier. But recall that our fist attempt at using the concatenation operator on the 4 variables found in Table 1 produced a 32-by-1 vector. Well, we can now transform that vector into a matrix of conformable size as follows. Because we have 32 elements, we can create an 8-by-4 matrix from it as follows.

```
> combo = c(mpg, es, hp, wt)
> mat = matrix(combo, nrow = 8, ncols = 4)
> mat
```

Thus, we see that we obtain the data in the form of Table 1, although there are no column names. This can be fixed using the `colnames()` function like thus:

```
> colnames(mat) = idnames
> mat
```

Now, let us proceed to look at some very useful matrix operations. We will start by creating the identity matrix. The easiest way to do this in R is to use the `diag()` function which is typically used to create diagonal matrices. Consider the identity matrix of size 6 for example:

```
> ID = diag(6)
> ID
```

The above command may be used in several other ways as well. For example, it can create a matrix whose diagonal elements can be anything we want, and any dimension we want, not necessarily squared. Consider a diagonal matrix of dimension 5 elements of which take on the value 3.4 for instance.

```
> D = diag(3.4, 5)
> D
```

Or, consider a matrix of dimension 6-by-8 whose diagonal elements of the first block take on some value, say 4.1. This is done as follows:

```
> D = diag(4.1, 6, 8)
> D
```

Suppose now that I want my diagonal elements to take on some vector of values, not just one repeating number. Let's use some notation we used so far. Suppose I want a diagonal matrix of dimension 5 whose diagonal elements are the values of the sequence running from 1 to 10 in increments of 2. We can accomplish the task as follows:

```
> s = seq(1,10,2)
> D = diag(seq,5,5)
> D
```

And finally, the `diag()` function can be used to extract the diagonal elements of some matrix. This in turn creates a vector of the diagonal elements. So, let us recover the diagonal elements of the matrix above as follows:

```
> d = diag(D)
[1] 1 3 5 7 9
```

Now, suppose I want to multiply two matrices together. Unlike what you may think, using the multiplication operator `*` naively will NOT give you the correct answer. This is because the latter is used for element by element multiplication, whereas matrix multiplication has its own syntax. Let's take a look at an example.

Consider two matrices A and B as follows:

$$A = \begin{bmatrix} 1 & 0 & -2 \\ 0 & -3 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix}$$

The product AB is the following:

$$\begin{bmatrix} 1 & 0 & -2 \\ 0 & -3 & 1 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ -2 & -1 \\ 0 & 4 \end{bmatrix} = \begin{bmatrix} 0 & -5 \\ -6 & -7 \end{bmatrix}$$

Thus, consider doing the same in R. We begin by creating the two matrices as follows:

```
> A = c(1, 0, 0, 3, -2, 1)
> A = matrix(A, 2, 3)
> A
> B = c(0, -2, 0, 3, -1, 4)
> B = matrix(B, 3, 2)
> B
```

If we were now to naively issue the `*` operator in order to multiply the two matrices above, we would get an error. This is because matrix multiplication is done through the following syntax:

```
> C = A%*%B
> C
```

This indeed does give us the correct answer. However, suppose now that I want the inverse of this matrix. This can be done using the `solve` command as follows:

```
> Cinv = solve(C)
> Cinv
```

Notice that the above is written in decimals, but the numbers look relatively nice. This indicates that they may be nicely expressed as fractions. How would we do this. Well, `fractions()`, the function which converts floating points into fractions is located in the "MASS" package. We loaded it as follows:

```
> library(MASS)
```

Now, we proceed to express our matrix in fractions as follows:

```
> Cinv = fractions(Cinv)
> Cinv
```

Finally, let's at look at how to compute the determinant and eigenvalues and eigenvectors of a matrix by computing it for the matrix "C" above. Consider first the determinant computation:

```
> det(C)
[1] -30
```

And now, let's compute the eigenvalues and eigenvectors like so:

```
> eigen(C)
```

We see that executing the above command gives us both the eigenvalues and the vectors associated with them.

2.4 Combining, Importing, and Exporting Data

Now that we have created some data, let us look at a very useful R construction, the "data frame". This is something very similar to the matrix of data we created earlier, but much more versatile. In fact, imagine the data frame as a vault in which you can place any data of both numerical and string types, as long as they are conformable. Moreover, a data frame is never operated on, but rather, one simply draws what one needs from it and then proceeds to manipulate the data on the extracted data. Thus, a data frame can be considered as a

vault from which one draws information, and it serves as an archive of the data.

For example, recall that we created a matrix of the car data by combining our four variables. But now, let us package those same four variables into a data frame as follows:

```
> dtfrm = data.frame(MPG = mpg, ES = es, HP = hp, WT = wt)
> dtfrm
```

Now, the first thing to notice is that the variables MPG, ES, HP, and WT are NOT in the variables list. To see this, just issue the `ls()` command. This is because the latter variables are in the archive so to speak and they are not meant to be manipulated, but rather used. So, let us for example use the variable MPG in the data frame to create a new variable which is twice its value. This is done using the $ operator as follows:

```
> mpgx2 = 2*dtfrm$MPG
```

So what's the big deal? We could have done the same thing with a matrix and vector constructions. True, but the importance lies in the fact a data frame can contain vectors of strings in addition to numbers, so long as they correspond to the same data and are conformable. So our data is in a different container and it is NOT a matrix! In fact, if we had a vector of size 8 consisting of string values, we would be able to add that as well. This is also relevant because most data one finds or uses usually comes in a data frame container.

Suppose then we have some data we wish to use. How do we import it? This is perhaps one of the most useful operations in R and should be committed to memory. Let us begin by noting that most data is either in its raw ".dat" or ".txt" form, and usually comes with variable headers. For example, consider the following set of data taken from the net, describing some quarterly, log-transformed time series from the German economy starting from 1960.

The above data is located in our working directory under the name "e1.dat". Let us therefore attempt to load it into R. The first thing to notice is that this data is nicely ordered into columns and has column headers, or variable names. So, the appropriate command to load the data will be:

```
> e1 = read.table("e1.dat", header = TRUE)
```

Issuing the above command creates a data frame using the original data. Thus, it is not a matrix and so, any manipulations on the data need to be done on constructions we take henceforth.

A related concept of course is data exportation. This is done in an analogous, but reverse manner. Let us suppose that we have some object in R that we wish to save; perhaps a data

Table 2: German Time Series

	invest	income	cons
1	5.19	6.11	6.03
2	5.19	6.14	6.04
3	5.22	6.18	6.07
4	5.26	6.20	6.10
5	5.35	6.23	6.13
⋮	⋮	⋮	⋮
90	6.71	7.87	7.71
91	6.72	7.87	7.72
92	6.72	7.88	7.73

frame, a matrix, a vector, etc. This can be done as follows:

```
> write.table("object", "nameoffile.txt", col.names = TRUE)
```

Suppose we name our file "rdata.txt" and the object we save is data frame "dtfrm". Then, our command will look like this:

```
> write.table("dtfrm", "rdata.txt", col.names = TRUE)
```

So where is this file? Well, if you recall, last time we defined our working directory using the `setwd()` command. So, our file is located in our working directory, which, if we needed a reminder, could be displayed with the `getwd()` command.

It is crucial to note that when exporting data with the `write.table()` command, the object we save is a data frame. So, if had saved the matrix "A" we created earlier using the above method, it would no longer be a matrix when we load it via the `read.table()` command. So how do we save and load variables themselves? This is done with the `save()` and `load()` functions.

The `save()` command takes the following syntax:

```
> save(var1, var2, var3, etc.., file = "nameoffile.rda")
```

Let us use the above to save the variables "idnames", "A", and "dtfrm" into a file called "rdata.rda". This is done as follows:

```
> save(idnames, A, dtfrm, file = "rdata.rda")
```

Now, let us proceed to delete those same variables from our current R session. This is done using the `rm()` command like this:

```
> rm(idnames, A, dtfrm)
```

But let's suppose we now want to use those same variables again. We can load them from our saved file as follows:

```
> load("rdata.rda")
```

Again, recall that all files are saved to and loaded from the working directory.

3 Plotting

One of the things R is renown for is plotting capabilities. It is arguably one of the most powerful plotting platforms in existence today! In this section we will cover some of these capabilities in pragmatic detail.

In the previous section we briefly loaded the German investment data "e1.dat". This data comes from Lutkepohl's jMULTi website under the data section for his book *New Introduction the Multiple Time Series Analysis*. If the data is already loaded into R we can proceed to the next paragraph. However, if it's not, let's load it first:

```
> e1 = read.table("e1.dat", header = TRUE)
```

Now, using the function `is.data.frame()`, one can verify that the element "e1" is indeed a data frame. This means we can use variables using the $ operator. However, before doing so, let's take a look at what variable were actually loaded using the names command.

```
> names(e1)
["invest" "income" "cons"]
```

Thus, we have three variables under the above headings. Let's look at a primitive plot of how the data variables behave in relation to each other. In other words, let's look at a scatter plot. We'll start first by comparing "invest" and "income".

```
> plot(e1$invest, e1$income, xlab = "Investment", ylab = "Income", main
= "German Investment Data", sub = "Income vs. Investment")
```

As can be seen, the first 2 arguments are the variables on the x and y axes respectively, while the last two arguments are the labels we wish to impose on the respective axes. However,

suppose, we wanted to determine the inverse relationship. One way is to switch the first two arguments, but another way is as follows:

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment")
```

3.1 Working With Symbols

By default, R prints plots using circles as points. This can be changed however, and sometimes it's necessary to do so. Different points are represented by numbers running from 1 to 255 so there is a large variety. The argument which needs to be specified in the plot function is "pch". Let's look at some examples.

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = 2)
```

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = 3)
```

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = 12)
```

But suppose now that I only wish to isolate a few points and represent them by different points for easy identification. In this case, I can create a vector the size of my data set with each number representing a "pch" we would like to use. For concreteness, suppose we want to isolate points 10, 20, and 30 and we want to represent these three points with three different symbols, leaving all other points unchanged. A quick application of the `dim()` function tells us that each of our variables has 92 points. So consider the following:

```
> threep = rep(1,92)
> threep[c(50,60,70)] = c(2,3,12)
> threep
[1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[26] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2
[51] 1 1 1 1 1 1 1 1 1 3 1 1 1 1 1 1 1 1 1 1 12 1 1 1 1
[75] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep)
```

3.2 Working With Colours

Color can be a very important aspect of graphs so it merits our attention. Colour is activated via the "col" argument. Its functionality is identical to that of "pch". Thus, we repesent

colours using numbers or vectors of numbers. Let's see some examples.

```
   > plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = 3)
```

, Or, if we want to represent our three points in red, consider:

```
   > threec = rep(1,92)
   > threep[c(50,60,70)] = rep(2,3)
   > plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = threec)
```

Or, if you've been eating Skittles all day and want to see the rainbow, try this:

```
   > rnbw = c(1:92)
   > plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = rnbw)
```

3.3 Working With Sizes

Another very important argument is "cex". Its function determines the size of points and its usage is identical to "pch" or "col". As a quick example, let's enlarge our three points.

```
   > threes = rep(1,92)
   > threes[c(50,60,70)] = rep(2,3)
   > plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = threec, cex = threes)
```

And to close, consider a very fun example:

```
   > library(MASS
   > rnbws = abs(mvrnorm(92,1,1))
   > plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = rnbw, cex = rnbws)
```

What we did above was as follows. First, we loaded the "MASS" package. This is because we want to use the function `mvrnorm()` to simulate points from the Guassian (Normal) distribution with mean 1 and standard deviation 1. This is so that we can get a random collection

of points which we can then use as the basis for our sizes. Finally we use this vector and create a plot whose points are random, following the aforementioned distribution.

To close this section, let's briefly take a look at the size, or length in this case, of the axes. Using the parameters "xlim" and "ylim" we can adjust the limits of our axes. So, as an example, we will limit the x-axis to the interval [750,2250] and the y-axis to the interval [350,650]. This yields the following:

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", pch = threep,
col = rnbw, cex = rnbws, xlim = c(750,2250), ylim = c(350,650))
```

3.4 Superimposing Lines

Sometimes it is necessary or useful to superimpose lines on the original graph in order to indicate the presence of something of importance. For example, suppose that for some reason, the number 500 on the y-axis is of importance. In this case, we can create a horizontal line on this axis crossing 500 at the origin. Consider the following code:

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", cex = 1.5)
```

Now that we have our graph, let's plot the horizontal line.

```
> lines(x = e1$income, y = rep(500,92))
```

Or, if we wanted a horizontal line at 1000, the following code will suffice.

```
> lines(y = e1$invest, x = rep(1000,92))
```

However, as important as such lines may be, a far more important superimposition in practice is that indicating a smoothed fit for the set of points. This is achieved through a combination of functions which should be studied using the help functions.

The smoothing method we will use is called LOESS smoothing function and we can see it on our graph using the following sequence of commands:

```
> m.loess = loess(invest   income, data = e1)
> fit = fitted(m.loess)
> lines(e1$income, fit, col = 2, lwd = 2, lty = 3)
```

So what did we do? First fitted a LOESS model to the relationship between investement and income. Secondly, we extracted the set of fitted points from this model. This is in a

sense the regression line form a non-linear regression model. Finally, we superimposed this fitted line on our original scatter plot. However, a picture tells a thousand words so to speak and our graph is no different. Moreover, using the "col" argument in the `lines()` function we were able to change the colour of the smoothing curve, while using "lwd" and "lty" we were able to change the width and style of the line respectively.

3.5 Saving Plots

Now that we know how to create plots. Let's take a look at how to save them. The method of saving plots is a multi step process in R. First, we need to issue the `jpeg()` function and specify the name of the file we wish to save our plot to. Let's do that now.

```
> jpeg("savedplot.jpeg")
```

Next, we create our plots using the methods covered in the previous subsections. However, this time, when we issue the `plot()` commands, no output will be shown. This is because the plots will be directly written to the file we specified. So, let's recreate our plot with the smoothing line.

```
> plot(y = e1$invest, x = e1$income, xlab = "Investment", ylab = "Income",
main = "German Investment Data", sub = "Income vs. Investment", cex = 1.5)
> lines(e1$income, fit, col = 2, lwd = 2, lty = 3)
```

Once we are satisfied, we can turn off the "save-to-jpeg" environment by issuing the following command:

```
> dev.off()
```

Now, as before, issuing `plot()` commands will produce the plot on the screen.

3.6 Plot Annotations

Now, learning a few more tricks of the trade, let's put what we've learned together into a very nice example. What we want to do here is to draw points from the Gaussian density, plot those points, and then draw the theoretical density of the latter distribution on top of those points to show that they do indeed fall on the said curve. Finally, we want to annotate the graph with textual and mathematical expressions.

Our first task here is the simulate a large number of points from the standard normal distribution. We saw earlier that this can be done using the `mvrnorm()` function from the "MASS" package.

```
> xgauss = mvrnorm(100,0,1)
```

Now that we have obtained the drawings, it's time to convert these drawings into the values of the density that correspond to them. This can be done by making use of the `dnorm()` function from the "stats" package we looked a few lessons ago. This function takes the vector of inputs and yields the vector of values of the density that corresponds to this vector of inputs.

```
> ygauss = dnorm(xgauss)
```

We now have the points we wanted to simulate. Now, let's use the `curve()` function to plot the theoretical curve from the standard normal density. Adding of course some titles, colours, and adjustements, this can be done as follows:

```
> curve(dnorm, from = -3, to = 3, col = 24728, lwd = 3, main = "Standard
Normal Density", ylab = expression(phi(x)), xlab = expression(x))
```

Next, let's see how our points compare to this graphic. To superimpose our points we will use something very similar to the `lines()` function we used earlier. We will use the the `points` function.

```
> points(xgauss, ygauss, col = 2)
```

Next, let's add some lines for the mean and standard deviation.

```
> lines(rep(0,100),seq(0,3.96,0.04), lwd = 2, col = 4)
> lines(rep(-1,100),seq(0,3.96,0.04), lwd = 2, lty = 3, col = 3)
> lines(rep(1,100),seq(0,3.96,0.04), lwd = 2, lty = 3, col = 3)
```

Now, let's annotate our graph to make it look very pretty. Let's start with the mathematical expression for the standard normal density. This is achieved using the "expression" argument.

```
> text(-2.75,0.2, expression(f(x) == frac(1, sigma ~~
sqrt(2*pi)) ~~ e^{-frac((x - mu)^2, 2*sigma^2)}),
adj = 0, cex = 1.25)}
```

Next, add the annotations for the mean and standard deviation.

```
> text(0,.15, expression(mu), cex = 1.25)
> text(.5,.15, expression(+sigma), cex = 1.25)
> text(.5,-.15, expression(-sigma), cex = 1.25)
```

And finally, if we really wanted to see what distribution our sample takes on, we can add tick marks to each of the axes using the `rugs()` function as follows:

```
> rug(xgauss)
> rug(ygauss), side = 2
```

At the end, when we see how pretty our graph looks, we can see that our effort was not in vain! Nedless to say, the possibilities of making pretty graphs is endless in R, so have fun!

The material covered so far should be enough for a basic understanding of the `plot()` function and related concepts. Due to the fact that plotting in R is a vast area of expertise, additional concepts shall be covered as they arise throughout the remaining material.

4 Data Exploration

In this brief section, and before delving into more serious matters of linear regression, let's take a look at some primitive means of data analysis. To start, let's load the data set we looked at earlier on car makes and specifications. This data set is located in the "MASS" library. If it's not loaded, load it now, and then proceed to load a data set called "Cars93" as follows.

```
> data("Cars93")
```

Now that the data has been loaded, let's look at some facts whcih characterize it. Start with the function `str()`. This should give us a very primitive overview of what the variables are, their structure, size, etc.

```
> str(Cars93)
```

The above tells us that our data is contained in a data frame, it has 93 observations, 27 variables, and then proceeds to tell us what those variables are and the type of data contained in them. Here, "Factor" means that the data is a cateogrical indicator, "Num" indicates it's numerical, therefore floating point, and "Int" implies that the data is an integer, therefore whole numbers.

We can also view the first and last 6 entries of this data table using the `head()` and `tail()` functions respectively.

```
> head(Cars93)
> tail(Cars93)
```

Yet a more detailed description is obtained using the `summary()` command, which, in addition to descriptors we've seen so far, also yields descriptive statistics for numerical and integer-based variables.

```
> summary(Cars93)
```

Here on forth we will need to access the variables in this data set. As we've seen so far, we've been doing this using the $ operator. For example, if we want to "MPG.city" variable, we would obtain it via the command:

```
> Cars93$MPG.city
```

As this can get very tedious very quickly, we're going to introduce a new command called `attach()`. This command will attach the data set "Cars93" directly into the R search tree so that we can access variables directly by their names. Consider:

```
> attach(Cars93)
> MPG.city
```

A word of caution here. It is very important that when using the `attach()` command that there are no variables with conflicting names. That is, if we had attached the "Cars93" and wanted to use the variable "MPG.city" that was NOT in the "Cars93" data.frame, we would not get it. We would get the variable "MPG.city", but the one in the "Cars93" data frame. Thus, it is very important to name variables appropriately. Moreover, when done using the data, it should be detached in a similar fashion using the `detach()` command.

5 Linear Regression

The linear regression model is the foundation of modern econometrics. As such, learning how to work with it is of the utmost importance. Its typical form in most of the literature is presented as the following vector expression:

$$y = X\beta + \epsilon$$

Above, y and ϵ are $nx1$ vectors, X is an nxk matrix, while β is a $kx1$ vector. Moreover, y is the regressand vector, or the endogenous variable, whereas as X is the matrix of regressors, or the matrix of exogenous variables. Furthermore, as is typical in this context, we assume that the error terms which constitute the ϵ vecotr, are independent of the regressors and are IID with mean zero and variance σ^2. Mathematically, this implies the following two conditions:

$$E(\epsilon|X) = 0$$
$$V(\epsilon|X) = \sigma^2 I$$

The most widely used estimator of regression coefficients from the above model is known as the "ordinary least squares" (OLS) estimator, usually denoted $\hat{\beta}$ is provided below for reference.

$$\hat{\beta} = (X^T X)^{-1} X^T y \tag{1}$$

From the above, note the following two very important quantities. The fitted values, which we denoted as $\hat{y} = X\hat{\beta}$ and the vector of residuals, $\hat{\epsilon} = y - \hat{y}$.

5.1 Classical Linear Model

We will begin our journey by looking at a model which will analyse subscriptions to major academic journals in Economics. In particular, we will look at the relationship between the number of subscriptions to Economics journals at US libraries as a function of subscription prices. To begin, we must first download the data set. It is found in the package "AER". We must first download it and then load it.

```
> install.packages("AER")
> library(AER)
```

Now that we have loaded the package, let us load the data set "Journals".

```
> data(Journals)
```

Let us now briefly look at the data set to see what it contains. To see the variables contained in the data frame, we issue the following command:

```
> names(Journals)
```

We can get more insight issuing the `summary()` command. Consider:

```
> summary(Journals)
```

Because we are interested in looking at the subscriptions as a function of price, let us therefore create a data set which will extract these two variables.

```
> journals = Journals[, c("subs", "price")]
```

Moreover, to get a sense for the price per quality of a journal, we will create such a measure by looking at subscription price per citations of the journal. Here, "citations" is the total number of times the journal is cited in academic works. Clearly then, the number of citations is an indicator of the quality of the journal. Consequently, a lower price / citation ratio indicates that one is paying less for higher quality journals. Thus create the following variable:

```
> journals$citeprice = Journals$price / Journals$citations
```

Now, before proceeding any further, it is time to see first hand why studying your data BEFORE doing any major analysis is absolutely essential! Start by plotting "citeprice" vs. "subs".

```
> plot(journals$citeprice,journals$subs)
```

It is difficult to see what the relationship looks like. However, a little reflection will show that the pattern is really a decaying exponential curve. To see this, consider:

```
> par(new=TRUE)
> curve(exp(-x), from = 0, to = 50)
```

What this implies is that if we had naively run a linear regression model on "subs" on "citeprice", we would have severely misspecified the model as $exp(-x)$ is clearly far from linear. However, given that we know what curve the data follows, it is an easy matter to linearize the model by transforming the data using the logarithmic function. Thus, consider the transformed variables:

$$\begin{aligned} y &= e^{-x} \\ log(y) &= e^{-log(x)} \\ &= -x \end{aligned}$$

It is clear then that the the two variables which do yield a linear relationship are $log(y)$ and $log(x)$. To see this then, consider the plot of the transformed variables.

```
> plot(log(journals$citeprice), log(journals$subs))
```

Clearly the data points look far more linear after the transformation. Given this new insight, let us proceed to fit a linear regression model of the form:

$$log(subs)_t = \beta_0 + \beta_1 log(citeprice)_t$$

The main function which deals with regressions and model fitting is the `lm()` function, which stands for "linear model". We start by first attaching the data set "journals" so that we can call variables up without referring to their data.frame.

```
> attach(journals)
```

Next, let us fit the linear regression in question as follows:

```
> journals.lm = lm(log(subs)~log(citeprice))
```

Let us now add the regression line to our existing plot using the `abline()` function. The latter function extracting the coefficients from the estimated model and superimposing the corresponding regression line in one shot. Consider:

```
> abline(journals.lm)
```

Here, it should be noted that, although we have only specified one regressor, namely "citeprice", it is implied that there is a constant term involved. It is important to keep this in mind whenever performing linear regressions in R.

The element "journals.lm" is very important here and contains much useful information. To see what is located in it, let us do some snooping.

```
> names(journals.lm)
> summary(journals.lm)
```

This last input is very important. It yields the essential information from the regression. That is, the coefficient estimates, the t statistics, the p-values, and some other very important information. Two of these are particularly important. The first is the interpretation of the p-values presented. Here, the p-value corresponds to the null hypothesis that the value of the coefficient is equal to zero. In our particular case, the fact that p-values are very close to zero means that we reject the null hypothesis at higher significance levels. That is, whenever the p-value is less than the level we specify, we reject the null hypothesis. Because we do reject the null hypothesis in our case, we say that the coefficient estimates are significant. That is, they are significantly different from zero!

The second important aspect we need to analyse here are the standard errors of coefficient estimates. These are particularly important when we desire to obtain confidence intervals of the OLS estimator. The latter are obtained by noting the following relationship:

$$Pr\{t_{\alpha/2} \leq \frac{\hat{\beta} - \beta}{\hat{\sigma}_{\hat{\beta}}} \leq t_{1-\alpha/2}\} = 1 - \alpha \tag{2}$$

We note here that the fraction in the middle of (2) above is distributed as a t-distribution with $(n - \#regressors)$ degrees of freedom, where n is the sample size. As a result, $t_{\alpha/2}$ and $t_{1-\alpha/2}$ are the $\alpha/2$ and $1 - \alpha/2$ quantiles of the t distribution, and alpha is the level corresponding to the significance level. In this light, we note that the standard errors required to compute the above are indeed given in the summary of the model we just estimated, and are the square roots of the diagonal elements of the covariance matrix of $\hat{\beta}$. The latter is, recall, given by the following:

$$V\{\hat{\beta}\} = \hat{\sigma}^2_{\hat{\epsilon}}(X^T X)^{-1}$$

We can indeed compute the confidence interval manually by manipulating the matrix of regressors, the vector of residuals, and the vector of coefficients, the latter two quantities being extracted from our estimate model as follows:

```
> resid(journals.lm)
> coef(journals.lm)
```

Or, R can do the dirty work for us, and we can obtain the confidence interval of the OLS coefficients automatically by issuing the following command:

```
> confint(journals.lm, level = 91.2345%)
```

Here the level is any value we wish and corresponds to α in the theoretical derivation of the confidence interval we derived above.

As long as we are on the subject of confidence intervals, let us also discuss prediction. That is, now that we have our model estimated, we can actually put it to good use and do some prediction with it. However, before doing so, we must make sure that we distinguish between "mean prediction" and "individual prediction".

Mean prediction is the following derivation given the basic assumptions of the classical linear model.

$$\begin{aligned} y^*_{mean} = E\{y|X\} &= X\hat{\beta} + E\{\hat{\epsilon}\} \\ &= X\hat{\beta} \end{aligned}$$

Individual prediction however is the following statement

$$y^*_{individual} = X\hat{\beta} + \hat{\epsilon}$$

The difference between the two can be shown to be the variance of the predicted values. The former has a smaller variance than the latter and hence, the former yields a narrower prediction band than the latter. To see this, consider

$$\begin{aligned} V\{y^*_{mean}\} &= V\{X\hat{\beta}\} \\ &= V\{Py\} \end{aligned}$$

The above holds true because the fitted values are Py. Analogously, we have the following:

$$\begin{aligned} V\{y^*_{individual}\} &= V\{X\hat{\beta}\} + V\{X\hat{\epsilon}\} \\ &= V\{Py\} + V\{My\} \end{aligned}$$

Again, the above holds because the residuals are My and the fact that the fitted values and residuals are uncorrelated. Moreover, the fact that the variances of the fitted values and residuals are positive semidefinite, it holds that the variances of the data points predicted using the individual method have wider confidence bands than those of the points predicted using the mean method.

Now that we have this distinction, we can generate mean predicted values for y by first specifying the vector of x values we wish to correspond to our prediction. For example, if we wanted to predict y for the 61 values of x ranging from -6 to 6 by increments of 0.2. Note that this example gives the prediction for y for values of x which are NOT in the original set of data points. In fact, it extends the data points beyond the minimum and maximum value of "log(citeprice)". All in all, we can do this as follows:

```
> x = seq(-6,6,.2)
> predc = predict(journals.lm, data.frame(citeprice = exp(x)), interval
= "confidence")
```

On the other hand, if we wanted the wider prediction corresponding to the individual prediction method, we would issue the following command:

```
> predp = predict(journals.lm, data.frame(citeprice = exp(x)), interval
= "prediction")
```

A very important thing to note here is the "data.frame()" argument used above. Note that we have used "citeprice" and NOT $log(citeprice)$. This is because the variable of interest is "citeprice", even though when we ran our regression, we had used the transformed variable, $log(citeprice)$. Thus, R is aware that we had used a transformation. As a result, when we specify our "data.frame()" argument, we specify it in terms of the base variable of interest "citeprice". However, note that $citeprice = exp(x)$ really implies that $log(citeprice) = x$ and so our scales are not different.

Note also that if we issue the command to view the above variables, we would note that they have three columns. The first, called "fit", is the OLS fitted value which we know as Py, corresponding to the values of x we chose for prediction. The other two columns are "lwr" and "upr", corresponding to the lower and upper confidence interval values for our prediction, respectively.

```
> predc
> predp
```

Finally, to see all this in graphical form, where it is easier to visualize the impact of prediction, consider issuing the following commands:

```
> plot(log(citeprice),log(subs), xlim=c(-6,6))
> abline(journals.lm)
> lines(x, predc[,2], col = 2, lty = 2)
> lines(x, predc[,3], col = 2, lty = 2)
> lines(x, predp[,2], col = 4, lty = 2)
> lines(x, predp[,3], col = 4, lty = 2)
```