

# A GENTLE INTRODUCTION TO R\*

CHRISTOPHER F. PARMETER

## 1. R YOU READY FOR R?

**1.1. What is R?.** R is an implementation of the object-oriented mathematical programming language S. The S programming environment (S stands for statistics) was developed by AT&T as the single letter competitor to C (also developed by AT&T). R is developed by statisticians around the world and is free software, covered by the GNU General Public License. Syntactically and functionally it is very similar (if not identical) to S+, the popular statistics package. R is capable of running high powered statistical simulations, producing elegant graphics and computing econometric estimates for a wide array of popular estimators. R also has the ability to seamlessly integrate with  $\text{\LaTeX}$  to construct fully reproducible scientific research.

**1.2. How Can I get R?.** R is open source software enabling anyone with an internet connection to instantly download it to your computer and begin working. The easiest place to find R is through its host website The R Project, see Figure 1. Once you have navigated to The R Project webpage seek out the CRAN link on the left hand side. The CRAN tab will then ask you to select a mirror which hosts R, see Figure 2. For this introduction I have elected to use the CRAN link through the University of California, Berkeley. R is supported on Linux, Mac OS/X and Windows operating systems. Depending upon your operating system you will need to select the appropriate link to begin the process of installing R on your computer see Figure 3. Figures 4 and 5 show this for a Mac OS/X operating system where I have selected the link for R-3.0.1.pkg.

Once you have the R binaries downloaded onto your computer you will want to install it onto your computer. Once your have R installed on your machine its time to let R rip!

**1.3. Why Use R?.** There are a multitude of reasons for using R (see Racine & Hyndman 2002). I will only list several of the advantageous features that make R so appealing to applied economists and statisticians alike.

- Comparable in power to commercial products,
- Effective data handling and storage facility,
- Large collection of integrated intermediate tools for data analysis,

---

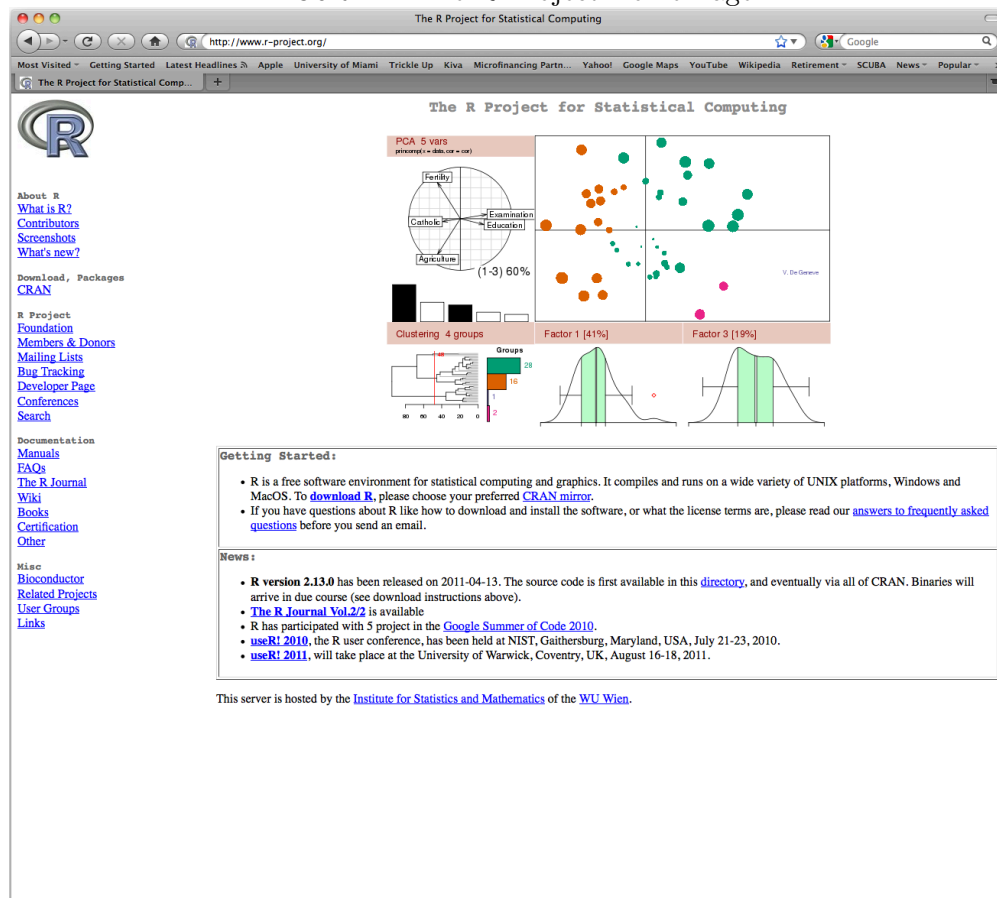
UNIVERSITY OF MIAMI

*Date:* August 8, 2013.

Christopher F. Parmeter, Department of Economics, University of Miami; e-mail: cparmeter@bus.miami.edu.

\*These notes have been prepared for ‘Applied Panel Data Econometrics’ in Dakar, Senegal, September 9-13, 2013.

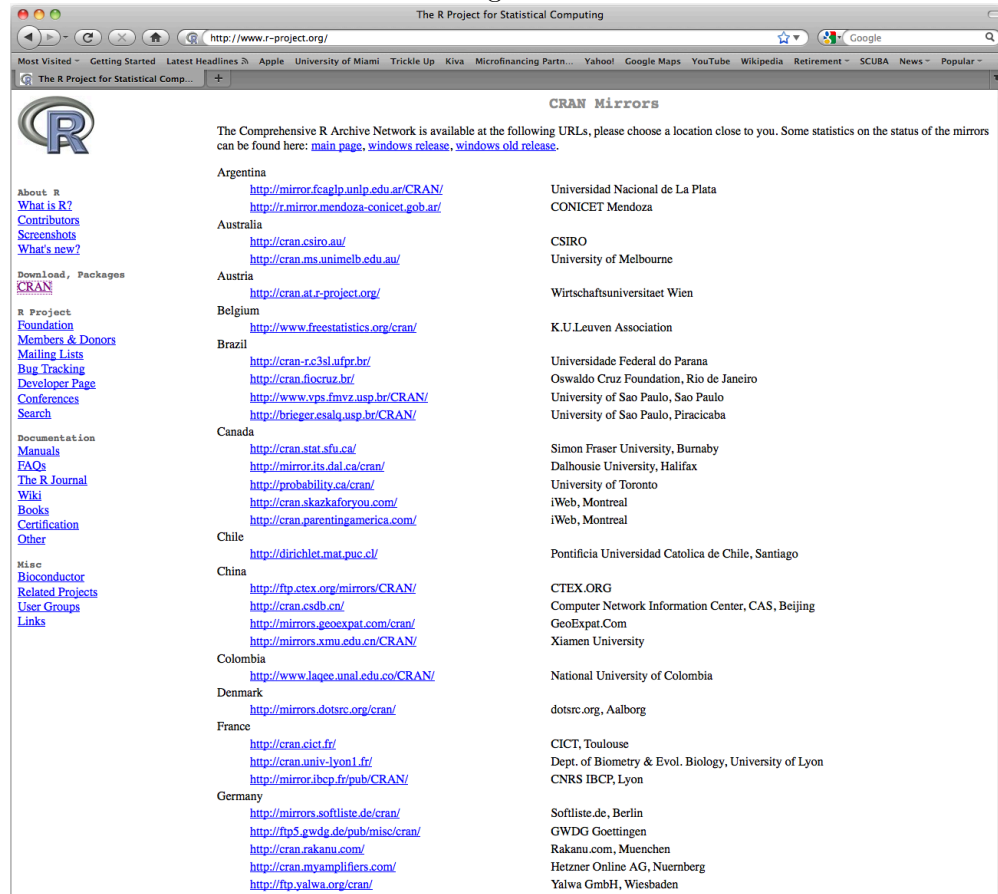
FIGURE 1. The R Project Home Page



- General object-oriented programming language, allowing you to construct your own statistical functions,
- R is polymorphic – same function can be applied to different objects with tailored results,
- Links with Sweave to produce fully replicable research,
- Its Free,
- Its Free!

Don't be turned off by the fact that R is an object-oriented language. In this instance this is wonderful for the analyst since you have the ability to implement an estimator or test which currently does not exist using base calls within R. An appeal of using any object-oriented programming language is that programming is much easier for the user. R is polymorphic. This means that the same function called within an R session can be used on different objects and different results are calculated based on the type of object passed to the function. A further benefit of constructing objects is that your thought process for instructing R what to do is enhanced. When you run a regression in SAS, for example, a litany of results is spewed onto the screen and it can be confusing to find what you need. With an object-oriented language the regression call returns an object

FIGURE 2. Selecting the Mirror on CRAN



(meaning nothing is actually printed on your screen) and this objects holds the results of interest, estimates, standard errors,  $R^2$ ,  $p$ -values, etc. The user then decides which pieces of the object to use.

FIGURE 3. The Initial Download Page

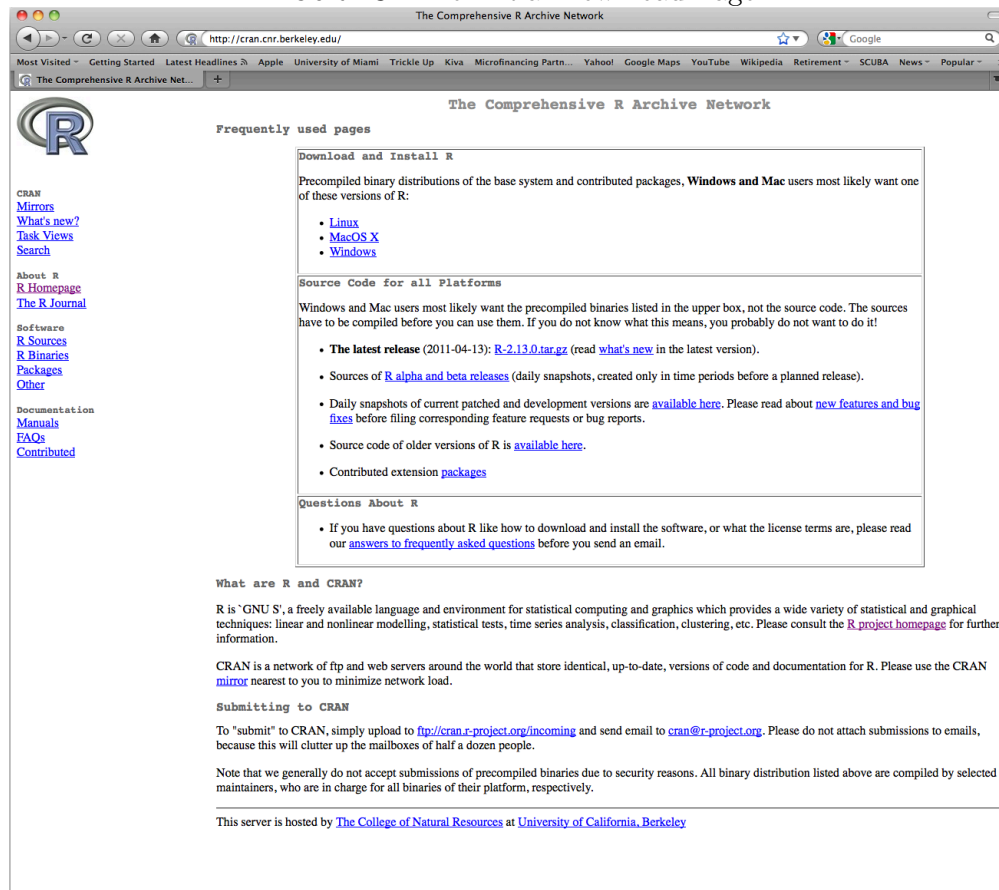
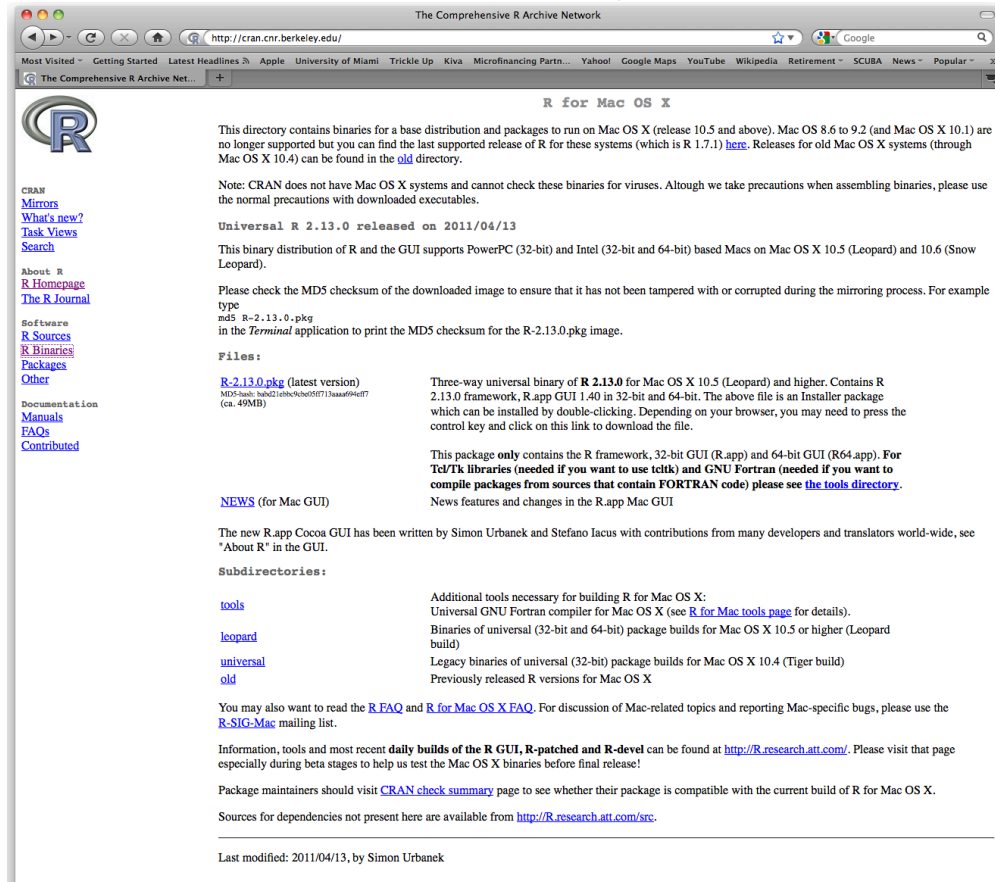


FIGURE 4. Downloading for a Mac OS/X Operating System



The screenshot shows a web browser window displaying the CRAN website for Mac OS X. The browser's address bar shows the URL <http://cran.cnr.berkeley.edu/>. The page title is "R for Mac OS X". The left sidebar contains a navigation menu with links: CRAN, Mirrors, What's new?, Task Views, Search, About R, R Homepage, The R Journal, Software, R Sources, R Binaries, Packages, Other, Documentation, Manuals, FAQs, and Contributed. The main content area has the heading "R for Mac OS X" and contains the following text:

This directory contains binaries for a base distribution and packages to run on Mac OS X (release 10.5 and above). Mac OS 8.6 to 9.2 (and Mac OS X 10.1) are no longer supported but you can find the last supported release of R for these systems (which is R 1.7.1) [here](#). Releases for old Mac OS X systems (through Mac OS X 10.4) can be found in the [old](#) directory.

Note: CRAN does not have Mac OS X systems and cannot check these binaries for viruses. Although we take precautions when assembling binaries, please use the normal precautions with downloaded executables.

Universal R 2.13.0 released on 2011/04/13

This binary distribution of R and the GUI supports PowerPC (32-bit) and Intel (32-bit and 64-bit) based Macs on Mac OS X 10.5 (Leopard) and 10.6 (Snow Leopard).

Please check the MD5 checksum of the downloaded image to ensure that it has not been tampered with or corrupted during the mirroring process. For example type

```
md5 R-2.13.0.pkg
```

in the *Terminal* application to print the MD5 checksum for the R-2.13.0.pkg image.

**Files:**

[R-2.13.0.pkg](#) (latest version)  
 MD5=8a6c 7a6c21e1b0c9b0e01713aaad99d077  
 (ca. 49MB)

Three-way universal binary of R 2.13.0 for Mac OS X 10.5 (Leopard) and higher. Contains R 2.13.0 framework, R.app GUI 1.40 in 32-bit and 64-bit. The above file is an Installer package which can be installed by double-clicking. Depending on your browser, you may need to press the control key and click on this link to download the file.

This package **only** contains the R framework, 32-bit GUI (R.app) and 64-bit GUI (R64.app). For Tcl/Tk libraries (needed if you want to use tcltk) and GNU Fortran (needed if you want to compile packages from sources that contain FORTRAN code) please see [the tools directory](#).

[NEWS](#) (for Mac GUI)  
 News features and changes in the R.app Mac GUI

The new R.app Cocoa GUI has been written by Simon Urbanek and Stefano Iacus with contributions from many developers and translators world-wide, see "About R" in the GUI.

**Subdirectories:**

[tools](#)  
 Additional tools necessary for building R for Mac OS X:  
 Universal GNU Fortran compiler for Mac OS X (see [R for Mac tools page](#) for details).

[leopard](#)  
 Binaries of universal (32-bit and 64-bit) package builds for Mac OS X 10.5 or higher (Leopard build)

[universal](#)  
 Legacy binaries of universal (32-bit) package builds for Mac OS X 10.4 (Tiger build)

[old](#)  
 Previously released R versions for Mac OS X

You may also want to read the [R FAQ](#) and [R for Mac OS X FAQ](#). For discussion of Mac-related topics and reporting Mac-specific bugs, please use the [R-SIG-Mac](#) mailing list.

Information, tools and most recent **daily builds of the R GUI, R-patched and R-devel** can be found at <http://R.research.att.com/>. Please visit that page especially during beta stages to help us test the Mac OS X binaries before final release!

Package maintainers should visit [CRAN check summary](#) page to see whether their package is compatible with the current build of R for Mac OS X.

Sources for dependencies not present here are available from <http://R.research.att.com/src>.

Last modified: 2011/04/13, by Simon Urbanek

FIGURE 5. Starting the Download

Please check the MD5 checksum of the downloaded image to ensure that it has not been tampered with or corrupted during the mirroring process. For example, type

```
md5 R-2.13.0.pkg
```

in the *Terminal* application to print the MD5 checksum for the R-2.13.0.pkg image.

## Files:

[R-2.13.0.pkg](#) (latest version)  
MD5-haah:haah21ebhc9cb050713aaad09ad77  
(ca. 49MB)

Three-way universal binary of **R 2.13.0** for Mac OS X 10.5 (Leopard) and higher. Contains R 2.13.0 framework, R.app GUI 1.40 in 32-bit and 64-bit. The above file is an installer package

Opening R-2.13.0.pkg

You have chosen to open

📦 R-2.13.0.pkg

which is a: [Binary File](#)

from: <http://cran.cnr.berkeley.edu>

Would you like to save this file?

Cancel

Save File

[NEWS](#) (for Mac GUI)

The new R.app Cocoa  
"About R" in the GUI.

## Subdirectories:

[tools](#)

[leopard](#)

[universal](#)

[old](#)

Additional tools necessary for building R for Mac OS X:

Universal GNU Fortran compiler for Mac OS X (see [R for Mac tools page](#) for details).

Binaries of universal (32-bit and 64-bit) package builds for Mac OS X 10.5 or higher (Leopard build)

Legacy binaries of universal (32-bit) package builds for Mac OS X 10.4 (Tiger build)

Previously released R versions for Mac OS X

You may also want to read the [R FAQ](#) and [R for Mac OS X FAQ](#). For discussion of Mac-related topics and reporting Mac-specific bugs, please use the [R-SIG-Mac](#) mailing list.

Information, tools and most recent **daily builds of the R GUI, R-patched and R-devel** can be found at <http://R.research.att.com/>. Please visit that page especially during beta stages to help us test the Mac OS X binaries before final release!

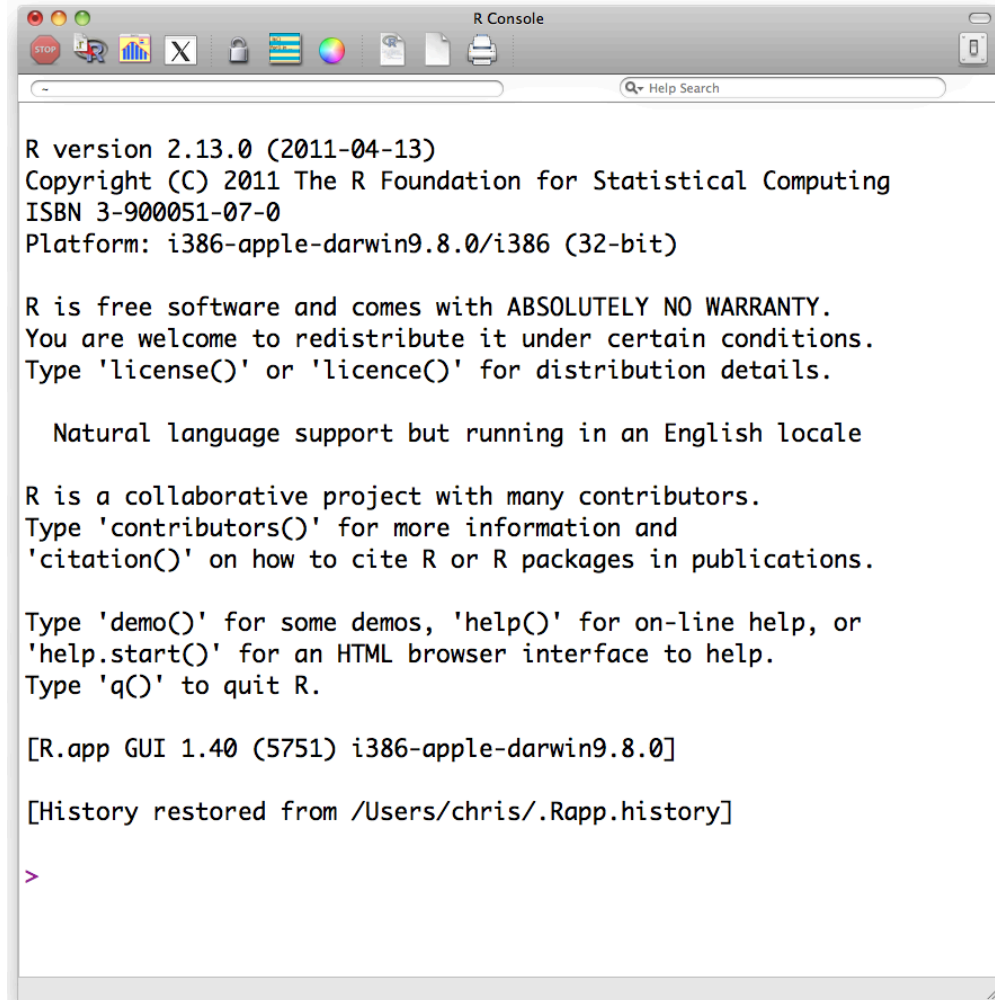
## 2. GETTING UP AND RUNNING

**2.1. Just Getting R Started.** Once R is installed successfully on your machine you will want to click on the R icon to launch the GUI (Figure 6). This will launch the R console which is shown in Figure 7.

FIGURE 6. The R Icon

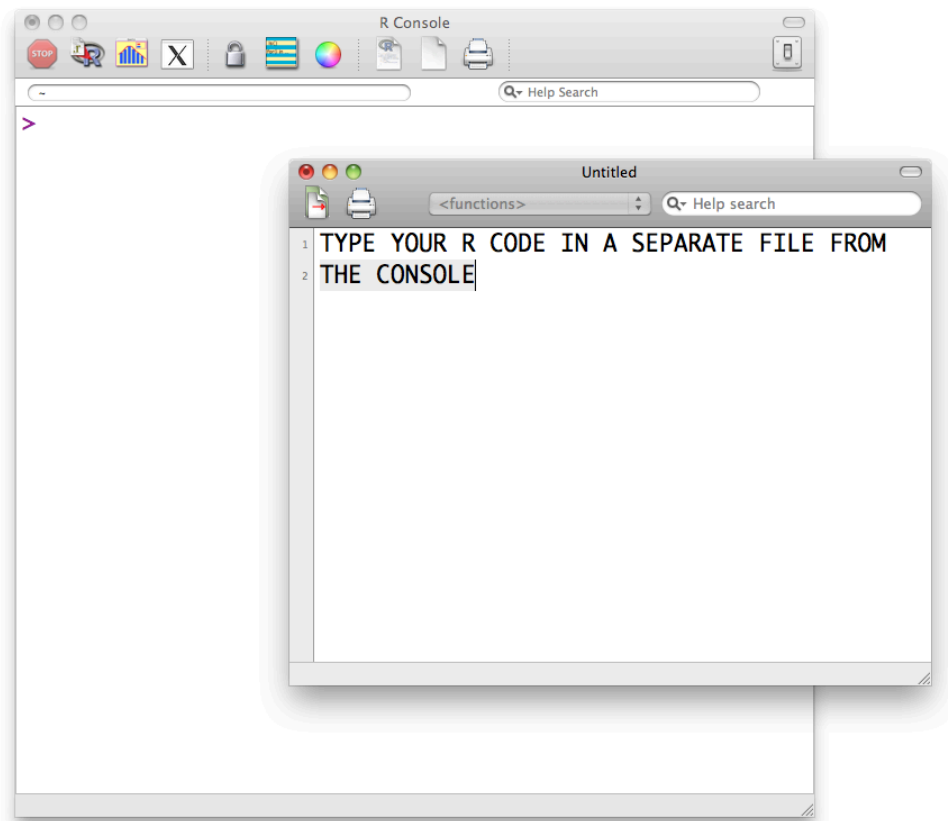


FIGURE 7. The R Console



**2.2. An R Session.** Having installed and run R you will find yourself at the `>` prompt. Prior to discussing the crucial details of running a statistical analysis in R lets focus on some necessary heuristics. There are two main platforms to run R code in an R session. First you can simply type commands at the command prompt. Alternatively, you can open a separate R document, a `.R` file, write all of you code and comments in there (similar to a `.do` file in Stata), and then compile this document in the R console. I would recommend always saving your R code in a separate document as opposed to running code directly in the R console.

FIGURE 8. A separate file to write all your R code.



From now on I will be describing operating R from the GUI and will use ‘Console’ to describe typing in the main R console that opens up when you launch R and a separate ‘R file’ which you can write your own code to and save.

**2.3. Packages.** To conduct statistical analysis one will need to make use of ‘packages’. R’s functionality is entirely based on the concept of packages and each package is a collection of functions which are designed to carry out specific tasks, much the same way that toolboxes operate in Matlab or modules in Gauss. Your installation of R resulted in the `base` package being installed. While



there are numerous functions which one can evoke within the `base` package, additional packages are needed to perform a variety of econometric operations, for example, to estimate panel data models we will want to use the `plm` package. These additional packages are commonly referred to as ‘recommended’ or contributed packages since members of the R community have graciously done the dirty work to construct functions that we desire to deploy.

Installing packages in R is easy and the R project homepage allows you to connect to the ever growing collection of contributed packages. Currently there are over 4738 contributed packages available. While it may seem a daunting task to parse through the numerous packages available to find just the right set of functions, R offers two accessible avenues to find packages most relevant to your empirical work. First, groups of thematic packages have been collected into ‘Task Views’ amongst them the `Econometrics` Task View which will install more than 50 packages which perform a range of econometric methods. In an interesting pedagogical twist, to install any of the ‘Task Views’ one needs to install the `ctv` package. This can be installed directly from the R console with the following code:

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

Once the `ctv` package has been installed in must be enabled in the current R session to load ‘Task Views’. This can be done with the `library()` command. If this is your first time installing R, or if you have recently updated R, the following views will load a majority of the packages you are likely to require for your empirical work. Installing these packages will take a few minutes.

```
> library("ctv")
> install.views("Econometrics")
> install.views("Cluster")
> install.views("Distributions")
> install.views("Graphics")
> install.views("HighPerformanceComputing")
> install.views("Optimization")
> install.views("Robust")
> install.views("ReproducibleResearch")
> install.views("TimeSeries")
> install.views("Survival")
> install.views("SocialSciences")
> install.views("Spatial")
```

Anytime you update R or install it on another computer you will need to reload all of the contributed packages which you had previously installed. After installing these packages, or views, you can update them at any point by typing

```
> update.packages()
> update.views("Econometrics")
```

If you desire to load a package not linked with one of the views you can do this with a simple call. For example, if I wanted to load the `plm` package which contains commands for a range of estimation and inference across myriad panel models I would type

```
> install.packages(plm)
```

## 2.4. Getting Help in R.

2.4.1. *From the Command Prompt.* At some point you will most likely want to look up how to perform a specific operation, or you will need help with a command within one of R's packages. Seeking help in R is very easy and there are a variety of ways to ask for help. If you know the subject you would like help on you can use the `??` call which searches for the term following the double question marks over all the packages you currently have installed in R. For example, if you wanted to know all the available commands related to 'regression' you could type

```
> ??regression
```

Alternatively, after having searched for a subject you found a command for, such as `lm()`, you could open the help file for this command by calling

```
> ?lm
```

Notice that you use `??` when you are searching for help on a topic, but you use `?` to open a help file (if one has been created) for a specific command within R. Rather than use the `??` and `?` calls you may also evoke `help.start()` or `help.search("foo")` where `foo` is a string, such as 'regression' or 'variance'.

To search for help in a web browser you may use the `help.start()` command

```
> help.start()
```

The call to `help.start()` spawns your web browser where you can interactively search for help on any topic you desire.

2.4.2. *Web sites.* A number of sites are devoted to helping R users, and we briefly mention a few of them below.

**<http://www.R-project.org/>:** This is the R home page from which you can download the program itself and many R packages. There are also manuals, other links, and facilities for joining various R mailing lists.

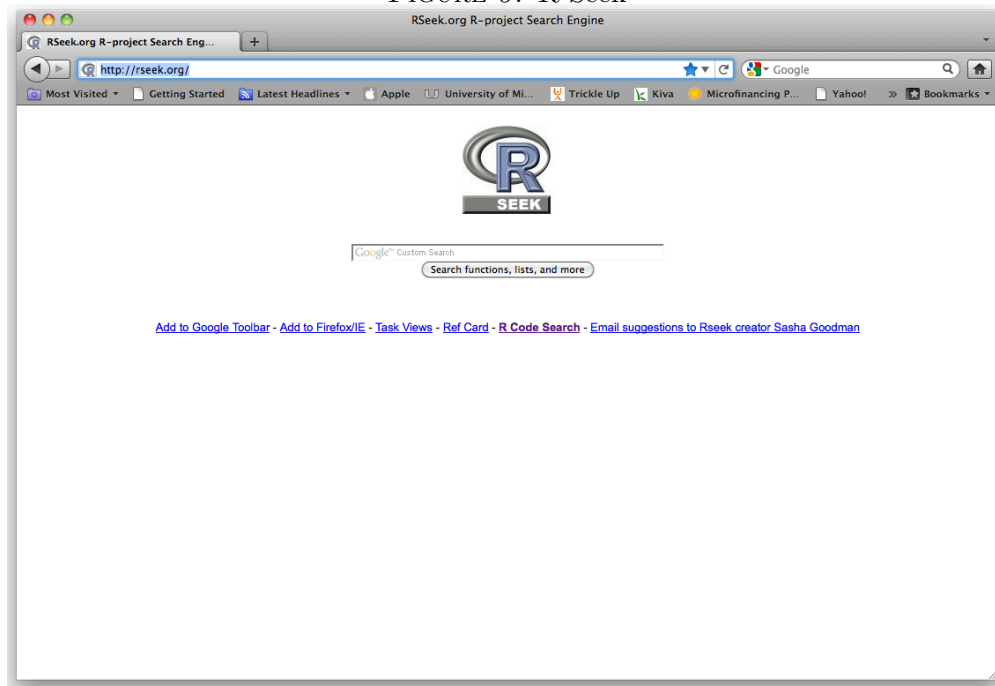
**<http://CRAN.R-project.org/>:** This is the 'Comprehensive R Archive Network,' "a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for the R statistical package." Packages are only put on CRAN when they pass a rather stringent collection of quality assurance checks, and in particular are guaranteed to build and run on standard platforms.

**<http://cran.r-project.org/web/views/Econometrics.html>:** This is the CRAN 'task view' for computational econometrics. "Base R ships with a lot of functionality useful for computational econometrics, in particular in the stats package. This functionality is complemented

by many packages on CRAN, a brief overview is given below.” This provides an excellent summary of both parametric and nonparametric packages that exist for the R environment. **<http://pj.freefaculty.org/R/Rtips.html>**: This site provides a large and excellent collection of R tips.

2.4.3. *The R Search Engine.* An additional resource available to the R community is R Seek (Figure 9). Not only does R Seek allow you to search for help on commands and packages, but it also searches the web for R code so that you can (potentially) track down code that has as yet to be placed on CRAN.

FIGURE 9. R Seek



2.5. **Importing data.** R allows users to import a variety of different data sources. Within the **base** package users can import ASCII/text file and comma separated files directly using the commands **read.table** and **read.csv**, respectively. However, not all data files come in these forms and so for other data types the **foreign** package allows you to read data created by different popular programs. To load it, simply type **library(foreign)** from within R. Supported formats include

**read.arff:** Read Data from ARFF Files

**read.dbf:** Read a DBF File

**read.dta:** Read Stata Binary Files

**read.epiinfo:** Read Epi Info Data Files

**read.mtp:** Read a Minitab Portable Worksheet

**read.octave:** Read Octave Text Data Files

**read.S:** Read an S3 Binary or data.dump File

**read.spss:** Read an SPSS Data File

**read.ssd:** Obtain a Data Frame from a SAS Permanent Dataset, via read.xport

**read.systat:** Obtain a Data Frame from a Systat File

**read.xport:** Read a SAS XPORT Format Library

Moreover, the `xlsx` package allows you read the contents of an excel spreadsheet into an R dataframe. Again to load this package simply type `library(xlsx)` from within R. To read in data use the call `read.xlsx()`.

The following code snippet would read a Stata file named ‘growth.dta’ and lists the names of variables in the data frame. Further, the `head()` command will print the first six rows of the dataset.

```
> library(foreign)
> mydat <- read.dta(file="growth.dta")
> names(mydat)
> head(mydat)
```

Clearly R makes it simple to migrate data from one environment to another.

**2.6. Setting the Working Directory.** Typically you will want R to recognize a specified location where it should look for data, source code and to store output. This is known as the working directory. The stock installation of R on your machine has given it a standard working directory which you can determine by simply typing

```
> getwd()
```

If you were to attempt to import data prior to setting your working directory then R would return an error saying that it cannot locate the file you asked for. Setting the working directory in R is very easy. In the Console you can manually set it via

```
> setwd(workingdirectorylocation)
```

where `workingdirectorylocation` is the path to your working directory. Additionally, if you were to open an R file directly (instead of from the R Console) then R will set the working directory to the location of this file. Moreover, you can always include as the first line of code in an R file `setwd(location)`.

Aside from using the `setwd()` command within the R Console you can also use the drop down menus available within the GUI to set your working directory.

## 2.7. Some Coding Preliminaries.

**2.7.1. Naming.** R is an expression language, and as with most UNIX based packages it is case sensitive. This means that the symbols `B` and `b` refer to different things. All alphanumeric symbols are allowed within the R Console along with ‘.’ and ‘\_’. For portable R code one should only use the symbols A-Z, a-z and 0-9 to name things. R’s naming restrictions require that a variable’s

name begin with either a letter (upper or lowercase is acceptable) or ‘.’. However, if you name a variable beginning with the character ‘.’ the next character cannot be a digit.

**2.7.2. Elementary Code.** Many commands are either of the form of expressions or assignments. An expression is a command that is evaluated, printed and the value is subsequently not stored in memory. An expression would be of the form

```
> 2+2
```

```
[1] 4
```

An assignment uses the `<-` command and like an expression it is a command that is evaluated. However, the value is not printed to the screen but stored to the variable which it is assigned to. An assignment would be of the form

```
> b <- 2+2
```

When writing code it is common to separate commands. This can be done in two ways in R. First, R views a newline (Enter or Return) as the beginning of a new command. Additionally, the semi-colon (;) can be used to separate commands that are written on the same line. In my own preference I prefer never to write two commands on the same line and so I always have the newline as my form of separating commands.

```
> ## Newline separating commands
```

```
> 2+2
```

```
[1] 4
```

```
> 3+3
```

```
[1] 6
```

```
> ## Semi-colon separating commands
```

```
> 2+2; 3+3
```

```
[1] 4
```

```
[1] 6
```

**2.7.3. Commenting.** To comment your code, which I highly recommend so that others can understand what you have done and so that when you get revisions you can remember what you did, use the `#` symbol. I like to comment as often as possible without adding unnecessary length or clutter to my code. In principle there are two main ways to comment your code, I call them ‘Comment and Code’ and ‘Parallel Commenting’. Examples of each style are given here

```
> ## Comment and Code
```

```
> ## Here I demonstrate Addition and Subtraction in R
```

```
> ## Addition
```

```
> 2+2
```

```
> ## Subtraction
```

```
> 3-2
```

```

> ## Parallel Commenting
> ## Here I demonstrate Addition and Subtraction in R
> 2+2      ## Addition
> 3-2      ## Subtraction

```

Either form is acceptable and you should go with whichever fits your preferences, but always comment regardless.

Sometimes when coding in the Console one will make simple mistakes that upon attempting to run the code (typically by pressing enter) will result in the next line showing a '+' symbol. This symbol signifies that the previous call was not wrapped up properly. Typical reasons for this are not including enough delimiters, such as ] or ), or forgetting to close off parentheses. For debugging code it is common practice to write code over numerous lines using indentation to signify a single command. This makes it easier on the eyes. When making calls to commands with numerous controls and inputs, it is common to begin new lines using a comma ',' which R will recognize as being part of the same command.

*2.7.4. Command History.* If you are coding in the Console, it is common that you will make mistakes and have to rerun code that has some error within it. Rather than type the entire command again, simply pressing the 'up arrow' on your key board will reproduce the last call in the command history. Repeatedly pressing the up arrow will scroll through the set of commands loaded within the command history. Once you have scrolled through the command history you can easily scroll back down by using the 'down arrow' on your keyboard.

*2.7.5. Executing Commands From and Diverting Output to Files.* As mentioned earlier, typically you will want to construct R code in a separate R file. To then run these codes at any point within an R session one simply uses the `source()` command. If I had a set of commands named `PanelData.R` then to run this in my R session (making sure this file is in the working directory) I would use the command

```

> source("PanelData.R")      ## Run PanelData.R

```

For an R session on a Windows platform there is also a drop down menu with the 'Source' command that will allow you to select the R file you wish to run without it being located in your working directory. On an Apple platform one can source code using the Command button while pressing the E key.

To send output to a specific file use the `sink()` command. Keep in mind that once `sink()` has been called no output will be produced on the screen until the sink has been turned off. A typical incantation of this would be to direct output to a file called `PanelData.out` and then to turn the sink off using

```

> sink("PanelData.out")      ## Direct output to PanelData.out
> 2+2
> sink()                     ## Stop directing output

```

2.7.6. *Objects in Memory.* All entities which you create in your R workspace are known as objects. Your objects may be character strings, vectors, arrays, lists, matrices, functions or more general structures. In your R session you can list the objects known in the workspace by typing

```
> objects()      ## List objects in R session
> ls()           ## Alternative command to list objects
```

You can unilaterally remove an object from the R workspace using the `rm()` command. Here is an example

```
> b <- 2+2      ## Create an object
> objects()     ## List objects in R session
[1] "b"
> rm(b)         ## Remove b from workspace
> b             ## Display error so that you see b was removed from memory
[1] "Error in try(b) : object 'b' not found\n"
```

If you wanted to remove all objects from your workspace you can simple type

```
> rm(list=ls())
```

When you finish your R session you will be asked if you would like to save all currently available objects. If you elect to save your current session then the objects are written to a file with the extension `‘.RData’` within the working directory while the command lines are written to a file with the extension `‘.Rhistory’` in the working directory. The next time you open an R session from this directory the workspace is reloaded with these objects and command lines. Be careful where you open your R session as it is common to have variables with the same name within different directories making it easy to overwrite things if you are not cautious. Additionally, you can eschew saving your R workspace if your codes are quick to compute or you have an easily manageable dataset. Or, if you are concerned that you may have objects with the same name loaded from memory you can begin you code with `rm(list=ls())`.

It is also common to construct functions and variables whose names conflict with one another. A simple mechanism to check for conflicts is the `conflicts()` command. This command will produce a set of named conflicts so that you can determine if you have objects which might conflict with R’s core operating environment. If you set `detail=TRUE` inside of your call to `conflicts()` then the objects where the conflicts arise are also printed out.

```
> T <- "time"      ## Create a known conflict
> conflicts()      ## Check for conflicts
[1] "body<-" "kronecker" "T"
> rm(T)            ## Eliminate conflicts
> conflicts(detail=TRUE) ## Check for conflicts
$`package:methods`
[1] "body<-" "kronecker"
```

```
$`package:base`
[1] "body<-"      "kronecker"
```

While `objects()` and `ls()` list object names in memory, if you would like more details of the things R is currently using you can invoke the `ls.str()` command. Also, `list.files()` will list all files in your current working directory.

### 3. SIMPLE DATA MANIPULATION

As with most matrix language software programs R allows you to create a variety of storage objects including numbers, vectors, matrices, strings and dataframes (more on dataframes later!). To assign a number to a variable we simply use the assignment operator

```
> a <- 7.2  ## Assignment of a scalar
```

which creates a numeric object named `a` with the value 7.2. Often we will be more interested in vectors and matrices as opposed to scalars. To create a vector in R we use the `c()` command (which stands for concatenation) which will create a column vector

```
> b <- c(a,4.3,2.4)      ## Assignment of a column vector
```

If we wanted a row vector instead we could transpose `b` using `t()` as follows

```
> B <- t(b)      ## Transposition
```

Given R's case sensitivity `b` and `B` are distinct. We can concatenate vectors as well as scalars. For example to concatenate by columns we can use `cbind()` while to concatenate by rows we can use `rbind`.

```
> BB <- rbind(B,B)      ## Row concatenation
> BB
```

```
      [,1] [,2] [,3]
[1,]  7.2  4.3  2.4
[2,]  7.2  4.3  2.4
```

```
> bb <- cbind(b,c(1,2,3))  ## Column concatenation
> bb
```

```
      b
[1,] 7.2 1
[2,] 4.3 2
[3,] 2.4 3
```

Elements of vectors and matrices are indexed via square brackets `[]`. Several examples are

```
> b[2]      ## 2nd element of b
```

```
[1] 4.3
```

```
> B[3]      ## 3rd element of B
```

```
[1] 2.4
```



```

> bb[2,]          ## 2nd row of bb
  b
4.3 2.0
> bb[,1]          ## 1st column of bb
[1] 7.2 4.3 2.4
> BB[2,3]         ## Cell 2,3 of BB
[1] 2.4
> bb[1:2,1:2]    ## Cells (1,1), (1,2), (2,1) and (2,2)
      b
[1,] 7.2 1
[2,] 4.3 2

```

The blank that accompanies the comma inside the square brackets signifies to R to use either the whole column or row. The colon ':' can be used when consecutive rows/columns are desired to pull out of a matrix. One could also pull out non-consecutive rows/columns.

```

> bb[c(1,3),2]    ## Pull out 1st and third row, second column elements
[1] 1 3

```

Matrices are added/subtracted elementwise as with textbook matrix addition. If you add two matrices together that are not conformable R will return a matrix whose length is equal to the length of the longest vector which occurs in the expression along with a warning message. The smaller matrix is recycled as often as needed to match the size of the largest matrix in the expression. The following code illustrates this

```

> a <- c(0,1,2)    ## Create a 3x1 vector
> d <- c(a,a,2)    ## Create a 7x1 vector
> f <- a+2*d+7      ## In principle we cannot add these two
>                  ## vectors together since they are not
>                  ## conformable, but R will do this
> f
[1] 7 10 13 7 10 13 11

```

You will notice that I have already used the \* and + symbols for multiplication and addition. Other common mathematical operators in R are listed in Table 1.

Aside from basic mathematical operations there are also a host of additional commands that work with vectors and matrices to extract important information. If you wanted to know the length of a vector you could use the `length()` command, whereas `range()` will return a vector composed of the minimum and the maximum of the vector that is passed to it. Alternatively, you could simply use `min()` and `max()` separately. `min()` and `max()` are global operators as they act on everything that is passed into it, i.e., if I were to pass two vectors to `min()`, the return would be the smallest value across both vectors. The same thing goes for `max()`.

TABLE 1. Summary of Basic Mathematical Operators in R.

Operator	Mathematic Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
^	Power
sqrt()	Square Root
exp()	Exponential
log()	Logarithm in Base $e$
log10()	Logarithm in Base 10
sin()	Sine
cos()	Cosine
tan()	Tangent
abs()	Absolute Value
%%	Modular Arithmetic
/%	Integer Division

```

> length(f)      ## Return the length of the vector f
[1] 7
> range(f)       ## Return the min and max of f
[1] 7 13
> max(f,(f+2))   ## max() only returns a scalar
[1] 15

```

To order a vector the `sort()` command returns a vector of the same length with the elements of the original vector ordered from smallest to smallest. Additional sorting facilities include `order()` and `sort.list()`. To sum the elements of a vector simply use `sum()` and to multiply the elements of a vector use `prod()`. R also offers a range of cumulative operators. To perform cumulative addition, multiplication, minimum and maximum calculations use the commands `cumsum()`, `cumprod()`, `cummin()` and `cummax()`

When you are calling commands on either scalars, vectors or numbers you typically will not be concerned if R stores your answers as an integer, a real number or a complex number. From R's perspective all calculations are done in double precision treating values as real numbers. Thus, if you wanted to work with complex numbers you would need to explicitly include the complex component of the number. For example

```

> sqrt(-1)       ## Returns NaN along with a warning
[1] NaN
> sqrt(-1+0i)    ## Conducts the operation with no issues
[1] 0+1i

```

**3.1. Creating Sequences.** There are a number of available options to construct sequences of numbers in R. When you are constructing your own code typically you will have sequences of numbers, for example if you have a loop that runs over your observations this is a sequence of the integers from 1 to  $n$ , where  $n$  is your sample size. The simplest way to construct sequences is with the colon operator. A more precise way to construct sequences is to use the `seq()` command. This command generally has three inputs, the start of the sequence, the end of the sequence and the step length between elements in the sequence.

```
> 1:5          ## Simple sequence
[1] 1 2 3 4 5

> 1:5*2 ## Order of operation with colon
[1] 2 4 6 8 10

> seq(2,10,2)    ## More direct way to construct sequences
[1] 2 4 6 8 10
```

The `seq()` is a good way to introduce you to the capabilities of R when calling a function. Technically, the first element that the `seq()` command looks for is the **from** variable, the second is the **to** variable and the third is the **by** variable. So, putting this together we tell R to construct a sequence **from** a given number **to** another number, **by** taking steps of a given size. More formally we have

```
> seq(from=2,to=10,by=2)    ## Formal Construction
[1] 2 4 6 8 10

> seq(2,10,2)              ## Quick Construction
[1] 2 4 6 8 10
```

Notice that they are the same. R will interpret your inputs in a specific order so that you don't have to remember the names of each of the inputs. What is even more interesting is that you can send in variables with their names out of the order in which the R function is constructed

```
> seq(by=2,from=2,to=10)    ## Ordering Backwards
[1] 2 4 6 8 10
```

The `rep()` command is related to the `seq()` command. `rep()` will construct a vector of repeated values of the first element passed with length equal to the second value passed. An option within `rep()` is **each**. If you specify **each** then each element of the first value passed (if it is a vector) is repeated **each** times whereas if you do not specify **each** the whole vector is repeated via the number of times passed to the `rep()` call.

```
> x <- c(1,2,3) ## Create Vector
> rep(1,5)      ## Illustrate simple call to rep()
[1] 1 1 1 1 1

> rep(x,2)      ## Basic rep() call with vector
```

```
[1] 1 2 3 1 2 3
> rep(x,each=2)      ## rep() using each
[1] 1 1 2 2 3 3
```

**3.2. Constructing Logical Vectors.** Not only can R construct and hold information on numbers, it can also store logical information. This allows for easy manipulation of your data when subsetting and constructing ifelse statements. A logical vector in R can contain the values `TRUE`, `FALSE` and `NA`. The first two can be abbreviated in the R language as `T` and `F`. A common mistake is to use `T` and `F` exclusively. However, only `TRUE` and `FALSE` are reserved words so it is very easy to overwrite things if one only uses `T` and `F`. A logical vector is generated by a condition. An example is

```
> a <- c(0,1,-1,2) ## Construct vector
> b <- a > 0        ## Create logical vector with condition
> b                ## Print out logical vector
[1] FALSE  TRUE FALSE  TRUE
```

The condition works element wise on the vector it is assessing. The standard logical operators are in Table 2.

TABLE 2. Summary of Basic Logical Operators in R.

Operator	Logical Device
>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To
==	Equal To
!=	Not Equal To
&	Intersection of Conditions
	Union of Conditions
!	Negation of a Condition

A wonderful command at your disposal in R relating to logical vectors is `which()`. The `which()` command allows you to determine the elements of a vector which satisfy a set of conditions.

```
> which(a<=0) ## Determine elements that are nonpositive
[1] 1 3
```

**3.3. Character Vectors.** Constructing vectors which are nonnumeric is also easy in R. These types of vectors will be useful to you when you are plotting detailed information as well as naming a set of variables, amongst other settings. To construct a vector that contains characters simply use parentheses, " " around each variable.

```
> a <- c("Dakar", "Senegal", "AGRODEP") ## Create Vector
> a      ## Print out character vector
```

```
[1] "Dakar"      "Senegal" "AGRODEP"
```

**Mention `paste()` command as well**

#### 4. DATA MANAGEMENT

So at this point I am hopeful that you are comfortable with the basics of using R. There are many additional commands we have not covered yet, but just like learning a spoken language, it is useful to focus on the few words you will use often and then learn more sophisticated words later on as you become more comfortable with the language. Now, in general you will not be constructing all of your data, rather importing it into R to engage in statistical analysis. We already mentioned a range of options to load datasets into R's memory banks. Here we will discuss some preliminary methods you can use to manipulate your data.

For the example here I am going to load the famous hedonic pricing dataset of Harrison & Rubinfeld (1978). This dataset is available in the `Ecdat` package (Croissant 2011) which was loaded when you installed the `Econometrics Task View`.<sup>1</sup> The `summary()` command is a useful first pass to generically inspect the data you have loaded to make sure that things look appropriate. When you call `summary()` the names of the variables may not be intuitive and so investigating the help page (`?Hedonic`) associated with this dataset will naturally be of use.

```
> library(Ecdat)          ## Load Ecdat Library to gain access to Hedonic Dataset
> data("Hedonic")        ## Place Dataset in R's memory
> summary(Hedonic)       ## Summarize Data
```

**4.1. The `data.frame()` Environment.** Whether you load data from outside of R or call a dataset that belongs to a package it is important to recognize its type. That is, you are most likely loading a set of numbers each with a special name. This is not how we typically think of a matrix. In fact, from R's perspective your data is a `data.frame()`. The `data.frame()` environment is an excellent way to store and manage your data. Almost universally the datasets within R packages are loaded as `data.frames`. If you load your data in with headers R will typically construct a `data.frame` out of the data passed to it as well.

The interesting thing about the `data.frame()` environment is that you can have two different datasets with variables of the same name and this will not cause confusion within R. That is, within the `Hedonic` dataset there is a variable called `mv` which stands for the median value of owner occupied housing in a given census tract, measured in \$1000. If we create a new dataset using the `data.frame()` operator, say for census tracts along the Charles river (`chas==1`), we could label the median value of housing in this dataset as `mv` as well with no ill effects.

To access a particular variable from a `data.frame()` the `$` operator is indispensable. Continuing with our example, if we desired to know the minimum and maximum values of `mv` in our data we can simply call

---

<sup>1</sup>See the `spdep` package (Bivand 2011) which makes several corrections to Harrison & Rubinfeld's (1978) original data given the issues raised by Gilley & Pace (1996).

```
> range(Hedonic$mv)  ## Min and max of mv
[1] 8.51719 10.81980
```

If instead you typed `range(mv)`, an error would be produced (Try it!). It may seem like having to type the name of a dataset, followed by the `$` operator, followed by the name of the variable, just to access a particular variable is a lot of work. However, a way to circumvent this is to attach your data. By attaching your data the names of each of the variables are entered into R's memory which means that you can directly access variables without calling them out of the `data.frame()`. However, a potential issue with this is that you will overwrite variables of the same name and/or create conflicts based on variable names. Thus, users need to be careful when attaching data.

```
> attach(Hedonic)      ## Attach dataset
> range(mv)  ## Min and max of mv
[1] 8.51719 10.81980
```

We can also construct variables within a `data.frame()` with relative ease. To do this we again resort to the `$` operator. It is common in hedonic pricing models to estimate a log-level or log-log variant. In the `Hedonic` dataset we do not have the logarithm of median value owner occupied housing. We can easily create this variable though as

```
> Hedonic$lmv <- log(mv)      ## Construct log of mv
```

If you wanted direct access to `lmv` you would want to reattach the entire `Hedonic` dataset with a call to `attach()` again. To remove a variable, such as the one we just created we could assign to it the `NULL` value.

```
> Hedonic$lmv <- NULL      ## Remove log of mv
```

**4.2. Subsetting Your Data.** To construct `data.frames` by hand there are several commands available. First, to generically construct a `data.frame()` you can simply place the variables you want inside a call to this command

```
> myHedonic <- data.frame(price=mv,crime=crim,pollution=nox) ## Construct new data.frame
```

To construct a `data.frame()` as a subset of an existing `data.frame` use the `subset()` command

```
> Charles <- subset(Hedonic,select=(chas==1))      ## Only Tracts along Charles River
> highvalue <- subset(Hedonic,mv>9.9)      ## High value census tracts
> High.Charles <- subset(Hedonic, mv>=9.9 &
+                          chas==1) ## Both high value and on Charles River
> No.rad <- subset(Hedonic,select=-rad)      ## Remove rad variable
> Few <- subset(Hedonic,select=mv:nox)      ## Alternative way to subset
```

Again, remember that in each of these five new `data.frames` that I constructed there are variables of the same name, which is fine as long as I don't attach each of these five datasets.

4.2.1. *Creating New Variables with `replace()` and `ifelse()`.* An alternative way to construct values, based off of subsetting your data, is to use the `replace()` command. Earlier I created a subset of data based on high value homes. Alternatively, I could have created a new variable called `highval` and placed that inside the `Hedonic` dataset to subset off of, or to use as a threshold variable in a linear regression. I could have done this as

```
> Hedonic$highval <- Hedonic$mv ## Create new variable
> Hedonic$highval <- replace(Hedonic$highval, Hedonic$highval<9.9, 0) ## 0 for low value
> Hedonic$highval <- replace(Hedonic$highval, Hedonic$highval>=9.9, 1) ## 1 for high value
```

This example was purely for illustration. A different way to construct the same variable would have been to the `ifelse()` command.

```
> Hedonic$highval <- ifelse(Hedonic$mv>=9.9,1,0) ## Create new variable
```

I listed both of these commands here to expose you to the available commands within R.

4.3. **Missing Values.** In general missing values for some of your variables will arise. To remedy missing values a common approach is to drop full observations where a missing variable is present. Since there are no missing values in the `Hedonic` dataset we will load a different dataset to illustrate the handling of missing values. The `AER` package (Kleiber & Zeileis 2008) contains a number of prominent datasets, amongst them the cross-country economic growth dataset of Durlauf & Johnson (1995).

```
> library(AER) ## Load AER Library to gain access to GrowthDJ dataset
> data("GrowthDJ") ## Place Dataset in R's memory
> summary(GrowthDJ) ## Summarize Data
```

You will see that the `summary()` command informs you which variables have missing values and how many. To eliminate the missing values from our dataset one simple command is `na.omit()`. Using this call on a `data.frame` will remove all rows that have at least one missing observation.

```
> GrowthDJ1 <- na.omit(GrowthDJ) ## Remove NAs
> summary(GrowthDJ1) ## Summarize Data
```

Notice how I did not rename the dataset with the same name, this is commonly bad practice since I would need to reload the full dataset from the `AER` package if I wanted any information on the rows which were dropped.

## 5. MATRIX ALGEBRA

Working with matrices in R is also a breeze. Assume we have a matrix **A** of order  $n \times k$ . In R we can define a matrix in a variety of ways. Perhaps the most straightforward approach is to use the `matrix()` command. We construct a matrix as

```
> A <- matrix(1:10,5,2) ## Construct a 5 by 2 matrix
> A
```

```

      [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> A <- matrix(1:10,5,2,byrow=TRUE)      ## Construct a 5 by 2 matrix
>                                     ## but fill in by row first
> A
      [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10

```

Many times when working with multiple matrices you may have a conformability issue. To check the size of your matrix simply use

```

> dim(A)      ## Returns number of rows and columns
[1] 5 2
> nrow(A)     ## Just the number of rows
[1] 5
> ncol(A)     ## Just the number of cols
[1] 2

```

Indexing matrices works as dicussed earlier as well as constructing logical statements surrounding matrices

```

> A>=4        ## Determine which elements are at least as large as 4
      [,1] [,2]
[1,] FALSE FALSE
[2,] FALSE  TRUE
[3,]  TRUE  TRUE
[4,]  TRUE  TRUE
[5,]  TRUE  TRUE

```

Diagonal matrices can easily be constructed using the `diag()` command.

```

> B <- diag(1,3,3)      ## Create 3 by 3 diagonal matrix
> B
      [,1] [,2] [,3]
[1,]    1    0    0

```



```
[2,]    0    1    0
[3,]    0    0    1
```

The calls `upper.tri()` and `lower.tri()` can be used to construct upper and lower triangular matrices as well.

**5.1. Matrix Operations.** In general, operations on matrices in R work elementwise, so taking

```
> sin(A) ## Take sine of A
      [,1] [,2]
[1,] 0.8414710 0.9092974
[2,] 0.1411200 -0.7568025
[3,] -0.9589243 -0.2794155
[4,] 0.6569866 0.9893582
[5,] 0.4121185 -0.5440211
```

takes the sine of each of the 10 elements. Addition and subtraction work identically to that of scalar/vector addition discussed earlier.

```
> B <- matrix(31:40,5,2)      ## Create Additional 5 by 2 matrix
> 0.5*A+B
      [,1] [,2]
[1,] 31.5  37
[2,] 33.5  39
[3,] 35.5  41
[4,] 37.5  43
[5,] 39.5  45
```

Matrix multiplication works as described in a basic matrix algebra class but you cannot use the `*` operator. Instead, multiplication of matrices is undertaken using `%%` (inner product). In our previous example we had that both `A` and `B` were  $5 \times 2$  matrices so clearly  $A \cdot B$  will not be a viable object since the matrices do not conform for multiplication. We can transpose either of the two matrices however to obtain something meaningful. Matrix transposition is done using `t()`.

```
> t(A)%%B      ## Multiply A' by B
      [,1] [,2]
[1,]  845  970
[2,] 1010 1160

> tcrossprod(A,B)      ## Slightly faster way to take crossproducts
      [,1] [,2] [,3] [,4] [,5]
[1,]  103  106  109  112  115
[2,]  237  244  251  258  265
[3,]  371  382  393  404  415
```

```
[4,] 505 520 535 550 565
[5,] 639 658 677 696 715
```

Additional multiplication operators for matrices include calculation of the outerproduct, `%o%` and the Kronecker product, `%x%`.

Matrix inversion is also a bit more involved. The `solve()` command is the operator to invert a matrix. The `det()` command allows you to compute the determinant of a matrix to check if the matrix is capable of being inverted.

```
> det(crossprod(A))          ## Take determinant of A'A
[1] 200

> solve(crossprod(A))        ## Inverse of A'A
      [,1] [,2]
[1,]  1.10 -0.950
[2,] -0.95  0.825

> solve(t(A)%*%A)           ## Check
      [,1] [,2]
[1,]  1.10 -0.950
[2,] -0.95  0.825
```

To compute the spectral decomposition of a matrix the `eigen()` command is invaluable. This command creates an object that stores the eigenvalues, `$values`, and eigenvectors, `$vectors`.

```
> eigen(crossprod(A))       ## Compute eigen values/vectors of A'A
$values
[1] 384.4798166  0.5201834

$vectors
      [,1] [,2]
[1,] 0.6545058 -0.7560570
[2,] 0.7560570  0.6545058
```

## 6. BASIC STATISTICS

A variety of basic statistics commands are at one's disposal in R both within the main R setup as well as the `stats` package which is automatically loaded when you open R. These include calculation of the mean, variance, covariance, standard deviation and quantiles. A basic set of calls using these commands for the `Hedonic` dataset are

```
> mean(nox)                ## Mean of nox levels across tracts
[1] 32.1088

> sd(nox)                   ## Standard deviation of nox
[1] 13.92117
```

```

> quantile(nox,c(0.25,0.75))      ## Upper and lower quantiles of nox
      25%      75%
20.1601 38.9376
> IQR(nox)                       ## Inner quantile range of nox
[1] 18.7775
> cor(nox,mv)                    ## correlation b/w nox and mv
[1] -0.4964548
> mad(nox)                      ## Median absolute deviation of nox
[1] 13.28929

```

Several of the statistical calls will not work properly if the data passed contain missing observations. A simple way to avoid issues is to use the `na.rm` option that is available in many of the calls, for example in `mean`, `cor`, `median`, `quantile` and `sd`.

There are even a variety of graphical facilities at your disposal, such as a quantile-quantile (QQ) plot to assess normality. The following code

```

> nox.stud <- (nox-mean(nox))/sd(nox) ## Standardize nox
> qqnorm(nox.stud)                  ## Produce Q-Q plot for nox.stud
> qqline(nox.stud,col="red",lwd=2)

```

will produce the Normal Q-Q plot in Figure 10.

Another useful plotting device is the boxplot. We can use the following code

```

> boxplot(nox,notch=TRUE,xlab="NOX") ## Produce boxplot for nox

```

will produce the boxplot in Figure 11.

For a list of all the statistical options available type `library(help="stats")`.

**6.1. Probability Distributions.** R provides a powerful interface to obtain values for the cumulative distribution function, probability density function and the quantile function of a variety of distributions. Many of these commands also generate random deviates which is useful for generating your own datasets for DGPs as well as performing calculations with simulated methods. Table 3 lists many of the common calls. Most of the popular distributions can be accessed within the `base` package. Furthermore, there are two additional packages which offer additional distributions as well as many of the distributions listed here. These are `distr` and `distrEx` (Ruckdeschel, Kohl, Stabla & Camphausen 2006) and `SuppDists` (Wheeler 2009). In sum, R offers an attractive array of statistical distributions which can be accessed with relative ease and a common set of calls for each of the associated functions.

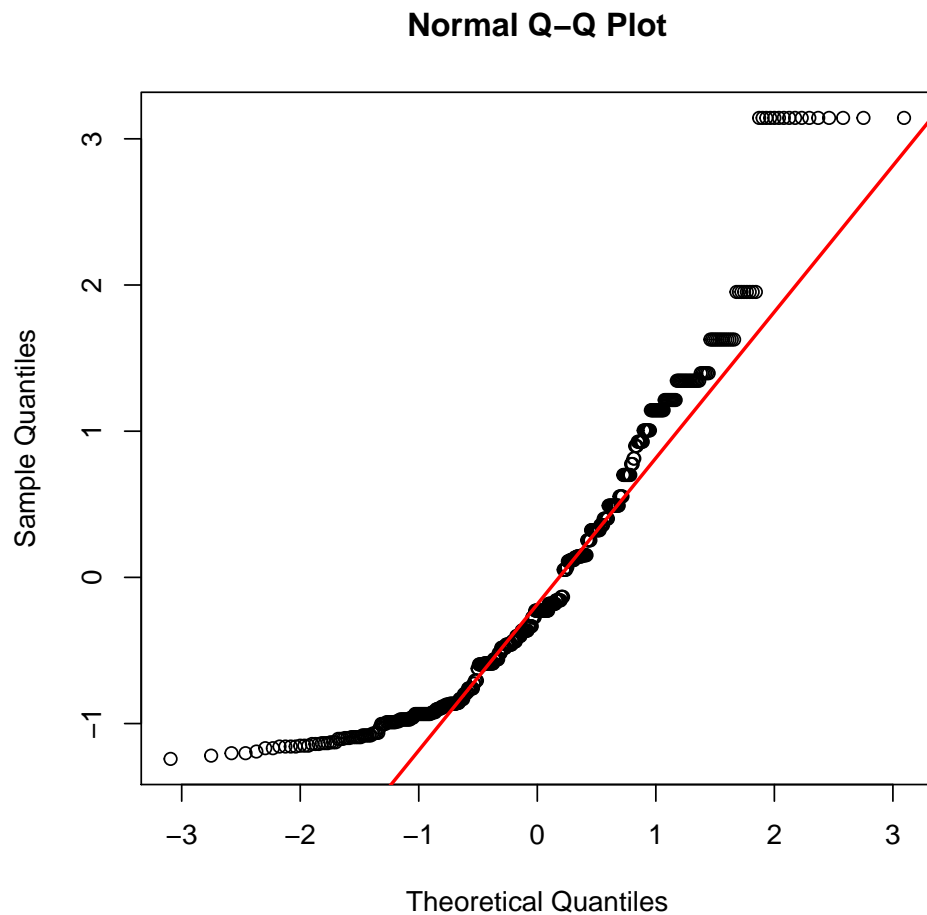
For each of the listed distributions, by placing the letters `d`, `p`, `q`, and `r` you can access the density, distribution, quantile and random generator, respectively. For example

```

> set.seed(1)                    ## Set seed of random number generator for full replicability
> dpois(c(0,1,2,3),lambda=1)    ## pdf values for c(0,1,2,3)

```

FIGURE 10. Q-Q plot for nox.

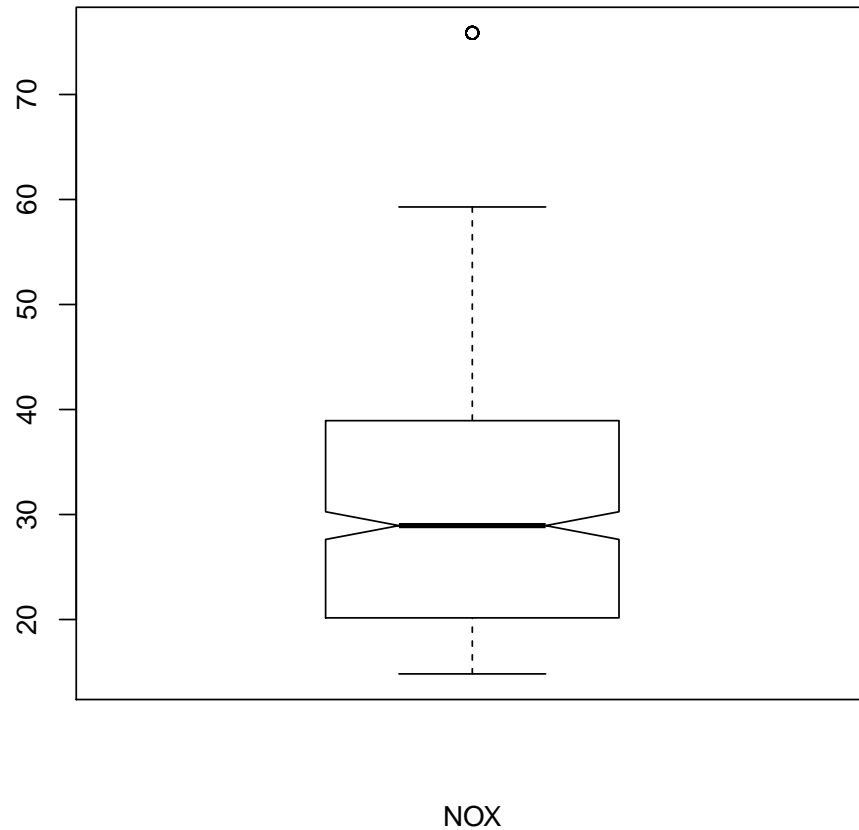


```
[1] 0.36787944 0.36787944 0.18393972 0.06131324
> ppois(c(0,1,2,3),lambda=1)      ## CDF values for c(0,1,2,3)
[1] 0.3678794 0.7357589 0.9196986 0.9810118
> qpois(c(0.01,0.45,0.75),lambda=1)  ## 0.01,0.45, and 0.75 quantiles
[1] 0 1 2
> rpois(10,lambda=1)              ## Generate pseudo-random Poisson deviates.
[1] 0 1 1 2 0 2 3 1 1 0
```

## 7. WRITING A LOOP

When constructing your own code typically you will have repeated commands which you loop over, such as when you engage in bootstrapping. Writing a loop in R is very simple. The `for()` construction will allow you to write a loop. The `for()` construction works as follows

FIGURE 11. Boxplot for nox.



```
> for (name in expression1) {expression2}

where name is the variable you are looping over, expression1 is a sequence or vector expression
and expression2 is the expression you wish to repeatedly evaluate. For example

> mn <- numeric()          ## Create storage
> for (j in 1:10){          ## Loop will repeat 10 times
+
+     mn[j] <- mean(rnorm(100)) ## Take mean of 100
+                               ## Standard normal deviates
+ }
> mean(mn)                  ## Take the mean of the means
[1] -0.01063824
```

TABLE 3. Summary of Probability Distributions.

Distribution	R Name	Package
beta	beta	stats
binomial	binom	stats
Cauchy	cauchy	stats
$\chi^2$	chisq	stats
Exponential	exp	stats
F	f	stats
gamma	gamma	stats
generalized extreme value	gev	evir
geometric	geom	stats
hypergeometric	hyper	stats
log-normal	lnorm	stats
logistic	logis	stats
Multivariate normal	mvnorm	mvtnorm
Multivariate t	mt	LearnBayes
negative binomial	nbinom	stats
normal	norm	stats
Pareto	pareto	actuar
Poisson	pois	stats
Student's t	t	stats
uniform	unif	stats
Weibull	weibull	stats
Wilcoxon	wilcox	stats

Other looping constructions in R include the `repeat` and `while` statements, though these are used much less often than `for`.

## 8. THE LINEAR MODEL

Undoubtedly, ordinary least squares (OLS) is the workhorse statistical model for economists. Universally, all statistical packages offer some form of conducting a regression analysis and R is no different. The main call for running a linear model in R is `lm()`. A generic invocation of `lm()` is undertaken with a formula. A call to `lm()` produces an object of class `lm` which will contain a number of key aspects of your regression analysis such as coefficient estimates and standard errors. In the following example we will conduct a simple log-linear hedonic analysis from within the `Hedonic` dataset. Our dependent variable is `log(mv)` and our explanatory variables are `chas` and `nox`.

```
> Hedonic$lmv <- log(mv)          ## Construct log of mv
> hedonic.model <- lm(lmv~nox+chas,data=Hedonic) ## Run regression
> attributes(hedonic.model)       ## List all attributes of lm object

$names
[1] "coefficients" "residuals"      "effects"          "rank"
```

```
[5] "fitted.values" "assign"      "qr"          "df.residual"
[9] "contrasts"     "xlevels"     "call"        "terms"
[13] "model"
```

```
$class
```

```
[1] "lm"
```

```
> summary(hedonic.model)      ## Summarize results
```

```
Call:
```

```
lm(formula = lmv ~ nox + chas, data = Hedonic)
```

```
Residuals:
```

```
      Min       1Q   Median       3Q      Max
-0.126756 -0.019075 -0.002391  0.018099  0.102954
```

```
Coefficients:
```

```
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.3434945   0.0039107  599.253 < 2e-16 ***
nox          -0.0015545   0.0001123  -13.840 < 2e-16 ***
chasyes       0.0340958   0.0061561    5.539 4.92e-08 ***
```

```
---
```

```
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
Residual standard error: 0.03496 on 503 degrees of freedom
```

```
Multiple R-squared:  0.2934,    Adjusted R-squared:  0.2906
```

```
F-statistic: 104.4 on 2 and 503 DF,  p-value: < 2.2e-16
```

The `attributes()` command is exceptionally useful when determining all of the pieces of an object. You will see that the `hedonic.model` object has 13 pieces. Any of these pieces can be easily accessed by typing `hedonic.model$name` where `name` is the object you want. Notice that we did not specify an intercept in our model, yet one appeared. This is because the default in `lm()` is to include an intercept. If you wanted to estimate a regression model without an intercept then the formula `lmv nox+chas-1` would do the trick.

Suppose you were concerned that the coefficients of the hedonic model varied whether or not the location of the property was near the Charles river. In this case a standard F-test would allow you to investigate this hypothesis. To formally test this two models would need to be estimated, one including only tracts along the Charles river and another for tracts away from the Charles river. In this case, we can use the `subset` option within `lm`

```
> hedonic.model.chas <- lm(lmv~nox,data=Hedonic,subset=chas=="yes") ## Run regression
> hedonic.model.nochas <- lm(lmv~nox,data=Hedonic,subset=chas!="yes") ## Run regression
```

The corresponding residual sum of squares could be stripped off from each of these objects to construct the desired F-statistic.

There are times when we will want to include transformations of our regressors into the linear model. Examples include linear combinations amongst several regressors, interactions, higher order terms and logarithmic transforms, to name a few. These types of transformations can easily be included directly into your `lm()` call without creating the corresponding variable within R's workspace. For example, suppose we wanted to include the square of `nox` because we believed a form of nonlinearity existed in our hedonic function. To account for this we would use

```
> hedonic.model <- lm(lmv~nox+I(nox^2)+chas,data=Hedonic) ## Run regression
> summary(hedonic.model)          ## Summarize results
```

Call:

```
lm(formula = lmv ~ nox + I(nox^2) + chas, data = Hedonic)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-0.121704	-0.018517	-0.002409	0.016570	0.109649

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	2.380e+00	8.909e-03	267.194	< 2e-16 ***
nox	-3.734e-03	4.881e-04	-7.650	1.03e-13 ***
I(nox^2)	2.722e-05	5.938e-06	4.584	5.77e-06 ***
chasyes	3.131e-02	6.068e-03	5.161	3.55e-07 ***

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03428 on 502 degrees of freedom

Multiple R-squared: 0.3218, Adjusted R-squared: 0.3178

F-statistic: 79.4 on 3 and 502 DF, p-value: < 2.2e-16

The use of `I()` is meant to inform `lm()` to interpret the mathematical command as a new variable within the `lm()` environment. `I()` literally means Identity function. A setting where you do not need the `I()` call is when you want interactions. We can include interactions of our regressors using the colon, `:`.

```
> hedonic.model <- lm(lmv~nox+chas+nox:chas,data=Hedonic) ## Run regression
> summary(hedonic.model)          ## Summarize results
```

Call:

```
lm(formula = lmv ~ nox + chas + nox:chas, data = Hedonic)
```



Residuals:

	Min	1Q	Median	3Q	Max
	-0.125533	-0.018733	-0.002874	0.017546	0.103714

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	2.3458738	0.0041257	568.602	<2e-16 ***
nox	-0.0016294	0.0001198	-13.606	<2e-16 ***
chasyes	0.0120167	0.0138615	0.867	0.3864
nox:chasyes	0.0006038	0.0003398	1.777	0.0762 .

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03488 on 502 degrees of freedom

Multiple R-squared: 0.2978, Adjusted R-squared: 0.2936

F-statistic: 70.98 on 3 and 502 DF, p-value: < 2.2e-16

A shorthand way to code exactly the same model in the `lm()` environment is to use

```
> hedonic.model <- lm(lmv~nox*chas,data=Hedonic) ## Run regression
> summary(hedonic.model) ## Summarize results
```

Call:

```
lm(formula = lmv ~ nox * chas, data = Hedonic)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-0.125533	-0.018733	-0.002874	0.017546	0.103714

Coefficients:

	Estimate	Std. Error	t value	Pr(> t )
(Intercept)	2.3458738	0.0041257	568.602	<2e-16 ***
nox	-0.0016294	0.0001198	-13.606	<2e-16 ***
chasyes	0.0120167	0.0138615	0.867	0.3864
nox:chasyes	0.0006038	0.0003398	1.777	0.0762 .

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03488 on 502 degrees of freedom

Multiple R-squared: 0.2978, Adjusted R-squared: 0.2936

F-statistic: 70.98 on 3 and 502 DF, p-value: < 2.2e-16

Using the “\*” in the formula for `lm` creates all possible interactions, whereas the “:” only creates a single interaction. Note the difference in the following two calls

```
> lm(lmv~nox*chas*crim,data=Hedonic) ## Run regression
```

Call:

```
lm(formula = lmv ~ nox * chas * crim, data = Hedonic)
```

Coefficients:

(Intercept)	nox	chasyes	crim
2.3312670	-0.0009388	0.0061531	0.0028423
nox:chasyes	nox:crim	chasyes:crim	nox:chasyes:crim
0.0002407	-0.0001062	0.0331575	-0.0005040

```
> lm(lmv~nox:chas:crim,data=Hedonic) ## Run regression
```

Call:

```
lm(formula = lmv ~ nox:chas:crim, data = Hedonic)
```

Coefficients:

(Intercept)	nox:chasno:crim	nox:chasyes:crim
2.305e+00	-5.984e-05	2.588e-05

To quickly access key features of interest from your `lm` object, a set of basic commands already exist to assist you. If you wanted to extract the coefficients of your model you can use

```
> model <- lm(lmv~nox+chas+crim,data=Hedonic) ## Run regression
```

```
> coefficients(model) ## Extract coefficients
```

(Intercept)	nox	chasyes	crim
2.335461267	-0.001081467	0.027958250	-0.001862245

or the variance covariance matrix

```
> vcov(model) ## Extract variance-covariance matrix
```

	(Intercept)	nox	chasyes	crim
(Intercept)	1.325092e-05	-3.666364e-07	1.461268e-07	1.406185e-07
nox	-3.666364e-07	1.253323e-08	-8.490343e-08	-8.279418e-09
chasyes	1.461268e-07	-8.490343e-08	3.168721e-05	1.074348e-07
crim	1.406185e-07	-8.279418e-09	1.074348e-07	3.259767e-08

```
> sqrt(diag(vcov(model))) ## Standard errors
```

(Intercept)	nox	chasyes	crim
0.0036401820	0.0001119519	0.0056291390	0.0001805482

or the residual sum of squares

```
> deviance(model) ## RSS
```

```
[1] 0.5072382
```

and the fitted values and residuals

```
> fit <- fitted(model)      ## Extract fitted values
> uhat <- residuals(model) ## Extract residuals
```

**8.1. Generalizations of the Linear Model.** Aside from the basic linear model a whole suite of alternative ‘regression’ based estimators exists. The main alternative to `lm()` within the stock install of R is `glm()` which stands for generalized linear model. The `glm()` environment will allow the user to estimate logit, Probit and Poisson regression models with ease. For example, if we load the `Somerville` dataset in the `Ecdat` package, we can examine the determinants of going on a fishing trip.

```
> data(Somerville) ## Load dataset
> Somerville$trip <- Somerville$visits>0 ## Observations where a trip was taken
> model.logit <- glm(trip~quality+ski+income+feeSom,
+                   data=Somerville,
+                   family=binomial(link="logit"))
> model.probit <- glm(trip~quality+ski+income+feeSom,
+                   data=Somerville,
+                   family=binomial(link="probit"))
> summary(model.logit)      ## Summarize logit results

Call:
glm(formula = trip ~ quality + ski + income + feeSom, family = binomial(link = "logit"),
    data = Somerville)
```

Deviance Residuals:

	Min	1Q	Median	3Q	Max
	-2.9978	-0.3204	-0.3047	0.3116	2.4794

Coefficients:

	Estimate	Std. Error	z value	Pr(> z )
(Intercept)	-2.84039	0.36786	-7.721	1.15e-14 ***
quality	1.48106	0.10069	14.709	< 2e-16 ***
skiyes	0.17503	0.31262	0.560	0.576
income	-0.05158	0.08205	-0.629	0.530
feeSomyes	16.79490	883.43139	0.019	0.985

---

Signif. codes: 0 ‘\*\*\*’ 0.001 ‘\*\*’ 0.01 ‘\*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 866.53 on 658 degrees of freedom  
 Residual deviance: 337.28 on 654 degrees of freedom  
 AIC: 347.28

Number of Fisher Scoring iterations: 16

```
> summary(model.probit)      ## Summarize probit results
```

Call:

```
glm(formula = trip ~ quality + ski + income + feeSom, family = binomial(link = "probit"),
    data = Somerville)
```

Deviance Residuals:

Min	1Q	Median	3Q	Max
-3.1494	-0.2975	-0.2788	0.3235	2.4847

Coefficients:

	Estimate	Std. Error	z value	Pr(> z )
(Intercept)	-1.69398	0.18801	-9.010	<2e-16 ***
quality	0.82112	0.04850	16.932	<2e-16 ***
skiyes	0.14358	0.16111	0.891	0.373
income	-0.01975	0.04232	-0.467	0.641
feeSomyes	5.74592	129.92844	0.044	0.965

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

Null deviance: 866.53 on 658 degrees of freedom  
 Residual deviance: 336.92 on 654 degrees of freedom  
 AIC: 346.92

Number of Fisher Scoring iterations: 15

The `erer` package (Sun 2011) is useful when marginal effects and their associated standard errors are desired for logit/probit models.

Instead of summarizing the number of visits to Lake Somerville as a binary indicator, we could use the true count nature of visits and estimate a Poisson regression. To do that with the `glm` command we would use

```
> model.pois <- glm(visits~quality+ski+income+feeSom,
+                   data=Somerville,
```

```

+                                family=poisson)
> summary(model.pois)           ## Summarize poisson results

Call:
glm(formula = visits ~ quality + ski + income + feeSom, family = poisson,
    data = Somerville)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-7.6003  -1.2508  -1.0677  -0.7387   22.3193

Coefficients:
              Estimate Std. Error z value Pr(>|z|)
(Intercept) -0.03705     0.08364  -0.443   0.658
quality      0.52814     0.01555  33.967 < 2e-16 ***
skiyes       0.31668     0.05505   5.753 8.78e-09 ***
income      -0.17505     0.01900  -9.214 < 2e-16 ***
feeSomyes    1.28430     0.07876  16.307 < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for poisson family taken to be 1)

Null deviance: 4849.7  on 658  degrees of freedom
Residual deviance: 2974.2  on 654  degrees of freedom
AIC: 3737.2

```

Number of Fisher Scoring iterations: 7

The `glm()` call is extremely powerful. If you specify `family=gaussian` you would conduct OLS. Use `?family` to learn more about the available statistical families that `glm` deploys.

Table 4 lists a variety of common econometric methods for which packages are available in R for implementation.

## 9. GRAPHICS IN R

Graphics are a key component to any econometric exercise. Graphics in R can be done interactively which provides an array of manifestations of your data/results. There exist both high-level plotting functions, which create new plots, and low-level plotting functions, which add additional features to an already existing plot. A high-level plotting command will always erase a current plot so it is important to know which calls do what to preserve already displayed information. Perhaps

TABLE 4. Summary of Mainstream ‘Regression’ Packages.

Method	R call	Package
IV Regression	<code>ivreg()</code>	AER
Heckman Selection	<code>heckit()</code>	sampleSelection
Seemingly Unrelated Regression	<code>systemfit()</code>	systemfit
3SLS	<code>systemfit()</code>	systemfit
Panel Data	<code>plm()</code>	plm
Quantile Regression	<code>rq()</code>	quantreg
Tobit Regression	<code>tobit()</code>	AER
Negative Binomial Regression	<code>glm.nb()</code>	MASS
Censored Regression	<code>censReg()</code>	censReg
Ordered Logistic/Probit Regression	<code>polr()</code>	MASS
Generalized Additive Modelling	<code>gam()</code>	mgcv
Spatial Regression	<code>GMerrorsar()</code>	spdep
Bayesian Model Averaging	<code>bms()</code>	bms

the most frequently used plotting command in R is the `plot()` command. `plot()` is designed to display the results of a number of different R objects.

The plot command can be used to plot out your raw data as follows:

```
> plot(nox,lmv,cex=0.9,col="red") ## Produce data plot for (nox,lmv)
```

This code produces Figure 12.

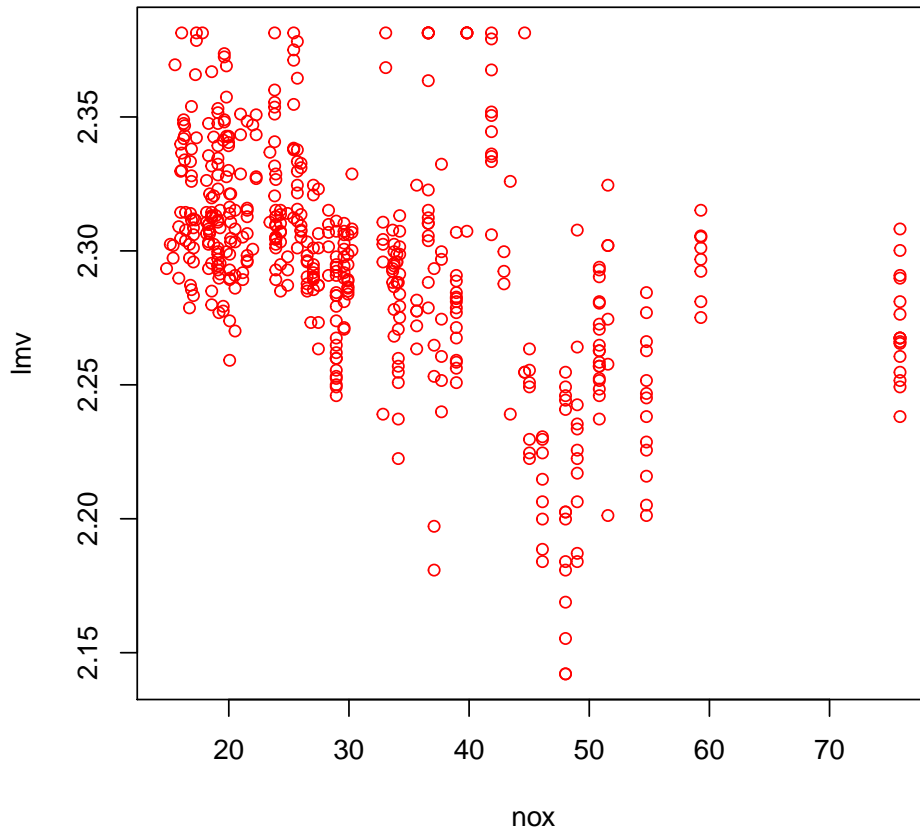
There are a number of available options to customize the plotting environment in R. Use `?par` to access these options. Additional high-level plotting facilities in the `graphics` package in R are `qqplot()` (Q-Q plot), `hist()` (Histogram), `persp()` (3D surface) and `contour()` (contour lines of a 3D surface).

We could augment our plots using low-level plotting facilities to make them more appealing and visually informative. For example, if I wanted to add the fitted OLS line to the scatterplot of data in Figure 12 I could use

```
> abline(lm(lmv~nox,data=Hedonic),
+       lty=2,lwd=2,col="blue") ## Produce fitted linear model
```

This additional code produces Figure 13.

Further low-level plotting devices include `points()` (overlay a scatterplot), `lines()` (overlay a line), `text()` (place text on your graph), and `legend()` (add a legend to your plot). When including a legend or text into your plot you will typically need to ‘fix’ a corner of the legend. A simple way to do this is with the `locator()` function. After producing a plot, to determine the location (on the plot) where you would like your legend, simply type `locator()` and a cross hatch will appear as your mouse when you move over the plot. Click on the area of the plot where you want the upper left corner of the legend to appear. Then hit escape and you will return to the R console where the coordinates have been printed out. You can then use these when you call the placement of the legend or text. If you want to use mathematical symbols in either your legend, the text you

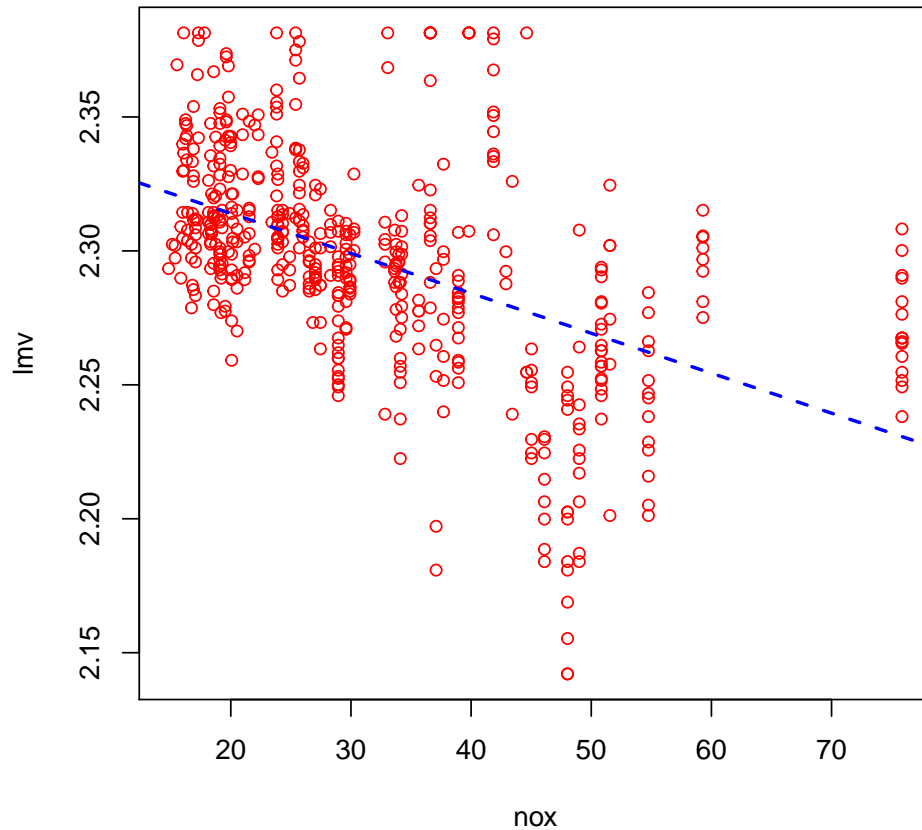
FIGURE 12. Scatter plot for `nox` and `lmv`.

add to your plot or the labeling of the axes and title of the plot, the `plotmath` environment is a lifesaver. Type `?plotmath` and a variety of examples will pop up that should get you on the right path.

If you want to produce multiple plots on the same figure you can use `par(mfrow=c(2,2))` prior to your invocations of `plot()` and this will create a plot surface that is two plots across and two plots down. Clearly you could change the `c(2,2)` condition to match your desired effect.

Invariably you will want to place your plots into your text editor (hopefully  $\text{\LaTeX}$ !). To do this you can tell R to create your plot as a variety of devices including postscript (.ps), pdf (.pdf) and JPEG (.jpeg). To do this you would use the code

```
> pdf(file="Scatterplot.pdf")          ## Send plots to pdf Device
> plot(nox,lmv,cex=0.9,col="red")      ## Produce data plot for (nox,lmv)
> dev.off()                            ## Turn off device
```

FIGURE 13. Fitted OLS regression line of `nox` and `lmv`.

which will produce what appears in Figure 12. Make sure that you call `dev.off()` after you are finished otherwise you may have trouble opening your plots. For more on the devices you can send plots to type `?Devices`.

For more sophisticated graphics, such as dynamic graphics see the `lattice` (Sarkar 2008), `igraph` (Csardi & Nepusz 2006) and `rgl` (Adler & Murdoch 2011) packages.

## 10. WRITING YOUR OWN FUNCTION

While R offers an impressive range of statistical commands to conduct an econometric analysis, inevitably there will be methods and procedures that you will want to use that are not currently available in R (perhaps you developed a new estimator). In these instances creating your own ‘function’ to construct an R object will be of interest to you. Becoming a skilled writer of functions is a key contributor to enhancing your enjoyment of R and will make you a more productive researcher.



A function is defined by an assignment expression of the form

```
> name <- function(argument1, argument2, ..., argumentJ) {expression}
```

The `expression` is a typical R assignment using as inputs the `arguments` you pass to the function. The value of `expression` is returned by the function. The function is then called as `name(argument1, argument2, ..., argumentJ)` anywhere it is legitimate.

A simple example of constructing a function is would be to calculate the mean of a vector. The function is defined as follows:

```
> Mean <- function(x) {
+     n <- length(x)          ## Calculate sample size
+     y.bar <- sum(x)/n      ## Calculate mean
+     y.bar                  ## Have function return mean
+ }
> y <- rnorm(100)
> Mean(x=y)                 ## Calculate mean with Mean()
[1] -0.02153563
> mean(y)                   ## Calculate mean with mean()
[1] -0.02153563
```

A more direct example would be to construct a function which returns the OLS estimates. We already discussed estimating a linear model in R but this example is more in the spirit of constructing a function. Recall that the OLS estimates are constructed as  $\hat{\beta} = (X'X)^{-1}X'y$ .

```
> ols <- function(y,x) {
+     XX <- crossprod(x)      ##Construct X'X
+     XY <- t(x)%*%y          ## Construct X'y
+     beta.hat <- solve(XX)%*%XY ## Construct Estimate
+     return(beta.hat)        ## Return Estimate
+ }
> ols(y=mv,x=cbind(1,nox)) ## Calculate with ols()
      [,1]
10.41032016
nox -0.01457707
> lm(mv~nox)$coefficients ## Calculate with lm()
(Intercept)      nox
10.41032016 -0.01457707
```

We can even create a binary operator in R that can be used directly in expressions instead of invocation of a function. Suppose we chose `r` as an internal character to use. Then we can have our function definition as

```

> "%r%" <- function(y,x) {
+     XX <- crossprod(x)          ##Construct X'X
+     XY <- t(x)%*%y              ## Construct X'y
+     beta.hat <- solve(XX)%*%XY ## Construct Estimate
+     return(beta.hat)           ## Return Estimate
+ }
> mv%r%cbind(1,nox)  ## Calculate with binary operator
      [,1]
10.41032016
nox -0.01457707

```

Any assignments occurring within the function are local and temporary. This means that once the function has been left those variables and assignments, aside from what is returned are lost. Thus, a call like `nox <- Mean(nox)` is perfectly legitimate (but you will overwrite the vector `nox` with a scalar).

## 11. FINAL THOUGHTS

The discussion here has been brief and does not truly do justice to everything that R encapsulates. R is a constantly evolving language with researchers continually adding updates and new packages with cutting edge methods. I encourage you to use R for your own research given the ability to incorporate Sweave and produce full replicable scientific documents from beginning to end. Moreover, explore R. Grab a dataset within any of the packages discussed here and experiment with the various options and calls. You will be amazed at the scope of the commands.

## REFERENCES

- Adler, D. & Murdoch, D. (2011), *rgl: 3D visualization device system (OpenGL)*. R package version 0.92.798.  
**URL:** <http://CRAN.R-project.org/package=rgl>
- Bivand, R. (2011), *spdep: Spatial dependence: weighting schemes, statistics and models*. R package version 0.5-37.  
**URL:** <http://CRAN.R-project.org/package=spdep>
- Croissant, Y. (2011), *Ecdat: Data sets for econometrics*. R package version 0.1-6.1.  
**URL:** <http://CRAN.R-project.org/package=Ecdat>
- Csardi, G. & Nepusz, T. (2006), ‘The igraph software package for complex network research’, *InterJournal Complex Systems*, 1695.  
**URL:** <http://igraph.sf.net>
- Durlauf, S. N. & Johnson, P. (1995), ‘Multiple regimes and cross-country growth behavior’, *Journal of Applied Econometrics* **10**, 365–384.
- Gilley, O. W. & Pace, R. K. (1996), ‘On the Harrison and Rubinfeld data’, *Journal of Environmental Economics and Management* **31**, 403–405.
- Harrison, D. & Rubinfeld, D. L. (1978), ‘Hedonic housing prices and the demand for clean air’, *Journal of Environmental Economics and Management* **5**, 81–102.
- Kleiber, C. & Zeileis, A. (2008), *Applied Econometrics with R*, Springer-Verlag, New York.
- Racine, J. S. & Hyndman, R. (2002), ‘Using R to teach econometrics’, *Journal of Applied Econometrics* **17**, 175–189.
- Ruckdeschel, P., Kohl, M., Stabla, T. & Camphausen, F. (2006), ‘S4 classes for distributions’, *R News* **6**(2), 2–6.  
**URL:** <http://www.uni-bayreuth.de/departments/math/org/mathe7/DISTR/distr.pdf>
- Sarkar, D. (2008), *Lattice: Multivariate Data Visualization with R*, Springer, New York.
- Sun, C. (2011), *erer: Empirical Research in Economics with R*. R package version 1.0.  
**URL:** <http://CRAN.R-project.org/package=erer>
- Wheeler, B. (2009), *SuppDists: Supplementary distributions*. R package version 1.1-8.  
**URL:** <http://CRAN.R-project.org/package=SuppDists>