Econometrics in R

Grant V. Farnsworth*

October 26, 2008

*This paper was originally written as part of a teaching assistantship and has subsequently become a personal reference. I learned most of this stuff by trial and error, so it may contain inefficiencies, inaccuracies, or incomplete explanations. If you find something here suboptimal or have suggestions, please let me know. Until at least 2009 I can be contacted at g-farnsworth@kellogg.northwestern.edu.

Contents

1 Introductory Comments

1.1 What is R?

R is an implementation of the object-oriented mathematical programming language S. It is developed by statisticians around the world and is free software, released under the GNU General Public License. Syntactically and functionally it is very similar (if not identical) to S+, the popular statistics package.

1.2 How is R Better Than Other Packages?

R is much more more flexible than most software used by econometricians because it is a modern mathematical programming language, not just a program that does regressions and tests. This means our analysis need not be restricted to the functions included in the default package. There is an extensive and constantly expanding collection of libraries online for use in many disciplines. As researchers develop new algorithms and processes, the corresponding libraries get posted on the R website. In this sense R is always at the forefront of statistical knowledge. Because of the ease and flexibility of programming in R it is easy to extend.

The S language is the *de facto* standard for statistical science. Reading the statistical literature, we find that examples and even pseudo-code are written in R-compatible syntax. Since most users have a statistical background, the jargon used by R experts sometimes differs from what an econometrician (especially a beginning econometrician) may expect. A primary purpose of this document is to eliminate this language barrier and allow the econometrician to tap into the work of these innovative statisticians.

Code written for R can be run on many computational platforms with or without a graphical user interface, and R comes standard with some of the most flexible and powerful graphics routines available anywhere.

And of course, R is completely free for any use.

1.3 Obtaining R

The R installation program can be downloaded free of charge from `http://www.r-project.org`. Because R is a programming language and not just an econometrics program, most of the functions we will be interested in are available through libraries (sometimes called packages) obtained from the R website. To obtain a library that does not come with the standard installation follow the *CRAN* link on the above website. Under *contrib* you will find is a list of compressed libraries ready for download. Click on the one you need and save it somewhere you can find it later. If you are using a gui, start R and click *install package from local directory* under the *package* menu. Then select the file that you downloaded. Now the package will be available for use in the future. If you are using R under linux, install new libraries by issuing the following command at the command prompt: "`R CMD INSTALL` *packagename*"

Alternately you can download and install packages at once from inside R by issuing a command like

```
> install.packages(c("car","systemfit"),repo="http://cran.stat.ucla.edu",dep=TRUE)
```

which installs the *car* and *systemfit* libraries. The `repo` parameter is usually auto-configured, so there is normally no need to specify it. The `dependencies` or `dep` parameter indicates that R should download packages that these depend on as well, and is recommended. Note: you must have administrator (or root) privileges to your computer to install the program and packages.

Contributed Packages Mentioned in this Paper and Why

(indicates package is included by default)*

adapt	Multivariate numerical integration
car	Regression tests and robust standard errors
DBI	Interact with databases
dse1	State space models, Kalman filtration, and Vector ARMA
filehash	Use hard disk instead of RAM for large datasets
fSeries	Garch models with nontrivial mean equations
fracdiff	Fractionally integrated ARIMA models
foreign*	Loading and saving data from other programs
ggplot2	Graphics and plotting
graphics*	Contour graphs and arrows for plots
grid	Graphics and plotting
Hmisc	LaTeX export
lattice	An alternate type of contour plot and other graphics
lmtest	Breusch-Pagan and Breusch-Godfrey tests
MASS*	Robust regression, ordered logit/probit
Matrix	Matrix norms and other matrix algebra stuff
MCMCpack	Inverse gamma distribution
MNP	Multinomial probit via MCMC
nlme*	Nonlinear fixed and random effects models
nls*	Nonlinear least squares
nnet	Multinomial logit/probit
quantreg	Quantile Regressions
R.matlab	Read matlab data files
RSQLite	Interact with SQL databases
sandwich (and zoo)	Heteroskedasticity and autocorrelation robust covariance
sem	Two stage least squares
survival*	Tobit and censored regression
systemfit	SUR and 2SLS on systems of equations
ts*	Time series manipulation functions
tseries	Garch, ARIMA, and other time series functions
VAR	Vector autoregressions
xtable	Alternative LaTeX export
zoo	required in order to have the sandwich package

From time to time we can get updates of the installed packages by running `update.packages()`.

1.4 Using R Interactively and Writing Scripts

We can interact directly with R through its command prompt. Under windows the prompt and what we type are in red and the output it returns is blue–although you can control the font colors though "GUI preferences" in the edit menu. Pressing the up arrow will generally cycle through commands from the history. Notice that R is case sensitive and that every function call has parentheses at the end. Instead of issuing commands directly we can load script files that we have previously written, which may include new function definitions.

Script files generally have the extension ".R". These files contain commands as you would enter them at the prompt, and they are recommended for any project of more than a few lines. In order to load a script file named "mcmc.R" we would use the command

```
> source("mcmc.R")
```

One way to run R is to have a script file open in an external text editor and run periodically from the R window. Commands executed from a script file may not print as much output to the screen as they do when run interactively. If we want interactive-level verbosity, we can use the `echo` argument

```
> source("mcmc.R",echo=TRUE)
```

If no path is specified to the script file, R assumes that the file is located in the current working directory. The working directory can be viewed or changed via R commands

```
> getwd()
[1] "/home/gvfarns/r"
> setwd("/home/gvfarns")
> getwd()
[1] "/home/gvfarns"
```

or under windows using the menu item *change working directory*. Also note that when using older versions of R under windows the slashes must be replaced with double backslashes.

```
> getwd()
[1] "C:\\Program Files\\R\\rw1051\\bin"
> setwd("C:\\Program Files\\R\\scripts")
> getwd()
[1] "C:\\Program Files\\R\\scripts"
```

We can also run R in batch (noninteractive) mode under linux by issuing the command:"R CMD BATCH *scriptname*.R" The output will be saved in a file named *scriptname*.Rout. Batch mode is also available under windows using Rcmd.exe instead of Rgui.exe.

Since every command we will use is a function that is stored in one of the libraries, we will often have to load libraries before working. Many of the common functions are in the library *base*, which is loaded by default. For access to any other function, however, we have to load the appropriate library.

```
> library(foreign)
```

will load the library that contains the functions for reading and writing data that is formatted for other programs, such as SAS and Stata. Alternately (under windows), we can pull down the *package* menu and select *library*

1.5 Getting Help

There are several methods of obtaining help in R

```
> ?qt
> help(qt)
> help.start()
> help.search("covariance")
```

Preceding the command with a question mark or giving it as an argument to `help()` gives a description of its usage and functionality. The `help.start()` function brings up a menu of help options and `help.search()` searches the help files for the word or phrase given as an argument. Many times, though, the best help available can be found by a search online. Remember as you search that the syntax and functionality of R is almost identical to that of the proprietary statistical package S+.

The help tools above only search through the R functions that belong to packages on your computer. A large percentage of R questions I hear are of the form "Does R have a function to do..." Users do not know if functionality exists because the corresponding package is not installed on their computer. To search the R website for functions and references, use

```
> RSiteSearch("Kalman Filter")
```

The results from the search should appear in your web browser.

2 Working with Data

2.1 Basic Data Manipulation

R allows you to create many types of data storage objects, such as numbers, vectors, matrices, strings, and dataframes. The command `ls()` gives a list of all data objects currently available. The command `rm()` removes the data object given it as an argument. We can determine the type of an object using the command `typeof()` or its class type (which is often more informative) using `class()`.

Entering the name of the object typically echos its data to the screen. In fact, a function is just another data member in R. We can see the function's code by typing its name without parenthesis.

The command for creating and/or assigning a value to a data object is the less-than sign followed by the minus sign.

```
> g <- 7.5
```

creates a numeric object called g, which contains the value 7.5. True vectors in R (as opposed to one dimensional matrices) are treated as COLUMN vectors, when the distinction needs to be made.

```
> f <- c(7.5,6,5)
> F <- t(f)
```

uses the `c()` (concatenate) command to create a vector with values 7.5, 6, and 5. `c()` is a generic function that can be used on multiple types of data. The `t()` command transposes `f` to create a 1x2 matrix—because "vectors" are always column vectors. The two data objects `f` and `F` are separate because of the case sensitivity of R. The command `cbind()` concatenates the objects given it side by side: into an array if they are vectors, and into a single dataframe if they are columns of named data.

```
> dat <- cbind(c(7.5,6,5),c(1,2,3))
```

Similarly, `rbind()` concatenates objects by rows—one above the other—and assumes that vectors given it are ROW vectors (see 2.3.1).

Notice that if we were concatenating strings instead of numeric values, we would have to put the strings in quotes. Alternately, we could use the `Cs()` command from the *Hmisc* library, which eliminates the need for quotes.

```
> Cs(Hey,you,guys)
[1] "Hey"  "you"  "guys"
```

Elements in vectors and similar data types are indexed using square brackets. R uses one-based indexing.

```
> f
 [1] 7.5 6.0 5.0
> f[2]
 [1] 6
```

Notice that for multidimensional data types, such as matrices and dataframes, leaving an index blank refers to the whole column or row corresponding to that index. Thus if `foo` is a 4x5 array of numbers,

```
> foo
```

will print the whole array to the screen,

```
> foo[1,]
```

will print the first row,

```
> foo[,3]
```

will print the third column, etc. We can get summary statistics on the data in `goo` using the `summary()` and we can determine its dimensionality using the `NROW()`, and `NCOL()` commands. More generally, we can use the `dim()` command to know the dimensions of many R objects.

If we wish to extract or print only certain rows or columns, we can use the concatenation operator.

```
> oddfoo <- foo[,c(1,3,5)]
```

makes a 4x3 array out of columns 1,3, and 5 of foo and saves it in oddfoo. By prepending the subtraction operator, we can remove certain columns

```
> nooddfoo <- foo[,-c(1,3,5)]
```

makes a 4x2 array out of columns 2 and 4 of foo (i.e., it removes columns 1,3, and 5). We can also use comparison operators to extract certain columns or rows.

```
> smallfoo <- foo[foo[,1]<1,]
```

compares each entry in the first column of foo to one and inserts the row corresponding to each match into smallfoo. We can also reorder data. If `wealth` is a dataframe with columns `year`, `gdp`, and `gnp`, we could sort the data by year using `order()` or extract a period of years using the colon operator

```
> wealth <- wealth[order(wealth$year),]
> firstten <- wealth[1:10,]
> eighty <- wealth[wealth$year==1980,]
```

This sorts by year and puts the first ten years of data in firstten. All rows from year 1980 are stored in eighty (notice the double equals sign).

Using double instead of single brackets for indexing changes the behavior slightly. Basically it doesn't allow referencing multiple objects using a vector of indices, as the single bracket case does. For example,

```
> w[[1:10]]
```

does not return a vector of the first ten elements of `w`, as it would in the single bracket case. Also, it strips off attributes and types. If the variable is a list, indexing it with single brackets yields a list containing the data, double brackets return the (vector of) data itself. Most often, when getting data out of lists, double brackets are wanted, otherwise single brackets are more common.

Occasionally we have data in the incorrect form (i.e., as a dataframe when we would prefer to have a matrix). In this case we can use the `as.` functionality. If all the values in `goo` are numeric, we could put them into a matrix named `mgoo` with the command

```
> mgoo <- as.matrix(goo)
```

Other data manipulation operations can be found in the standard R manual and online. There are a lot of them.

2.2 Caveat: Math Operations and the Recycling Rule

Mathematical operations such as addition and multiplication operate *elementwise* by default. The matrix algebra operations are generally surrounded by % (see section 8). The danger here happens when one tries to do math using certain objects of different sizes. Instead of halting and issuing an error as one might expect, R uses a *recycling rule* to decide how to do the math—that is, it repeats the values in the smaller data object. For example,

```
> a<-c(1,3,5,7)
> b<-c(2,8)
> a+b
[1]  3 11  7 15
```

Only if the dimensions are not multiples of each other does R return a warning (although it still does the computation)

```
> a<-c(2,4,16,7)
> b<-c(2,8,9)
> a+b
```

```
[1]  4 12 25  9
Warning message:
longer object length
        is not a multiple of shorter object length in: a + b
```

At first the recycling rule may seem like a dumb idea (and it can cause error if the programmer is not careful) but this is what makes operations like scalar addition and scalar multiplication of vectors and matrices (as well as vector-to-matrix addition) possible. One just needs to be careful about dimensionality when doing elementwise math in R.

Notice that although R recycles vectors when added to other vectors or data types, it does not recycle when adding, for example, two matrices. Adding matrices or arrays of different dimensions to each other produces an error.

2.3 Important Data Types

2.3.1 Vectors

The most fundamental numeric data type in R is an unnamed vector. A scalar is, in fact, a 1-vector. Vectors are more abstract than one dimensional matrices because they do not contain information about whether they are row or column vectors—although when the distinction must be made, R usually assumes that vectors are columns.

The vector abstraction away from rows/columns is a common source of confusion in R by people familiar with matrix oriented languages, such as matlab. The confusion associated with this abstraction can be shown by an example

```
a<-c(1,2,3)
b<-c(4,6,8)
```

Now we can make a matrix by stacking vertically or horizontally and R assumes that the vectors are either rows or columns, respectively.

```
> cbind(a,b)
     a b
[1,] 1 4
[2,] 2 6
[3,] 3 8
> rbind(a,b)
  [,1] [,2] [,3]
a    1    2    3
b    4    6    8
```

One function assumed that these were column vectors and the other that they were row vectors. The take home lesson is that a vector is not a one dimensional matrix, so don't expect them to work as they do in a linear algebra world. To convert a vector to a 1xN matrix for use in linear algebra-type operations (column vector) us `as.matrix()`.

Note that `t()` returns a matrix, so that the object `t(t(a))` is not the same as `a`.

2.3.2 Arrays, Matrices

In R, homogeneous (all elements are of the same type) multivariate data may be stored as an array or a matrix. A matrix is a two-dimensional object, whereas an array may be of many dimensions. These data types may or may not have special attributes giving names to columns or rows (although one cannot reference a column using the `$` operator as with dataframes) but can hold only numeric data. Note that one cannot make a matrix, array, or vector of two different types of data (numeric and character, for example). Either they will be coerced into the same type or an error will occur.

2.3.3 Dataframes

Most econometric data will be in the form of a dataframe. A dataframe is a collection of vectors (we think of them as columns) containing data, which need not all be of the same type, but each column must have the same number of elements. Each column has a title by which the whole vector may be addressed. If `goo` is a 3x4 data frame with titles `age`, `gender`, `education`, and `salary`, then we can print the `salary` column with the command

```
> goo$salary
```

or view the names of the columns in goo

```
> names(goo)
```

Most mathematical operations affect multidimensional data elementwise (unlike some mathematical languages, such as matlab). From the previous example,

```
> salarysq <- (goo$salary)^2
```

creates a dataframe with one column entitled `salary` with entries equal to the square of the corresponding entries in `goo$salary`. Output from actions can also be saved in the original variable, for example,

```
> salarysq <- sqrt(salarysq)
```

replaces each of the entries in salarysq with its square root.

```
> goo$lnsalary <- log(salarysq)
```

adds a column named lnsalary to `goo`, containing the log of the salary.

2.3.4 Lists

A list is more general than a dataframe. It is essentially a bunch of data objects bound together, optionally with a name given to each. These data objects may be scalars, strings, dataframes, or any other type. Functions that return many elements of data (like `summary()`) generally bind the returned data together as a list, since functions return only one data object. As with dataframes, we can see what objects are in a list (by name if they have them) using the `names()` command and refer to them either by name (if existent) using the `$` symbol or by number using brackets. **Remember that referencing a member of a list is generally done using double, not single, brackets (see section 2.1).** Sometimes we would like to simplify a list into a vector. For example, the function `strsplit()` returns a list containing substrings of its argument. In order to make them into a vector of strings, we must change the list to a vector using `unlist()`. Lists sometimes get annoying, so `unlist()` is a surprisingly useful function.

2.3.5 S3 Classes

Many functions return an object containing many types of data, like a list, but would like R to know something about what type of object it is. A list with an associated "class" attribute designating what type of list it is is an S3 class. If the class is passed as an argument, R will first search for an appropriately named function. If `x` is of class `foo` and you print it with

```
> print(x)
```

the `print()` routine first searches for a function named `print.foo()` and will use that if it exists. Otherwise it will use the generic `print.default()`. For example, if `x` is the output of a call to `lm()`, then

```
> print(x)
```

will call `print.lm(x)`, which prints regression output in a meaningful and aesthetically pleasing manner.

S3 lists are quite simple to use. The are really just lists with an extra attribute. We can create them either using the `class()` function or just adding the class attribute after creation of a list.

```
> h <- list(a=rnorm(3),b="This shouldn't print")
> class(h) <- "myclass"
> print.myclass<-function(x){cat("A is:",x$a,"\n")}
> print(h)
A is: -0.710968 -1.611896 0.6219214
```

If we were to call `print()` without assigning the class, we would get a different result.

Many R packages include extensions to common generic functions like `print()`, `summary()`, and `plot()` which operate on the particular classes produced by that package. The ability of R to choose a function to execute depending on the class of the data passed to it makes interacting with new classes very convenient. On the other hand, many extensions have options specific to them, so we must read the help file on that particular extension to know how to best use it. For example, we should read up on the regression print routine using

```
> ?summary.lm
```

instead of

```
> ?summary
```

2.3.6 S4 Classes

S4 classes are a recent addition to R. They generically hold data and functions, just like S3 classes, but have some technical advantages, which transcend the scope of this document. For our purposes, the most important difference between an S3 and S4 class is that attributes of the latter are referenced using `@` instead of `$` and it can only be created using the `new()` command.

```
> g <- garchFit(~arma(0,1)+garch(2,3),y)
> fitvalues <- g@fit
```

2.4 Working with Dates

The standard way of storing dates internally in R is as an object of class *Date*. This allows for such things as subtraction of one date from another yielding the number of days between them. To convert data to dates, we use `as.Date()`. This function takes as input a character string and a format. If the given vector of dates is stored as a numeric format (like "20050627") it should be converted to a string using `as.character()` first. The format argument informs the code what part of the string corresponds to what part of the date. Four digit year is `%Y`, two digit year is `%y`, numeric month is `%m`, alphabetic (abbreviated) month is `%b`, alphabetic (full) month is `%B`, day is `%d`. For other codes, see the help files on `strptime`. For example, if `d` is a vector of dates formatted like "2005-Jun-27", we could use

```
> g<-as.Date(d,format="%Y-%b-%d")
```

Internally, *Date* objects are numeric quantities, so they don't take up very much memory.

We can perform the reverse operation of `as.Date()`—formatting or extracting parts of a *Date* object—using `format()`. For example, given a column of numbers like "20040421", we can extract a character string representing the year using

```
> year<-format(as.Date(as.character(v$DATE),format="%Y%m%d"),format="%Y")
```

Although we can extract day of the week information in string or numeric form using `format()`, a simpler interface is available using the `weekdays()` function.

```
> mydates<-as.Date(c("19900307","19900308"),format="%Y%m%d")
> weekdays(mydates)
[1] "Wednesday" "Thursday"
```

Notice that in order to get a valid date, we need to have the year, month, and day. Suppose we were using monthly data, we would need to assign each data point to, for example, the first day of the month. In the case of a numeric format such as "041985" where the first two digits represent the month and the last four the year, we can simply add 10,000,000 to the number, convert to string, and use the format "%d%m%Y". In the case of a string date such as "April 1985" we can use

```
> as.Date(paste("1 ",v$DATE),format="%d %B %Y")
```

Notice that in order for paste to work correctly `v$DATE` must be a vector of character strings. Some read methods automatically convert strings to factors by default, which we can rectify by passing the `as.is=T` keyword to the read method or converting back using `as.character()`.

2.5 Merging Dataframes

If we have two dataframes covering the same time or observations but not completely aligned, we can merge the two dataframes using `merge()`. Either we can specify which column to use for aligning the data, or `merge()` will try to identify column names in common between the two.

For example, if `B` is a data frame of bond data prices over a certain period of time and had a column named `date` containing the dates of the observations and `E` was a similar dataframe of equity prices over about the same time period, we could merge them into one dataframe using

```
> OUT<-merge(B,E)
```

If the date column was named `date` in B but `day` in E, the command would instead be

```
> OUT<-merge(B,E,by.x="date",by.y="day")
```

Notice that by default `merge()` includes only rows in which data are present in both B and E. To put `NA` values in the empty spots instead of omitting the rows we include the `all=T` keyword. We can also specify whether to include `NA` values only from one or the other using the `all.x` and `all.y` keywords.

2.6 Opening a Data File

R is able to read data from many formats. The most common format is a text file with data separated into columns and with a header above each column describing the data. If `blah.dat` is a text file of this type and is located on the windows desktop we could read it using the command

```
> mydata <- read.table("C:/WINDOWS/Desktop/blah.dat",header=TRUE)
```

Now `mydata` is a dataframe with named columns, ready for analysis. Note that R assumes that there are no labels on the columns, and gives them default values, if you omit the `header=TRUE` argument. Now let's suppose that instead of `blah.dat` we have `blah.dta`, a stata file.

```
> library(foreign)
> mydata <- read.dta("C:/WINDOWS/Desktop/blah.dta")
```

Stata files automatically have headers.

Another data format we may read is .csv comma-delimited files (such as those exported by spreadsheets). These files are very similar to those mentioned above, but use punctuation to delimit columns and rows. Instead of `read.table()`, we use `read.csv()`. Fixed width files can be read using `read.fwf()`.

Matlab (`.mat`) files can be read using `readMat()` from the *R.matlab* package. The function `writeMat()` from the same package writes matlab data files.

2.7 Working With Very Large Data Files

R objects can be as big as our physical computer memory (and operating system) will allow, but it is not designed for very large datasets. This means that extremely large objects can slow everything down tremendously and suck up RAM greedily. The `read.table()` family of routines assume that we are not working with very large data sets and so are not careful to conserve on memory[1]. They load everything at once and probably make at least one copy of it. A better way to work with huge datasets is to read the file a line (or group of lines) at a time. We do this using connections. A connection is an R object that points to a file or other input/output stream. Each time we read from a connection the location in the file from which we read moves forward.

Before we can use a connection, we must create it using `file()` if our data source is a file or `url()` for an online source (there are other types of connections too). Then we use `open()` to open it. Now we can read one or many lines of text using `readLines()`, read fields of data using `scan()`, or write data using `cat()` or one of the `write.table()` family of routines. When we are done we close using `close()`.

2.7.1 Reading fields of data using scan()

Reading fields of data from a huge file is a very common task, so we give it special attention. The most important argument to `scan()` is `what`, which specifies what type of data to read in. If the file contains columns of data, `what` should be a list, with each member of the list representing a column of data. For example, if the file contains a name followed by a comma separator and an age, we could read a single line using

```
>  a <- scan(f,what=list(name="",age=0),sep=",",nlines=1)
```

where `f` is an open connection. Now `a` is a list with fields `name` and `age`. Example 13.4 shows how to read from a large data file.

If we try to scan when the connection has reached the end of the file, `scan()` returns an empty list. We can check it using `length()` in order to terminate our loop.

Frequently we know how many fields are in each line and we want to make sure the `scan()` gets all of them, filling the missing ones with *NA*. To do this we specify `fill=T`. Notice that in many cases `scan()` will fill empty fields with *NA* anyway.

Unfortunately scan returns an error if it tries to read a line and the data it finds is not what it is expecting. For example, if the string `"UNK"` appeared under the age column in the above example, we would have an error. If there are only a few possible exceptions, they can be passed to `scan()` as `na.strings`. Otherwise we need to read the data in as strings and then convert to numeric or other types using `as.numeric()` or some other tool.

Notice that reading one line at a time is not the fastest way to do things. R can comfortably read 100, 1000, or more lines at a time. Increasing how many lines are read per iteration could speed up large reads considerably. With large files, we could read lines 1000 at a time, transform them, and then write 1000 at a time to another open connection, thereby keep system memory free.

If all of the data is of the same type and belong in the same object (a 2000x2000 numeric matrix, for example) we can use `scan()` without including the `nlines` argument and get tremendously faster reads. The resulting vector would need only to be converted to type `matrix`.

2.7.2 Utilizing Unix Tools

If you are using R on a linux/unix machine[2] you can use various unix utilities (like `grep` and `awk`) to read only the colunms and rows of your file that you want. The utility `grep` trims out rows that do or do not contain a specificed pattern. The programming language `awk` is a record oriented tool that can pull out and manipulate columns as well as rows based on a number of criteria.

[1] According to the help file for `read.table()` you can improve memory usage by informing `read.table()` of the number of rows using the `nrows` parameter. On unix/linux you can obtain the number of rows in a text file using "wc -l".

[2] Thanks to Wayne Folta for these suggestions

Some of these tools are useful within R as well. For example, we can preallocate our dataframes according to the number of records (rows) we will be reading in. For example to know how large a dataframe to allocate for the calls in the above example, we could use

```
> howmany <- as.numeric(system ("grep -c ',C,' file.dat"))
```

Since allocating and reallocating memory is one of the time consuming parts of the `scan()` loop, this can save a lot of time and troubles this way. To just determine the number of rows, we can use the utility `wc`.

```
> totalrows <- as.numeric(strsplit(system("wc -l Week.txt",intern=T),split=" ")[[1]][1])
```

Here `system()` returns the number of rows, but with the file name as well, `strsplit()` breaks the output into words, and we then convert the first word to a number.

The bottom line is that we should use the right tool for the right job. Unix utilities like `grep`, `awk`, and `wc` can be fast and dirty ways to save a lot of work in R.

2.7.3 Using Disk instead of RAM

Unfortunately, R uses only system RAM by default. So if the dataset we are loading is very large it is likely that our operating system will not be able to allocate a large enough chunck of system RAM for it, resulting in termination of our program with a message that R could not allocate a vector of that size. Although R has no built in disk cache system, there is a package called *filehash* that allows us to store our variables on disk instead of in system RAM. Clearly this will be slower, but at least our programs will run as long as we have sufficient disk space and our file size does not exceed the limits of our operating system. And sometimes that makes all the difference.

Instead of reading a file into memory, we read into a database and load that database as an environment

```
>  dumpDF(read.table("large.txt", header=T), dbName="mydb")
>  myenv<-db2env(db="mydb")
```

Now we can operate on the data within its environment using `with()`

```
>  with(mydb, z<-y+x )
```

Actually I haven't spent much time with this package so I'm not familiar with its nuances, but it seems very promising[3].

2.7.4 Using RSQLite

Many times the best way to work with large datasets is to store them in SQL databases and then just query the stuff you need using mySQL or SQLite from within R. This functionality is available using the RSQLite and DBI libraries.

2.8 Issuing System Commands—Directory Listing

Sometimes we want to issue a command to the operating system from inside of R. For example, under unix we may want to get a list of the files in the current directory that begin with the letter x. We could execute this command using

```
> system("ls x*")
xaa  xab  xac  xad  xae
```

If we want to save the output of the command as an R object, we use the keyword `intern`

```
> files <- system("ls x*",intern=T)
```

[3]For more information see http://yusung.blogspot.com/2007/09/dealing-with-large-data-set-in-r.html

2.9 Reading Data From the Clipboard

When importing from other applications, such as spreadsheets and database managers, the quickest way to get the data is to highlight it in the other application, copy it to the desktop clipboard, and then read the data from the clipboard. R treats the clipboard like a file, so we use the standard *read.table()* command

```
> indata <- read.table("clipboard")
```

2.10 Editing Data Directly

R has a built-in spreadsheet-like interface for editing data. It's not very advanced, but it works in a pinch. Suppose `a` is a dataframe, we can edit it in place using

```
> data.entry(a)
```

Any changes we make (including changing the type of data contained in a column) will be reflected in a immediately. If we want to save the changed data to a different variable, we could have used

```
> b <- de(a)
```

Notice that both `de()` and `data.entry()` return a variable of type *list*. If what we wanted was a dataframe, for example, we need to convert it back after editing.

The function `edit()` works like `de()` but for many different data types. In practice, it calls either `de()` or the system default text editor (which is set using `options()`).

A similar useful function is `fix()`, which edits an R object in place. `fix()` operates on any kind of data: dataframes are opened with `de()` and functions are opened with a text editor. This can be useful for quick edits either to data or code.

3 Cross Sectional Regression

3.1 Ordinary Least Squares

Let's consider the simplest case. Suppose we have a data frame called `byu` containing columns for `age`, `salary`, and `exper`. We want to regress various forms of `age` and `exper` on `salary`. A simple linear regression might be

```
> lm(byu$salary ~ byu$age + byu$exper)
```

or alternately:

```
> lm(salary ~ age + exper,data=byu)
```

as a third alternative, we could "attach" the dataframe, which makes its columns available as regular variables

```
> attach(byu)
> lm(salary ~ age + exper)
```

Notice the syntax of the model argument (using the tilde). The above command would correspond to the linear model

$$salary = \beta_0 + \beta_1 age + \beta_2 exper + \epsilon \tag{1}$$

Using `lm()` results in an abbreviated summary being sent to the screen, giving only the β coefficient estimates. For more exhaustive analysis, we can save the results in as a data member or "fitted model"

```
> result <- lm(salary ~ age + exper + age*exper,data=byu)
> summary(result)
> myresid <- result$resid
> vcov(result)
```

The `summary()` command, run on raw data, such as `byu$age`, gives statistics, such as the mean and median (these are also available through their own functions, mean and median). When run on an ols object, summary gives important statistics about the regression, such as p-values and the R^2.

The residuals and several other pieces of data can also be extracted from result, for use in other computations. The variance-covariance matrix (of the beta coefficients) is accessible through the `vcov()` command.

Notice that more complex formulae are allowed, including interaction terms (specified by multiplying two data members) and functions such as `log()` and `sqrt()`. Unfortunately, in order to include a power term, such as `age` squared, we must either first compute the values, then run the regression, or use the `I()` operator, which forces computation of its argument before evaluation of the formula

```
> salary$agesq <- (salary$age)^2
> result <- lm(salary ~ age + agesq + log(exper) + age*log(exper),data=byu)
```

or

```
> result <- lm(salary ~ age + I(age^2) + log(exper) + age*log(exper),data=byu)
```

Notice that if we omit the `I()` operator and don't explicitly generate a power term variable (like `agesq`) then `lm()` will not behave as expected[4], but it will not give we an error or warning, so we must be careful.

In order to run a regression without an intercept, we simply specify the intercept explicitly, traditionally with a zero.

```
> result <- lm(smokes ~ 0 + male + female ,data=smokerdata)
```

3.2 Extracting Statistics from the Regression

The most important statistics and parameters of a regression are stored in the `lm` object or the `summary` object. Consider the smoking example above

```
> output <- summary(result)
> SSR <- deviance(result)
> LL <- logLik(result)
> DegreesOfFreedom <- result$df
> Yhat <- result$fitted.values
> Coef <- result$coefficients
> Resid <- result$residuals
> s <- output$sigma
> RSquared <- output$r.squared
> CovMatrix <- s^2*output$cov
> aic <- AIC(result)
```

Where `SSR` is the residual sum of squares, `LL` is the log likelihood statistic, `Yhat` is the vector of fitted values, `Resid` is the vector of residuals, `s` is the estimated standard deviation of the errors (assuming homoskedasticity), `CovMatrix` is the variance-covariance matrix of the coefficients (also available via `vcov()`), `aic` is the Akaike information criterion and other statistics are as named.

Note that the AIC criterion is define by R as

$$AIC = -2\log L(p) + 2p$$

where p is the number of estimated parameters and $L(p)$ is the likelihood. Some econometricians prefer to call AIC/N the information criterion. To obtain the Bayesian Information Criterion (or Schwartz Bayesian Criterion) we use AIC but specify a different "penalty" parameter as follows

```
> sbc <- AIC(result,k=log(NROW(smokerdata)))
```

which means

$$SBC = -2\log L(p) + p\log(N)$$

[4]Generally it just omits power terms whose first power is already included in the regression.

3.3 Heteroskedasticity and Friends

3.3.1 Breusch-Pagan Test for Heteroskedasticity

In order to test for the presence of heteroskedasticity, we can use the Breusch-Pagan test from the *lmtest* package. Alternately we can use the the `ncv.test()` function from the *car* package. They work pretty much the same way. After running the regression, we call the `bptest()` function with the fitted regression.

```
> unrestricted <- lm(z~x)
> bptest(unrestricted)

        Breusch-Pagan test

data:  unrestricted
BP = 44.5465, df = 1, p-value = 2.484e-11
```

This performs the "studentized" version of the test. In order to be consistent with some other software (including `ncv.test()`) we can specify `studentize=FALSE`.

3.3.2 Heteroskedasticity (Autocorrelation) Robust Covariance Matrix

In the presence of heteroskedasticity, the ols estimates remain unbiased, but the ols estimates of the variance of the beta coefficients are no longer correct. In order to compute the heteroskedasticity consistent covariance matrix[5] we use the `hccm()` function (from the *car* library) instead of `vcov()`. The diagonal entries are variances and off diagonals are covariance terms.

This functionality is also available via the `vcovHC()` command in the *sandwich* package. Also in that package is the heteroskedasticity and autocorrelation robust Newey-West estimator, available in the function `vcovHAC()` or the function `NeweyWest()`.

3.4 Linear Hypothesis Testing (Wald and F)

The *car* package provides a function that automatically performs linear hypothesis tests. It does either an F or a Wald test using either the regular or adjusted covariance matrix, depending on our specifications. In order to test hypotheses, we must construct a hypothesis matrix and a right hand side vector. For example, if we have a model with five parameters, including the intercept and we want to test against

$$H_0 : \beta_0 = 0, \beta_3 + \beta_4 = 1$$

The hypothesis matrix and right hand side vector would be

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \beta = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

and we could implement this as follows

```
> unrestricted <- lm(y~x1+x2+x3+x4)
> rhs <- c(0,1)
> hm <- rbind(c(1,0,0,0,0),c(0,0,1,1,0))
> linear.hypothesis(unrestricted,hm,rhs)
```

Notice that if `unrestricted` is an *lm* object, an F test is performed by default, if it is a *glm* object, a Wald χ^2 test is done instead. The type of test can be modified through the `type` argument.

Also, if we want to perform the test using heteroskedasticity or autocorrelation robust standard errors, we can either specify `white.adjust=TRUE` to use white standard errors, or we can supply our own covariance matrix using the `vcov` parameter. For example, if we had wished to use the Newey-West corrected covariance matrix above, we could have specified

[5] obtaining the White standard errors, or rather, their squares.

```
> linear.hypothesis(unrestricted,hm,rhs,vcov=NeweyWest(unrestricted))
```

See the section on heteroskedasticity robust covariance matrices for information about the `NeweyWest()` function. We should remember that the specification `white.adjust=TRUE` corrects for heteroskedasticity using an improvement to the white estimator. To use the classic white estimator, we can specify `white.adjust="hc0"`.

3.5 Weighted and Generalized Least Squares

You can do weighted least squares by passing a vector containing the weights to `lm()`.

```
> result <- lm(smokes ~ 0 + male + female ,data=smokerdata,weights=myweights)
```

Generalized least squares is available through the `lm.gls()` command in the *MASS* library. It takes a formula, weighting matrix, and (optionally) a dataframe from which to get the data as arguments.

The `glm()` command provides access to a plethora of other advanced linear regression methods. See the help file for more details.

3.6 Models With Factors/Groups

There is a separate datatype for qualitative factors in R. When a variable included in a regression is of type factor, the requisite dummy variables are automatically created. For example, if we wanted to regress the adoption of personal computers (pc) on the number of employees in the firm (emple) and include a dummy for each state (where `state` is a vector of two letter abbreviations), we could simply run the regression

```
> summary(lm(pc~emple+state))

Call:
lm(formula = pc ~ emple + state)

Residuals:
    Min      1Q  Median      3Q     Max
-1.7543 -0.5505  0.3512  0.4272  0.5904

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  5.572e-01  6.769e-02   8.232   <2e-16 ***
emple        1.459e-04  1.083e-05  13.475   <2e-16 ***
stateAL     -4.774e-03  7.382e-02  -0.065    0.948
stateAR      2.249e-02  8.004e-02   0.281    0.779
stateAZ     -7.023e-02  7.580e-02  -0.926    0.354
stateDE      1.521e-01  1.107e-01   1.375    0.169

 ...

stateFL     -4.573e-02  7.136e-02  -0.641    0.522
stateWY      1.200e-01  1.041e-01   1.153    0.249
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.4877 on 9948 degrees of freedom
Multiple R-Squared: 0.02451,    Adjusted R-squared: 0.01951
F-statistic: 4.902 on 51 and 9948 DF,  p-value: < 2.2e-16
```

The three dots indicate that some of the coefficients have been removed for the sake of brevity.

In order to convert data (either of type string or numeric) to a factor, simply use the `factor()` command. It can even be used inside the regression. For example, if we wanted to do the same regression, but by a numeric code specifying an area, we could use the command

```
> myout <- lm(pc~emple+factor(naics6))
```

which converts naics6 into a factor, generates the appropriate dummies, and runs a standard regression.

4 Special Regressions

4.1 Fixed/Random Effects Models

Warning: The definitions of fixed and random effects models are not standardized across disciplines. I describe fixed and random effects estimation as these terms are generally used by econometricians. The terms "fixed" and "random" have historical roots and are econometrically misleading.

Within the context of economics, fixed and random effects estimators are panel data models that account for cross sectional variation in the intercept. Letting i denote the cross sectional index (or the one by which data is grouped) and t the time index (or the index that varies within a group), a standard **fixed effects model** can be written

$$y_{it} = \alpha + u_i + \beta X_{it} + \epsilon_{it}. \tag{2}$$

Essentially, each individual has a different time-invariant intercept ($\alpha + u_i$). Usually we are interested in β but not any of the u_i. A **random effects model** has the same mean equation, but imposes the additional restriction that the individual specific effect is uncorrelated with the explanatory variables X_{it}. That is, $\mathbb{E}[u_i X_{it}] = 0$. Econometrically this is a more restrictive version of the fixed effects estimator (which allows for arbitrary correlation between the "effect" and exogenous variables). One should not let the unfortunate nomenclature confuse the relationship between these models.

4.1.1 Fixed Effects

A simple way to do a fixed effects estimation, particularly if the cross sectional dimension is not large, is to include a dummy for each individual—that is, make the cross sectional index a factor. If `index` identifies the individuals in the sample, then

```
> lm(y~factor(index)+x)
```

will do a fixed effects estimation and will report the correct standard errors on β. Unfortunately, in cases where there are many individuals in the sample and we are not interested in the value of their fixed effects, the `lm()` results are awkward to deal with and the estimation of a large number of u_i coefficients could render the problem numerically intractable.

A more common way to estimate fixed effects models is to remove the fixed effect by time demeaning each variable (the so called *within* estimator). Then equation (2) becomes

$$(y_{it} - \bar{y}_i) = \alpha + \beta(X_{it} - \bar{X}_i) + \zeta_{it}. \tag{3}$$

Most econometric packages (for example, stata's `xtreg`) use this method when doing fixed effects estimation. Using R, one can manually time demean the independent and dependent variables. If `d` is a dataframe containing `year`, `firm`, `profit`, and `predictor` and we wish to time demean each firm's profit and predictor, we could run something like

```
> g<-d
> for (i in unique(d$firm)){
+   timemean<-mean(d[d$firm==i,])
+   g$profit[d$firm==i]<-d$profit[d$firm==i]-timemean["profit"]
+   g$predictor[d$firm==i]<-d$predictor[d$firm==i]-timemean["predictor"]
+ }
> output<-lm(profit~predictor,data=g)
```

Notice that the standard errors reported by this regression will be biased downward. The $\hat{\sigma}^2$ reported by `lm()` is computed using

$$\hat{\sigma}^2 = \frac{SSR}{NT - K}$$

whereas the true standard errors in this fixed effects regression are

$$\hat{\sigma}^2 = \frac{SSR}{N(T-1)-K}$$

For small T this can be an important correction.

Another, generally less popular, way to do fixed effects estimation is to use the first differences estimator

$$(y_{it} - y_{i(t-1)}) = \alpha + \beta(X_{it} - X_{i(t-1)}) + \zeta_{it}.$$

which can be computed by hand in a manner similar to the within estimator.

4.1.2 Random Effects

The package *nlme* contains functions for doing random effects regression (but not fixed effects—the documentation refers to the statistics interpretation of the term "fixed effect") in a linear or nonlinear framework. Suppose we had a linear model with a random effect on the sic3 code.

$$ldsal = (\alpha + \alpha_i) + \beta lemp_{it} + \gamma ldnpt_{it} + \epsilon_{it}$$

We could fit this model using

```
> lme(ldsal~lemp+ldnpt,random=~1|sic3)
```

In general the random effects will be after the vertical bar in the `random` parameter of the model. Placing a `1` between the tilde and vertical bar indicates that the random effect is an intercept term. If we wanted a random effect on one of the exogenous variables as well as the intercept, we could put that variable in the same place as the 1. For example

```
> lme(ldsal~lemp+ldnpt,random=~1+lemp|sic3)
```

corresponds to

$$ldsal = (\alpha + \alpha_i) + (\beta + \beta_i) lemp_{it} + \gamma ldnpt_{it} + \epsilon_{it}$$

For nonlinear random effects models, use `nlme()` instead of `lme()`.

4.2 Qualitative Response

4.2.1 Logit/Probit

There are several ways to do logit and probit regressions in R. The simplest way may be to use the `glm()` command with the family option.

```
> h <- glm(c~y, family=binomial(link="logit"))
```

or replace `logit` with `probit` for a probit regression. The `glm()` function produces an object similar to the `lm()` function, so it can be analyzed using the `summary()` command. In order to extract the log likelihood statistic, use the `logLik()` command.

```
> logLik(h)
'log Lik.' -337.2659 (df=1)
```

4.2.2 Multinomial Logit

There is a function for performing a multinomial logit estimation in the *nnet* library called `multinom()`. To use it, simply transform our dependent variable to a vector of factors (including all cases) and use syntax like a normal regression. If our factors are stored as vectors of dummy variables, we can use the properties of decimal numbers to create unique factors for all combinations. Suppose my factors are `pc`, `inetacc`, and `iapp`, then

```
> g <- pc*1 + inetacc*10 + iapp*100
> multinom(factor(g)~pc.subsidy+inet.subsidy+iapp.subsidy+emple+msamissing)
```

and we get a multinomial logit using all combinations of factors.

Multinomial probit models are characteristically ill conditioned. A method that uses markov chain monte carlo simulations, `mnp()`, is available in the *MNP* library.

4.2.3 Ordered Logit/Probit

The *MASS* library has a function to perform ordered logit or probit regressions, called `polr()`. If `Sat` is an ordered factor vector, then

```
> house.plr <- polr(Sat ~ Infl + Type + Cont, method="probit")
```

4.3 Tobit and Censored Regression

In order to estimate a model in which the values of some of the data have been censored, we use the *survival* library. The function `survreg()` performs this type of regression, and takes as its dependent variable a *Surv* object. The best way to see how to do this type of regression is by example. Suppose we want to regress y on x and z, but a number of y observations were censored on the left and set to zero.

```
result <- survreg(Surv(y,y>0,type='left') ~ x + z, dist='gaussian')
```

The second argument to the `Surv()` function specifies whether each observation has been censored or not (one indicating that it was observed and zero that it was censored). The third argument indicates on which side the data was censored. Since it was the lower tail of this distribution that got censored, we specify `left`. The `dist` option passed to the survreg is necessary in order to get a classical Tobit model.

4.4 Quantile Regression

Ordinary least squares regression methods produce an estimate of the expectation of the dependent variable conditional on the independent. Fitted values, then, are an estimate of the conditional mean. If instead of the conditional mean we want an estimate of the expected conditional median or some other quantile, we use the `rq()` command from the *quantreg* package. The syntax is essentially the same as `lm()` except that we can specify the parameter `tau`, which is the quantile we want (it is between 0 and 1). By default, `tau`=.5, which corresponds to a median regression—another name for least absolute deviation regression.

4.5 Robust Regression - M Estimators

For some datasets, outliers influence the least squares regression line more than we would like them to. One solution is to use a minimization approach using something besides the sum of squared residuals (which corresponds to minimizing the L_2 norm) as our objective function. Common choices are the sum of absolute deviations (L_1) and the Huber method, which is something of a mix between the L_1 and L_2 methods. R implements this robust regression functionality through the `rlm()` command in the MASS library. The syntax is the same as that of the `lm()` command except that it allows the choice of objective function to minimize. That choice is specified by the `psi` parameter. Possible implemented choices are `psi.huber`, `psi.hampel`, and `psi.bisquare`.

In order to specify a custom psi function, we write a function that returns $\psi(x)/x$ if `deriv=0` and $\psi'(x)$ for `deriv=1`. This function than then be passed to `rlm()` using the `psi` parameter.

4.6 Nonlinear Least Squares

Sometimes the economic model just isn't linear. R has the capability of solving for the coefficients a generalized least squares model that can be expressed

$$Y = F(X;\beta) + \epsilon \tag{4}$$

Notice that the error term must be additive in the functional form. If it is not, transform the model equation so that it is. The R function for nonlinear least squares is `nls()` and has a syntax similar to `lm()`. Consider the following nonlinear example.

$$Y = \frac{\epsilon}{1 + e^{\beta_1 X_1 + \beta_2 X_2}} \tag{5}$$

$$\log(Y) = -\log(1 + e^{\beta_1 X_1 + \beta_2 X_2}) + \log(\epsilon) \tag{6}$$

The second equation is the transformed version that we will use for the estimation. `nls()` takes the formula as its first argument and also requires starting estimates for the parameters. The entire formula should be specified, including the parameters. R looks at the starting values to see which parameters it will estimate.

```
> result <- nls(log(Y)~-log(1+exp(a*X1+b*X2)),start=list(a=1,b=1),data=mydata)
```

stores estimates of `a` and `b` in an nls object called `result`. Estimates can be viewed using the `summary()` command. In the most recent versions of R, the `nls()` command is part of the base package, but in older versions, we may have to load the nls library.

4.7 Two Stage Least Squares on a Single Structural Equation

For single equation two stage least squares, the easiest function is probably `tsls()` from the *sem* library. If we want to find the effect of education on wage while controlling for marital status but think `educ` is endogenous, we could use `motheduc` and `fatheduc` as instruments by running

```
> library(sem)
> outputof2sls <- tsls(lwage~educ+married,~married+motheduc+fatheduc)
```

The first argument is the structural equation we want to estimate and the second is a tilde followed by all the instruments and exogenous variables from the structural equation—everything we need for the Z matrix in the 2SLS estimator $\tilde{\beta} = (X'Z(Z'Z)^{-1}Z'X)^{-1}X'Z(Z'Z)^{-1}Z'y$.

The resulting output can be analyzed using `summary()` and other ols analysis functions. Note that since this command produces a two stage least squares object, the summary statistics, including standard errors, will be correct. Recall that if we were to do this using an actual two stage approach, the resulting standard errors would be bogus.

4.8 Systems of Equations

The commands for working with systems of equations (including instrumental variables, two stage least squares, seemingly unrelated regression and variations) are contained in the *systemfit* library. In general these functions take as an argument a list of regression models. Note that in R an equation model (which must include the tilde) is just another data type. Thus we could create a list of equation models and a corresponding list of labels using the normal assignment operator

```
> demand <- q ~ p + d
> supply <- q ~ p + f + a
> system <- list(demand,supply)
> labels <- list("demand","supply")
```

4.8.1 Seemingly Unrelated Regression

Once we have the system and (optionally) labels set up, we can use `systemfit()` with the `SUR` option to specify that the system describes a seemingly unrelated regression.

```
> resultsur <- systemfit("SUR",system,labels)
```

4.8.2 Two Stage Least Squares on a System

Instruments can be used as well in order to do a two stage least squares on the above system. We create a model object (with no left side) to specify the instruments that we will use and specify the `2SLS` option

```
> inst <- ~ d + f + a
> result2sls <- systemfit("2SLS",system,labels,inst)
```

There are also routines for three stage least squares, weighted two stage least squares, and a host of others.

5 Time Series Regression

R has a special datatype, *ts*, for use in time series regressions. Vectors, arrays, and dataframes can be coerced into this type using the `ts()` command for use in time series functions.

```
> datats <- ts(data)
```

Most time-series related functions automatically coerce the data into *ts* format, so this command is often not necessary.

5.1 Differences and Lags

We can compute differences of a time series object using the `diff()` operator, which takes as optional arguments which difference to use and how much lag should be used in computing that difference. For example, to take the first difference with a lag of two, so that $w_t = v_t - v_{t-3}$ we would use

```
> w <- diff(v,lag=2,difference=1)
```

By default, `diff()` returns the simple first difference of its argument.

There are two general ways of generating lagged data. If we want to lag the data directly (without necessarily converting to a time series object), one way to do it is to omit the first few observations using the minus operator for indices. We can then remove the last few rows of un-lagged data in order to achieve conformity. The commands

```
> lagy <- y[-NROW(y)]
> ysmall <- y[-1]
```

produce a once lagged version of `y` relative to `ysmall`. This way of generating lags can get awkward if we are trying combinations of lags in regressions because for each lagged version of the variable, conformability requires that we have a corresponding version of the original data that has the first few observations removed.

Another way to lag data is to convert it to a time series object and use the `lag()` function. It is very important to remember that this function does not actually change the data, it changes an attribute of a time series object that indicates where the series starts. This allows for more flexibility with time series functions, but it can cause confusion for general functions such as `lm()` that do not understand time series attributes. Notice that `lag()` **only works usefully on time series objects**. For example, the code snippet

```
> d <- a - lag(a,-1)
```

creates a vector of zeros named `d` if `a` is a normal vector, but returns a *ts* object with the first difference of the series if `a` is a *ts* object. There is no warning issued if `lag()` is used on regular data, so care should be exercised.

In order to use lagged data in a regression, we can use time series functions to generate a dataframe with various lags of the data and NA characters stuck in the requisite leading and trailing positions. In order to do this, we use the `ts.union()` function. Suppose `X` and `Y` are vectors of ordinary data and we want to include a three times lagged version of `X` in the regression, then

```
> y <- ts(Y)
> x <- ts(X)
> x3 <- lag(x,-3)
> d <- ts.union(y,x,x3)
```

converts the vectors to *ts* data and forms a multivariate time series object with columns $\mathtt{y_t}$, $\mathtt{x_t}$, and $\mathtt{x_{t-3}}$. Again, remember that data must be converted to time series format **before** lagging or binding together with the union operator in order to get the desired offset. The `ts.union()` function automatically decides on a title for each column, must as the `data.frame()` command does. We can also do the lagging inside the union and assign our own titles

```
> y <- ts(Y)
> x <- ts(X)
> d <- ts.union(y,x,x1=lag(xt,-1),x2=lag(xt,-2),x3=lag(xt,-3))
```

It is critical to note that **the lag operator works in the opposite direction of what one might expect**: positive lag values result in leads and negative lag values result in lags.

When the resulting multivariate time series object is converted to a data frame (as it is read by `ls()` for example), the offset will be preserved. Then

```
> lm(y~x3,data=d)
```

will then regress y_t on x_{t-3}.

Also note that by default observations that have a missing value (NA) are omitted. This is what we want. If the default setting has somehow been changed, we should include the argument `na.action=na.omit` in the `lm()` call. In order to get the right omission behavior, it is generally necessary to bind all the data we want to use (dependent and independent variables) together in a single union.

In summary, in order to use time series data, convert all data to type *ts*, lag it appropriately (using the strange convention that positive lags are leads), and bind it all together using `ts.union()`. Then proceed with the regressions and other operations.

5.2 Filters

5.2.1 Canned AR and MA filters

One can pass data through filters constructed by polynomials in the lag operator using the `filter()` command. It handles two main types of filters: moving average or "convolution" filters and autoregressive or "recursive" filters. Convolution filters have the form

$$y = (a_0 + a_1 L + \ldots + a_p L^p)x$$

while recursive filters solve

$$y = (a_1 L + a_2 L^2 + \ldots + a_p L^p)y + x$$

In both cases, `x` is the input to the filter and `y` the output. If `x` is a vector of innovations, the convolution filter generates a moving average series and the recursive filter generates an autoregressive series. Notice that there is no a_0 term in the recursive filter (it would not make sense theoretically). The recursive filter can equivalently be thought of as solving

$$y = (1 - a_1 L - a_2 L^2 - \ldots - a_p L^p)^{-1} x$$

When we use the `filter()` command, we supply the $\{a_n\}$ vector as follows

```
> y <- filter(x,c(1,.2,-.35,.1),method="convolution",sides=1)
```

The data vector `x` may be a time series object or a regular vector of data, and the output `y` will be a *ts* object. It is necessary to specify `sides=1` for a convolution filter, otherwise the software tries to center the filter (in positive and negative lags), which is not usually what the econometrician wants. The recursive filter ignores the `sides` keyword. Notice that (except for data loss near the beginning of the series) we can recover the data from `y` above using a recursive filter

```
> X<-filter(y,c(-.2,.35,-.1),method="recursive")
```

5.2.2 Manual Filtration

If the `filter()` command does not work for our application—or we just prefer doing things ourselves—we can manually generate the lags and compute the result. We could imitate the convolution filter above with

```
> x <- ts(x)
> y <- x+.2*lag(x,-1)-.35*lag(x,-2)+.1*lag(x,-3)
```

The autoregressive filter would require a for loop to reproduce. **Remember that the lag method of filtering will only work if `x` is a *ts* object.**

5.2.3 Hodrick Prescott Filter

Data may be passed through the Hodrick-Prescott filter a couple of ways, neither of which require the data to be a time series vector. First, we can filter manually using the function defined below (included without prompts so it may be copied and pasted into R)

```
hpfilter <- function(x,lambda=1600){
  eye <- diag(length(x))
  result <- solve(eye+lambda*crossprod(diff(eye,lag=1,d=2)),x)
  return(result)
}
```

where `lambda` is the standard tuning parameter, often set to 1600 for macroeconomic data. Passing a series to this function will return the smoothed series.

This filter is also a special case of the `smooth.spline()` function in which the parameter tt all.knots=TRUE has been passed. Unfortunately, the tuning parameter for the `smooth.spline()` function, `spar` is different from the `lambda` above and we have not figured out how to convert from `spar` to `lambda`. If the reader knows how to use `smooth.spline()` to do HP filtration, please let me know.

5.2.4 Kalman Filter

R has multiple functions for smoothing, filtering, and evaluating the likelihood of a state space model using the Kalman filter. The most frequently mentioned is the `KalmanLike` family of functions, but they work only for univariate state space models (that is, models in which there is only one variable in the observations equation). For this reason, the methods in the *dse1* package (`SS.l()` and others) and *sspir* package are often preferred. Because of their greater simplicity, I describe the functions in the *sspir* package.

First we use a function to generate a state space model object, then we can Kalman filter it and optionally smooth the filtered results.. The function for generating the state space object is `SS()`. Recall that a state space model can be written

$$y_t = F_t'\theta_t + v_t$$
$$\theta_t = G_t\theta_{t-1} + w_t$$

where

$$v_t \sim N(0, V_t), \quad w_t \sim N(0, W_t).$$

θ_t is the unobserved state vector, and y_t is the observed data vector. For Kalman filtering, the initial value of θ is drawn from $N(m_0, C_0)$.

Because of the possibly time varying nature of this general model, the coefficient matrices must be given as functions. One caveat to remember is that the input `Fmat` is the transpose of F_t. After writing functions to input F_t, G_t, V_t, W_t and generating the input variables ϕ (a vector of parameters to be used by the functions that generate F_t, G_t, V_t, and W_t), m_0, and C_0, we run `SS()` to create the state space model. Then we run `kfilter()` on the model object to filter and obtain the log likelihood—it returns a copy of the model with estimated parameters included. If we want to run the Kalman smoother, we take the output model of `kfilter()` and run `smoother()` on it.

The functions in package *dse*1 appear more flexible but more complicated to initialize.

5.3 ARIMA/ARFIMA

The `arima()` command from the `ts()` library can fit time series data using an autoregressive integrated moving average model.

$$\Delta^d y_t = \mu + \gamma_1 \Delta^d y_{t-1} + ... + \gamma_p \Delta^d y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + ... + \theta_q \epsilon_{t-q} \tag{7}$$

where

$$\Delta y_t = y_t - y_{t-1} \tag{8}$$

The parameters `p`, `d`, and `q` specify the order of the arima model. These values are passed as a vector `c(p,d,q)` to `arima()`. Notice that the model used by R makes no assumption about the sign of the θ terms, so the sign of the corresponding coefficients may differ from those of other software packages (such as S+).

```
> ar1 <- arima(y,order=c(1,0,0))
> ma1 <- arima(y,order=c(0,0,1))
```

Data-members `ar1` and `ma1` contain estimated coefficients obtained by fitting y with an AR(1) and MA(1) model, respectively. They also contain the log likelihood statistic and estimated standard errors.

Sometimes we want to estimate a high order arima model but set the first few coefficients to zero (or some other value). We do this using the `fixed` parameter. It takes a vector of the same length as the number of estimable parameters. An `NA` entry indicates you want the corresponding parameter to be estimated. For example, to estimate

$$y_t = \gamma_2 y_{t-2} + \theta_1 \epsilon_{t-1} + \epsilon_t \tag{9}$$

we could use

```
> output <- arima(y,order=c(2,0,1),fixed=c(0,NA,NA))
```

I have had reports that if the `fixed` parameter is used, the parameter `transform.pars=FALSE` should also be passed.

If we are modeling a simple autoregressive model, we could also use the `ar()` command, from the *ts* package, which either takes as an argument the order of the model or picks a reasonable default order.

```
> ar3 <- ar(y,order.max=3)
```

fits an AR(3) model, for example.

The function `fracdiff()`, from the *fracdiff* library fits a specified ARMA(p,q) model to our data and finds the optimal fractional value of d for an ARFIMA(p,d,q). Its syntax differs somewhat from the `arima()` command.

```
> library(fracdiff)
> fracdiff(y,nar=2,nma=1)
```

finds the optimal d value using p=2 and q=1. Then it estimates the resulting ARFIMA(p,d,q) model.

5.4 ARCH/GARCH

5.4.1 Basic GARCH–garch()

R can numerically fit data using a generalized autoregressive conditional heteroskedasticity model GARCH(p,q), written

$$y = C + \epsilon \tag{10}$$

$$\sigma_t^2 = \alpha_0 + \delta_1 \sigma_{t-1}^2 + ... + \delta_p \sigma_{t-p}^2 + \alpha_1 \epsilon_t^2 + ... + \alpha_q \epsilon_{t-q}^2 \tag{11}$$

setting $p = 0$ we obtain the ARCH(q) model. The R command `garch()` comes from the *tseries* library. It's syntax is

```
> archoutput <- garch(y,order=c(0,3))
> garchoutput <- garch(y,order=c(2,3))
```

so that `archoutput` is the result of modeling an ARCH(3) model and `garchoutput` is the result of modeling a GARCH(2,3). Notice that the first value in the `order` argument is q, the number of deltas, and the second argument is q, the number of alpha parameters. The resulting coefficient estimates will be named `a0`, `a1`, ...for the alpha and `b1`, `b2`, ...for the delta parameters. Estimated values of the conditional standard deviation process are available via

```
fitted(garchoutput)
```

5.4.2 Advanced GARCH–garchFit()

Of course, we may want to include exogenous variables in the mean equation (10), which `garch()` does not allow. For this we can use the more flexible function `garchFit()` in the *fSeries* package.

```
> garchFitoutput <- garchFit(~arma(0,1)+garch(2,3),y)
```

fits the same model as above, but the mean equation now is an MA(1).

This function returns an S4 class object, which means to access the data inside we use the `@` operator

```
> coef <- garchFitoutput@fit$coef
> fitted <- garchFitoutput@fit$series$h
```

Here `h` gives the estimated σ^2 process, not σ, so it is the square of the "fitted" values from `garch()`, above.

As of this writing, `garchFit()` produces copious output as it estimates the parameters. Some of this can be avoided by passing the parameter `trace=FALSE`.

5.4.3 Miscellaneous GARCH–Ox G@RCH

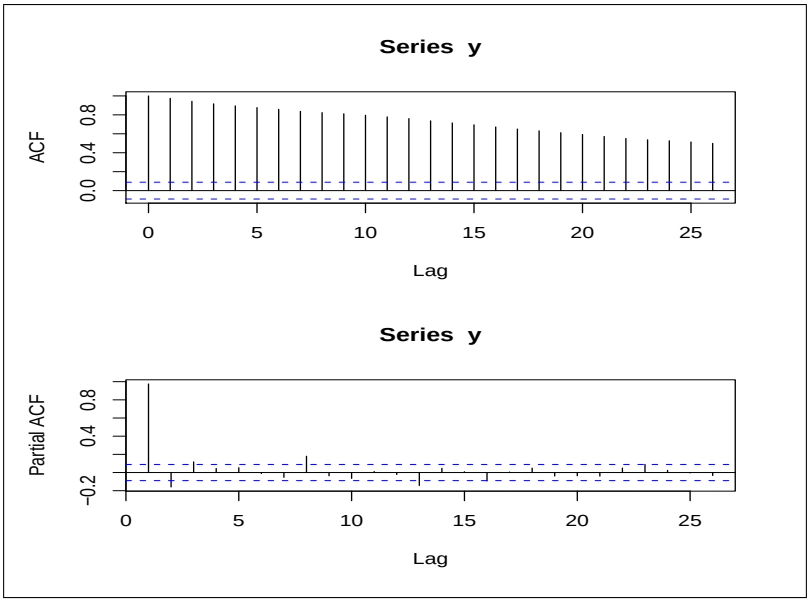
fSeries also provides an interface to some functions from the `Ox G@RCH`[6] software, which is not free in the same sense as R is, although as of this writing it was free for academic use. That interface provides routines for estimation of EGARCH, GJR, APARCH, IGARCH, FIGARCH, FIEGARCH, FIAPARCH and HYGARCH models, according to the documentation.

5.5 Correlograms

It is common practice when analyzing time series data to plot the autocorrelation and partial autocorrelation functions in order to try to guess the functional form of the data. To plot the autocorrelation and partial autocorrelation functions, use the *ts* library functions `acf()` and `pacf()`, respectively. The following commands plot the ACF and PACF on the same graph, one above (not on top of) the other. See section 6.5 for more details on arranging multiple graphs on the canvas.

```
> par(mfrow=c(2,1))
> acf(y)
> pacf(y)
```

These functions also return the numeric values of the ACF and PACF functions along with some other output. Plotting can be suppressed by passing `plot=F`.



[6]Ox and G@RCH are distributed by Timberlake Consultants Ltd. Timberlake Consultants can be contacted through the web site http://www.timberlake.co.uk

5.6 Predicted Values

The `predict()` command takes as its input an lm, glm, arima, or other regression object and some options and returns corresponding predicted values. For time series regressions, such as `arima()` the argument is the number of periods into the future to predict.

```
> a <- arima(y,order=c(1,1,2))
> predict(a,5)
```

returns predictions on five periods following the data in `y`, along with corresponding standard error estimates.

5.7 Time Series Tests

5.7.1 Durbin-Watson Test for Autocorrelation

The Durbin-Watson test for autocorrelation can be administered using the `durbin.watson()` function from the *car* library. It takes as its argument an *lm* object (the output from an `lm()` command) and returns the autocorrelation, DW statistic, and an estimated p-value. The number of lags can be specified using the `max.lag` argument. See help file for more details.

```
> library(car)
> results <- lm(Y ~ x1 + x2)
> durbin.watson(results,max.lag=2)
```

5.7.2 Box-Pierce and Breusch-Godfrey Tests for Autocorrelation

In order to test the residuals (or some other dataset) for autocorrelation, we can use the Box-Pierce test from the *ts* library.

```
> library(ts)
> a <- arima(y,order=c(1,1,0))
> Box.test(a$resid)

        Box-Pierce test

data:  a$resid
X-squared = 18.5114, df = 1, p-value = 1.689e-05
```

would lead us to believe that the model may not be correctly specified, since we soundly reject the Box-Pierce null. If we want to the Ljung-Box test instead, we include the parameter `type="Ljung-Box"`.

For an appropriate model, this test is asymptotically equivalent to the Breusch-Godfrey test, which is available in the `lmtest()` library as `bgtest()`. It takes a fitted *lm* object instead of a vector of data as an argument.

5.7.3 Dickey-Fuller Test for Unit Root

The augmented Dickey-Fuller test checks whether a series has a unit root. The default null hypothesis is that the series does have a unit root. Use the `adf.test()` command from the *tseries* library for this test.

```
> library(tseries)
> adf.test(y)

        Augmented Dickey-Fuller Test

data:  y
Dickey-Fuller = -2.0135, Lag order = 7, p-value = 0.5724
alternative hypothesis: stationary
```

5.8 Vector Autoregressions (VAR)

The standard `ar()` routine can do the estimation part of a vector autoregression. In order to do this type of regression, one need only bind the vectors together as a dataframe and give that dataframe as an argument to `ar()`. Notice that `ar()` by default uses AIC to determine how many lags to use, so it may be necessary to specifiy `aic=FALSE` and/or an `order.max` parameter. Remember that if `aic` is `TRUE` (the default), the function uses AIC to choose a model using up to the number of lags specified by `order.max`.

```
> y <- ts.union(Y1,Y2,Y3)
> var6 <- ar(y,aic=FALSE,order=6)
```

Unfortunately, the `ar()` approach does not have built in functionality for such things as predictions and impulse response functions. The reader may have to code those up by hand if necessary.

Alternately, the `ARMA()` function in the *dse1* library can fit multivariate time series regression in great generality, but the programming overhead is correspondingly great.

There is also a vector autoregression package on CRAN named *VAR*, but I have not used it.

6 Plotting

One of R's strongest points is its graphical ability. It provides both high level plotting commands and the ability to edit even the smallest details of the plots.

The `plot()` command opens a new window and plots the the series of data given it. By default a single vector is plotted as a time series line. If two vectors are given to `plot()`, the values are plotted in the x-y place using small circles. The type of plot (scatter, lines, histogram, etc.) can be determined using the `type` argument. Strings for the main, x, and y labels can also be passed to plot.

```
> plot(x,y,type="l", main="X and Y example",ylab="y values",xlab="x values")
```

plots a line in the x-y plane, for example. Colors, symbols, and many other options can be passed to `plot()`. For more detailed information, see the help system entries for `plot()` and `par()`.

After a plotting window is open, if we wish to superimpose another plot on top of what we already have, we use the `lines()` command or the `points()` command, which draw connected lines and scatter plots, respectively. Many of the same options that apply to `plot()` apply to `lines()` and a host of other graphical functions.

We can plot a line, given its coefficients, using the `abline()` command. This is often useful in visualizing the placement of a regression line after a bivariate regression

```
> results <- lm(y ~ x)
> plot(x,y)
> abline(results$coef)
```

`abline()` can also be used to plot a vertical or horizontal line at a particular value using the parameters `v` or `h` respectively.

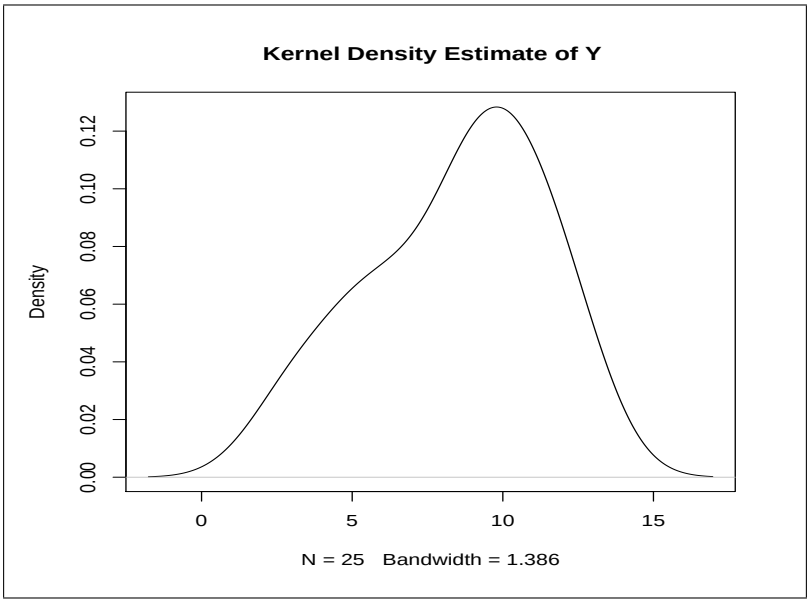
To draw a nonlinear deterministic function, like $f(x) = x^3$, we don't need to generate a bunch of data that lie on the function and then connect those dots. We can plot the function directly using the `curve()` function. If we want to lay the function on top of an already created plot, we can pass the `add=TRUE` parameter.

```
> x <- 1:10
> y <- (x+rnorm(10))^3
> plot(x,y)
> curve(x^3,add=TRUE)
```

6.1 Plotting Empirical Distributions

We typically illustrate the distribution of a vector of data by separating it into bins and plotting it as a histogram. This functionality is available via the `hist()` command. Histograms can often hide true trends in the distribution because they depend heavily on the choice of bin width. A more reliable way of visualizing univariate data is the use of a kernel density estimator, which gives an actual empirical estimate of the PDF of the data. The `density()` function computes a kernel estimator and can be plotted using the `plot()` command.

```
> d <- density(y)
> plot(d,main="Kernel Density Estimate of Y")
```



We can also plot the empirical CDF of a set of data using the `ecdf()` command from the *stepfun* library, which is included in the default distribution. We could then plot the estimated CDF using `plot()`.

```
> library(stepfun)
> d <- ecdf(y)
> plot(d,main="Empirical CDF of Y")
```

6.2 Contour Plots

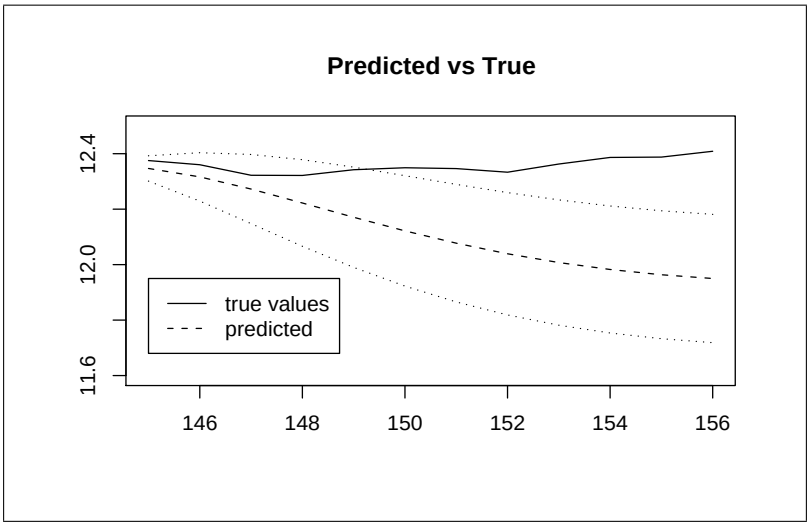
The command `contour()` from the *graphics* package takes a grid of function values and optionally two vectors indicating the x and y values of the grid and draws the contour lines. Contour lines can be added to another plot using the `contourLines()` function in a similar manner. The *lattice* package provides a functions called `levelplot()` and `contourplot()` that are more flexible but less simple to use in my experience. A contour example appears in appendix 13.5.

6.3 Adding Legends and Stuff

After plotting we often wish to add annotations or other graphics that should really be placed manually. Functions like `text()` (see below) and `legend()` take as their first two arguments coordinates on the graph where the resulting objects should be placed. In order to manually determine the location of a point on the graph, use the `locator()` function. The location of one or several right clicks on the graph will be returned by this function after a left click. Those coordinates can then be used to place text, legends, or other add-ons to the graph.

An example of a time series, with a predicted curve and standard error lines around it

```
> plot(a.true,type="l",lty=1,ylim=c(11.6,12.5),main="Predicted vs True",xlab="",ylab="")
> lines(a.predict$pred,lty=2,type="l")
> lines(a.predict$pred+a.predict$se,lty=3,type="l")
> lines(a.predict$pred-a.predict$se,lty=3,type="l")
> legend(145,11.95,c("true values","predicted"),lty=c(1,2))
```



6.4 Adding Arrows, Text, and Markers

After drawing a plot of some type, we can add arrows using the `arrows()` function from the *graphics* library. It takes "from" and "to" coordinates. Text and markers can be added anywhere on the plot using the `text()` and `points()` functions. For `points()` the type of marker is determined by the `pch` parameter. There are many values this can take on, including letters. A quick chart of possible values is the last output of running the command

```
> example(points)
```

An example plot using some of these features is in appendix 13.5.

6.5 Multiple Plots

We can partition the drawing canvas to hold several plots. There are several functions that can be used to do this, including `split.screen()`, `layout()`, and `par()`. The simplest and most important is probably `par()`, so we will examine only it for now. The `par()` function sets many types of defaults about the plots, including margins, tick marks, and layout. We arrange several plots on one canvas by modifying the `mfrow` attribute. It is a vector whose first entry specifies the number of rows of figures we will be plotting and the second, the number of columns. Sometimes when plotting several figures, the default spacing may not be pleasing to the eye. In this case we can modify the default margin (for each plot) using the `mar` attribute. This is a four entry vector specifying the default margins in the form (bottom, left, top, right). The default setting is $\texttt{c}(5,4,4,2)+0.1$. For a top/bottom plot, we may be inclined to decrease the top and bottom margins somewhat. In order to plot a time series with a seasonally adjusted version of it below, we could use

```
> op <- par(no.readonly=TRUE)
> par(mfrow=c(2,1),mar=c(3,4,2,2)+.1)
> plot(d[,1],main="Seasonally Adjusted",ylab=NULL)
> plot(d[,2],main="Unadjusted", ylab=NULL)
> par(op)
```

Notice that we saved the current settings in `op` before plotting so that we could restore them after our plotting and that we must set the `no.readonly` attribute while doing this.

6.6 Saving Plots—png, jpg, eps, pdf, xfig

In order to save plots to files we change the graphics device via the `png()`, `jpg()`, or `postscript()` commands, then we plot what we want and close the special graphics device using `dev.off()`. For example,

```
> png("myplot.png")
> plot(x,y,main="A Graph Worth Saving")
> dev.off()
```

creates a png file of the plot of `x` and `y`. In the case of the postscript file, if we intend to include the graphics in another file (like in a LaTeX document), we could modify the default postscript settings controlling the paper size and orientation. Notice that when the `special` paper size is used (and for best results at other times as well), the width and height must be specified. Actually with LaTeX we often resize the image explicitly, so the resizing may not be that important.

```
> postscript("myplot.eps",paper="special",width=4,height=4,horizontal=FALSE)
> plot(x,y,main="A Graph Worth Including in LaTeX")
> dev.off()
```

One more thing to notice is that the default paper size is a4, which is the European standard. For 8.5x11 paper, we use `paper="letter"`. When using images that have been generated as a postscript, then converted to pdf, incorrect paper specifications are a common problem.

There is also a `pdf()` command that works the same way the `postscript` command does, except that by default its paper size is `special` with a height and width of 6 inches. A common example with `pdf()`, which includes a little room for margins, would be

```
> pdf("myplot.pdf",paper="letter",width=8,height=10.5)
> par(mfrow=c(2,1))
> plot(x,y,main="First Graph (on top)")
> plot(x,z,main="Second Graph (on bottom)")
> dev.off()
```

Notice also that the `par()` command is used *after* the device command, `pdf()`.

Finally, many scientific diagrams are written using the free software *xfig*. R has the capability to export to *xfig* format, which allows us complete flexibility in adding to and altering our plots. If we want to use R to make a generic plot (like indifference curves), we remove the axis numbers and other extraneous marks from the figure.

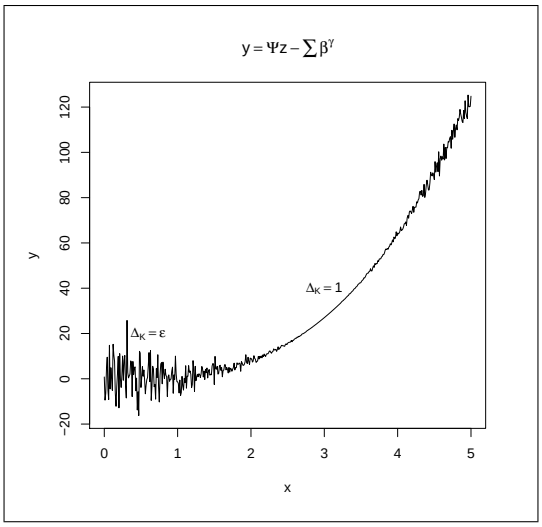
```
> xfig("myoutput.fig", horizontal=F)
> plot(x,(x-.3)^2,type="l",xlab="",ylab="",xaxt="n",yaxt="n")
> dev.off()
```

The `xaxt` and `yaxt` parameters remove the numbers and tic marks from the axes.

6.7 Adding Greek Letters and Math Symbols to Plots

R can typeset a number of mathematical expressions for use in plots using the `substitute()` command. I illustrate with an example (which, by the way, is completely devoid of economic meaning, so don't try to understand the function).

```
> plot(x,y,main=substitute(y==Psi*z-sum(beta^gamma)),type="l")
> text(3,40,substitute(Delta[K]==1))
> text(0.6,20,substitute(Delta[K]==epsilon))
```



Capitalizing the first letter of the Greek symbol results in the "capital" version of the symbol. Notice that to get the equal sign in the expression, one must use the double equal sign, as above. Brackets indicate subscripts. We can optionally pass variables to `substitute()` to include their value in the formula. For example

```
> for (g in seq(.1,1,.1)){
+ plot(f(g),main=substitute(gamma==x,list(x=g)))
> }
```

will make ten plots, in each plot the title will reflect the value of γ that was passed to $f()$. The rules for generating mathematical expressions are available through the help for `plotmath`, which is the mathematical typesetting engine used in R plots.

To mix text and symbols, use the `paste()` command inside of `substitute()`

```
plot(density(tstats),main=substitute(paste("t-stat of ",beta[0])))
```

6.8 Other Graphics Packages

So far I have discussed the R base plotting package. It is very sophisticated and useful when compared with the plotting capabilities of many other statistical software packages. It is not all inclusive, however, and there are other graphics libraries in R which might prove useful, including *grid*, *lattice*, *and ggplot2*.

7 Statistics

R has extensive statistical functionality. The functions `mean()`, `sd()`, `min()`, `max()`, and `var()` operate on data as we would expect[7]. If our data is a matrix and we would like to find the mean or sum of each row or column, the fastest and best way is to use one of `rowMeans()`, `colMeans()`, `rowSums()`, `colSums()`.

7.1 Working with Common Statistical Distributions

R can also generate and analyze realizations of random variables from the standard distributions. Commands that generate random realizations begin with the letter 'r' and take as their first argument the number of observations to generate; commands that return the value of the pdf at a particular observation begin with 'd'; commands that return the cdf value of a particular observation begin with 'p'; commands that return the number corresponding to a cdf value begin with q. Note that the 'p' and 'q' functions are inverses of each other.

[7]Note: the functions `pmax()` and `pmin()` function like max and min but elementwise on vectors or matrices.

```
> rnorm(1,mean=2,sd=3)
[1] 2.418665
> pnorm(2.418665,mean=2,sd=3)
[1] 0.5554942
> dnorm(2.418665,mean=2,sd=3)
[1] 0.1316921
> qnorm(.5554942,mean=2,sd=3)
[1] 2.418665
```

These functions generate a random number from the N(2,9) distribution, calculate its cdf and pdf value, and then verify that the cdf value corresponds to the original observation. If we had not specified the mean and standard deviation, R would have assumed standard normal.

Command	**Meaning**
rX()	Generate random vector from distribution X
dX()	Return the value of the PDF of distribution X
pX()	Return the value of the CDF of distribution X
qX()	Return the number at which the CDF hits input value [0,1]

Note that we could replace `norm` with one of the following standard distribution names

Distribution	**R Name**	**Possible Arguments**
beta	`beta`	shape1, shape2, ncp
binomial	`binom`	size, prob
Cauchy	`cauchy`	location, scale
chi-squared	`chisq`	df, ncp
exponential	`exp`	rate
F	`f`	df1, df1, ncp
gamma	`gamma`	shape, scale
geometric	`geom`	prob
hypergeometric	`hyper`	m, n, k
log-normal	`lnorm`	meanlog, sdlog
logistic	`logis`	location, scale
negative binomial	`nbinom`	size, prob
normal	`norm`	mean, sd
Poisson	`pois`	lambda
Students t	`t`	df, ncp
uniform	`unif`	min, max
Weibull	`weibull`	shape, scale
Wilcoxon	`wilcox`	m, n

The *mvtnorm* package provides the multivariate normal and t distributions with names `mvnorm` and `mvt`, respectively. Other distributions are found in other packages. For example, `invgamma` is available in *MCMCpack*.

7.2 P-Values

By way of example, in order to calculate the p-value of 3.6 using an $f(4, 43)$ distribution, we would use the command

```
> 1-pf(3.6,4,43)
[1] 0.01284459
```

and find that we fail to reject at the 1% level, but we would be able to reject at the 5% level. Remember, if the p-value is smaller than the alpha value, we are able to reject. Also recall that the p-value should be multiplied by two if it we are doing a two tailed test. For example, the one and two tailed tests of a t statistic of 2.8 with 21 degrees of freedom would be, respectively

```
> 1-pt(2.8,21)
[1] 0.005364828
> 2*(1-pt(2.8,21))
[1] 0.01072966
```

So that we would reject the null hypothesis of insignificance at the 10% level if it were a one tailed test (remember, small p-value, more evidence in favor of rejection), but we would fail to reject in the sign-agnostic case.

7.3 Sampling from Data

R provides a convenient and fast interface for sampling from data (e.g., for bootstrapping). Because it calls a compiled function, it is likely to be much faster than a hand-written sampler. The function is `sample()`. The first argument is either the data from which to sample or an integer—if an integer is given, then the sample is taken from the vector of integers between one and that number. The second is the size of the sample to obtain. The parameter `replace` indicates whether to sample with or without replacement. Finally, a vector of sample probabilities can optionally be passed.

8 Math in R

8.1 Matrix Operations

8.1.1 Matrix Algebra and Inversion

Most R commands work with multiple types of data. Most standard mathematical functions and operators (including multiplication, division, and powers) operate on each component of multidimensional objects. Thus the operation `A*B`, where `A` and `B` are matrices, multiplies corresponding components. In order to do matrix multiplication or inner products, use the `%*%` operator. Notice that in the case of matrix-vector multiplication, R will automatically make the vector a row or column vector, whichever is conformable. Matrix inversion is obtained via the `solve()` function. (Note: if `solve()` is passed a matrix and a vector, it solves the corresponding linear problem) The `t()` function transposes its argument. Thus

$$\beta = (X'X)^{-1}X'Y \tag{12}$$

would correspond to the command

```
> beta <- solve(t(X)%*%X)%*%t(X)%*%Y
```

or more efficiently

```
> beta <- solve(t(X)%*%X,t(X)%*%Y)
```

The Kronecker product is also supported and is specified by the the `%x%` operator.

```
> bigG <- g%x%h
```

calculates the Kronecker product of `g` with `h`. The outer product, `%o%` is also supported. When applied to pure vectors (which we recall have only one dimension and thus are neither rows or columns), both matrix products makes different assumptions about whether the arguments are row or column vectors, depending on their position. For example

```
> h<-c(1,2,3)
> h%*%h
     [,1]
[1,]   14
> h%o%h
     [,1] [,2] [,3]
[1,]    1    2    3
```

```
[2,]    2    4    6
[3,]    3    6    9
> t(h)%*%h
     [,1]
[1,]   14
> h%*%t(h)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    2    4    6
[3,]    3    6    9
```

Note that `t(h)%o%h` would produce a 1x3x3 array since the `t()` operator makes `h` into a row vector and `%o%` makes the second `h` into a row vector. Strictly speaking those arguments are not conformable. That combination should probably be avoided.

The trace of a square matrix is calculated by the function `tr()` and its determinant by `det()`. The *Matrix* package provides the various matrix norms (`norm()`), sparse and symmetric matrix support, and other linear algebra-ish functionality. It should be remembered that the *Matrix* package provides its own class `Matrix`, which is distinct from the standard R type `matrix`. In order to use functions from the *Matrix* package, they must be converted using `Matrix()`.

8.1.2 Factorizations

R can compute the standard matrix factorizations. The Cholesky factorization of a symmetric positive definite matrix is available via `chol()`. It should be noted that `chol()` does not check for symmetry in its argument, so the user must be careful.

We can also extract the eigenvalue decomposition of a symmetric matrix using `eigen()`. By default this routine checks the input matrix for symmetry, but it is probably better to specify whether the matrix is symmetric by construction or not using the parameter `symmetric`.

```
> J <- cbind(c(20,3),c(3,18))
> j <- eigen(J,symmetric=T)
> j$vec%*%diag(j$val)%*%t(j$vec)
     [,1] [,2]
[1,]   20    3
[2,]    3   18
```

If the more general singular value decomposition is desired, we use instead `svd()`. For the QR factorization, we use `qr()`. The `Matrix` package provides the `lu()` and `Schur()` decompositions—just remember to convert the matrix to type `Matrix` (not `matrix`) before using them.

8.2 Numerical Optimization

8.2.1 Unconstrained Minimization

R can numerically minimize an arbitrary function using either `nlm()` or `optim()`. I prefer the latter because it lets the user choose which optimization method to use (BFGS, conjugate gradients, simulated annealing, and others), but they work in similar ways. For simplicity I describe `nlm()`.

The `nlm()` function takes as an argument a function and a starting vector at which to evaluate the function. The fist argument of the user-defined function should be the parameter(s) over which R will minimize the function, additional arguments to the function (constants) should be specified by name in the `nlm()` call.

```
> g <- function(x,A,B){
+ out <- sin(x[1])-sin(x[2]-A)+x[3]^2+B
+ out
+ }
```

```
> results <- nlm(g,c(1,2,3),A=4,B=2)
> results$min
[1] 6.497025e-13
> results$est
[1] -1.570797e+00 -7.123895e-01 -4.990333e-07
```

Here `nlm()` uses a matrix-secant method that numerically approximates the gradient, but if the return value of the function contains an attribute called `gradient`, it will use a quasi-newton method. The gradient based optimization corresponding to the above would be

```
> g <- function(x,A,B){
+ out <- sin(x[1])-sin(x[2]-A)+x[3]^2+B
+ grad <- function(x,A){
+   c(cos(x[1]),-cos(x[2]-A),2*x[3])
+ }
+ attr(out,"gradient") <- grad(x,A)
+ return(out)
+ }
> results <- nlm(g,c(1,2,3),A=4,B=2)
```

If function maximization is wanted one should multiply the function by -1 and minimize. If `optim()` is used, one can instead pass the parameter `control=list(fnscale=-1)`, which indicates a multiplier for the objective function and gradient. It can also be used to scale up functions that are nearly flat so as to avoid numerical inaccuracies.

Other optimization functions which may be of interest are `optimize()` for one-dimensional minimization, `uniroot()` for root finding, and `deriv()` for calculating numerical derivatives.

8.2.2 Minimization with Linear Constraints

Function minimization subject to a set of linear inequality constraints is provided by `constrOptim()`. The set of constraints must be expressible in the form

$$U_i'\theta - C_i \geq 0$$

where U_i and C_i are known constant vectors and θ is the vector of parameters over which we are optimizing. For example, to solve

$$\hat{\theta} = \text{argmax}_{\theta' m_i \leq 1} f(\theta)$$

we would use

```
> thetahat<-constrOptim(c(0,0,0,0,0),-f,NULL,ui=-m,ci=-1)$par
```

The first argument is the starting value (θ has five parameters), the function is multiplied by -1 since we are maximizing. The next argument should be a function giving the gradient of f. If we do not have such a function, we must insert `NULL` so that a non-gradient method optimization method is used.

8.3 Numerical Integration

We can use the function `integrate()` from the *stats* package to do unidimensional integration of a known function. For example, if we wanted to find the constant of integration for a posterior density function we could define the function and then integrate it

```
> postdensity <- function(x){
+  exp(-1/2*((.12-x)^2+(.07-x)^2+(.08-x)^2))
+ }
> const <- 1/integrate(postdensity,-Inf,Inf)$value
```

We notice that `integrate()` returns additional information, such as error bounds, so we extract the value using `$value`. Also, in addition to a function name, `integrate()` takes limits of integration, which—as in this case—may be infinite. For multidimensional integration we instead use `adapt()` from the *adapt* library, which does not allow infinite bounds.

9 Programming

9.1 Writing Functions

A function can be treated as any other object in R. It is created with the assignment operator and `function()`, which is passed an argument list (use the equal sign to denote default arguments; all other arguments will be required at runtime). The code that will operate on the arguments follows, surrounded by curly brackets if it comprises more than one line.

If an expression or variable is evaluated within a function, it will not echo to the screen. However, if it is the last evaluation within the function, it will act as the return value. This means the following functions are equivalent

```
> g <- function(x,Alpha=1,B=0) sin(x[1])-sin(x[2]-Alpha)+x[3]^2+B
> f <- function(x,Alpha=1,B=0){
+ out <- sin(x[1])-sin(x[2]-Alpha)+x[3]^2+B
+ return(out)
+ }
```

Notice that R changes the prompt to a "+" sign to remind us that we are inside brackets.

Because R does not distinguish what kind of data object a variable in the parameter list is, we should be careful how we write our functions. If `x` is a vector, the above functions would return a vector of the same dimension. Also, notice that if an argument has a long name, it can be abbreviated as long as the abbreviation is unique. Thus the following two statements are equivalent

```
> f(c(2,4,1),Al=3)
> f(c(2,4,1),Alpha=3)
```

Function parameters are passed by value, so changing them inside the function does not change them outside of the function. Also variables defined within functions are unavailable outside of the function. If a variable is referenced inside of a function, first the function scope is checked for that variable, then the scope above it, etc. In other words, variables outside of the function are available to code inside for reading, but changes made to a variable defined outside a function are lost when the function terminates. For example,

```
> a<-c(1,2)
> k<-function(){
+ cat("Before: ",a,"\n")
+ a<-c(a,3)
+ cat("After: ",a,"\n")
+ }
> k()
Before:  1 2
After:  1 2 3
> a
[1] 1 2
```

If a function wishes to write to a variable defined in the scope above it, it can use the "superassignment" operator `<<-`. The programmer should think twice about his or her program structure before using this operator. Its need can easily be the result of bad programming practices.

9.2 Looping

Looping is performed using the `for` command. It's syntax is as follows

```
> for (i in 1:20){
+ cat(i)
> }
```

Where `cat()` may be replaced with the block of code we wish to repeat. Instead of `1:20`, a vector or matrix of values can be used. The index variable will take on each value in the vector or matrix and run the code contained in curly brackets.

If we simply want a loop to run until something happens to stop it, we could use the `repeat` loop and a `break`

```
> repeat {
+ g <- rnorm(1)
+ if (g > 2.0) break
+ cat(g);cat("\n")
> }
```

Notice the second `cat` command issues a newline character, so the output is not squashed onto one line. The semicolon acts to let R know where the end of our command is, when we put several commands on a line. For example, the above is equivalent to

```
> repeat {g <- rnorm(1);if (g>2.0) break;cat(g);cat("\n");}
```

In addition to the `break` keyword, R provides the `next` keyword for dealing with loops. This terminates the current iteration of the `for` or `repeat` loop and proceeds to the beginning of the next iteration (programmers experienced in other languages sometimes expect this keyword to be *continue* but it is not).

9.3 Avoiding Loops

9.3.1 Applying a Function to an Array (or a Cross Section of it)

To help the programmer avoid the sluggishness associated with writing and executing loops, R has a command to call a function with each of the rows or columns of an array. We specify one of the dimensions in the array, and for each element in that dimension, the resulting cross section is passed to the function.

For example, if `X` is a 50x10 array representing 10 quantities associated with 50 individuals and we want to find the mean of each row (or column), we could write

```
> apply(X,1,mean) # for a 50-vector of individual (row) means
> apply(X,2,mean) # for a 10-vector of observation (column) means
```

Of course, an array may be more than two dimensional, so the second argument (the dimension over which to apply the function) may go above 2. A better way to do this particular example, of course, would be to use `rowMeans()` or `colMeans()`.

We can use `apply()` to apply a function to every element of an array individually by specifying more than one dimension. In the above example, we could return a 50x10 matrix of normal quantiles using

```
> apply(X,c(1,2),qnorm,mean=3,sd=4)
```

After the required three arguments, any additional arguments are passed to the inside function, `qnorm` in this case.

In order to execute a function on each member of a list, vector, or other R object, we can use the function `lapply()`. The result will be a new list, each of whose elements is the result of executing the supplied function on each element of the input. The syntax is the same as `apply()` except the dimension attribute is not needed. To get the results of these function evaluations as a vector instead of a list, we can use `sapply()`.

9.3.2 Replicating

If the programmer wishes to run a loop to generate values (as in a simulation) that do not depend on the index, the function `replicate()` provides a convenient and fast interface. To generate a vector of 50000 draws from a user defined function called `GetEstimate()` which could, for example, generate simulated data and return a corresponding parameter estimate, the programmer could execute

```
> estimates<-replicate(50000,GetEstimate(alpha=1.5,beta=1))
```

If `GetEstimate()` returns a scalar, `replicate()` generates a vector. It it returns a vector, `replicate()` will column bind them into a matrix. Notice that `replicate()` always calls its argument function with the same parameters—in this case 1.5 and 1.

9.4 Conditionals

9.4.1 Binary Operators

Conditionals, like the rest of R, are highly vectorized. The comparison

```
> x < 3
```

returns a vector of TRUE/FALSE values, if x is a vector. This vector can then be used in computations. For example. We could set all x values that are less that 3 to zero with one command

```
> x[x<3] <- 0
```

The conditional within the brackets evaluates to a TRUE/FALSE vector. Wherever the value is TRUE, the assignment is made. Of course, the same computation could be done using a `for` loop and the `if` command.

```
> for (i in 1:NROW(x)){
+ if (x[i] < 3) {
+  x[i] <- 0
+ }
+ }
```

Because R is highly vectorized, the latter code works much more slowly than the former. It is generally good programming practice to avoid loops and `if` statements whenever possible when writing in any scripting language[8].

The Boolean Operators

`!  x`	NOT x
`x & y`	x and y elementwise
`x && y`	x and y total object
`x | y`	x or y elementwise
`x || y`	x or y total object
`xor(x, y)`	x xor y (true if one and only one argument is true)

9.4.2 WARNING: Conditionals and NA

It should be noted that using a conditional operator with an *NA* or *NaN* value returns *NA*. This is often what we want, but causes problems when we use conditionals within an `if` statement. For example

```
> x <- NA
> if (x == 45) cat("Hey There")
Error in if (x == 45) cat("Hey There") : missing value where TRUE/FALSE needed
```

For this reason we must be careful to include plenty of `is.na()` checks within our code.

9.5 The Ternary Operator

Since code segments of the form

```
> if (x) {
+ y } else {
+ z }
```

[8] Although it is also possible to try too hard to remove loops, complicating beyond recognition and possibly even slowing the code.

come up very often in programming, R includes a ternary operator that performs this in one line

```
> ifelse(x,y,z)
```

If `x` evaluates to `TRUE`, then `y` is returned. Otherwise `z` is returned. This turns out to be helpful because of the vectorized nature of R programming. For example, `x` could be a vector of `TRUE/FALSE` values, whereas the long form would have to be in a loop or use a roundabout coding method to achieve the same result.

9.6 Outputting Text

Character strings in R can be printed out using the `cat()` function. All arguments are coerced into strings and then concatenated using a separator character. The default separator is a space.

```
> remaining <- 10
> cat("We have",remaining,"left\n")
We have 10 left
> cat("We have",remaining,"left\n",sep="")
We have10left
```

In order to print special characters such as tabs and newlines, R uses the C escape system. Characters preceded by a backslash are interpreted as special characters. Examples are `\n` and `\t` for newline and tab, respectively. In order to print a backslash, we use a double backslash `\\`. When outputting from within scripts (especially if there are bugs in the script) there may be a delay between the output command and the printing. To force printing of everything in the buffer, use `flush(stdout())`.

To generate a string for saving (instead of displaying) we use the `paste()` command, which turns its arguments into a string and returns it instead of printing immediately.

9.7 Pausing/Getting Input

Execution of a script can be halted pending input from a user via the `readline()` command. If `readline()` is passed an argument, it is treated as a prompt. For example, a command to pause the script at some point might read

```
> blah <- readline("Press <ENTER> to Continue.")
```

the function returns whatever the user inputted. It is good practice to assign the output of `readline()` to something (even if it is just a throw-away variable) so as to avoid inadvertently printing the input to the screen or returning it—recall that if a function does not explicitly call `return()` the last returned value inside of it becomes its return value.

The function `readline()` always returns a string. If a numeric quantity is wanted, it should be converted using `as.numeric()`.

9.8 Timing Blocks of Code

If we want to know the compute time of a certain block of code, we can pass it as an argument to the `system.time()` function. Suppose we have written a function called `slowfunction()` and we wish to know how much processor time it consumes.

```
> mytime <- system.time(myoutput <- slowfunction(a,b))
> mytime
[1] 16.76666667  0.08333333 17.00000000  0.00000000  0.00000000
```

The output of `slowfunction()` is stored in `myoutput` and the elements of `mytime` are user, system and total times (in seconds), followed by totals of user and system times of child processes spawned by the expression.

I often find it inconvenient to pass code to `system.time()`, so instead we can call `proc.time()`, which tells how much time this R session has consumed, directly and subtract.

```
> mytime <- proc.time()
> myoutput <- slowfunction(a,b)
> proc.time() - mytime
[1] 16.76666667  0.08333333 17.00000000  0.00000000  0.00000000
```

We are generally interested only in the first number returned by `system.time()` or `proc.time()`.

9.9 Calling C functions from R

Some programming problems have elements that are just not made for an interpreted language because they require too much computing power (especially if they require too many loops). These functions can be written in C, compiled, and then called from within R[9]. R uses the system compiler (if you have one) to create a shared library (ending in `.so` or `.dll`, depending on your system) which can then be loaded using the `dyn.load()` function.

9.9.1 How to Write the C Code

A function that will be called from within R should have type `void` (it should not return anything except through its arguments). Values are passed to and from R by reference, so all arguments are pointers. Real numbers (or vectors) are passed as type `double*`, integers and boolean as type `int*`, and strings as type `char**`. If inputs to the function are vectors, their length should be passed manually. Also note that objects such as matrices and arrays are just vectors with associated dimension attributes. When passed to C, only the vector is passed, so the dimensions should be passed manually and the matrix recreated in C if necessary.

Here is an example of a C file to compute the dot product of two vectors

```
void gdot(double *x,double *y,int *n,double *output){
  int i;
  *output=0;
  for (i=0;i<*n;i++){
    *output+=x[i]*y[i];
  }
}
```

No header files need to be included unless more advanced functionality is required, such as passing complex numbers (which are passed as a particular C structure). In that case, include the file `R.h`.

Do not use `malloc()` or `free()` in C code to be used with R. Instead use the R functions `Calloc()` and `Free()`. R does its own memory management, and mixing it with the default C memory stuff is a bad idea.

Outputting from inside C code should be done using `Rprintf()`, `warning()`, or `error()`. These functions have the same syntax as the regular C command `printf()`, which should not be used.

It should also be noted that long computations in compiled code cannot be interrupted by the user. In order to check for an interrupt signal from the user, we include

```
#include <R_ext/Utils.h>
...
R_CheckUserInterrupt();
```

in appropriate places in the code.

9.9.2 How to Use the Compiled Functions

To compile the library, from the command line (not inside of R) use the command

```
R CMD SHLIB mycode.c
```

[9]My experience has been that the speedup of coding in C is not enough to warrant the extra programming time except for extremely demanding problems. If possible, I suggest working directly in R. It's quite fast—as interpreted languages go. It is somewhat harder to debug C code from within R and the C/R interface introduces a new set of possible bugs as well.

This will generate a shared library called `mycode.so`. To call a function from this library we load the library using `dyn.load()` and then call the function using the `.C()` command. This command makes a copy of each of its arguments and passes them all by reference to C, then returns them as a list. For example, to call the dot product function above, we could use

```
> x<-c(1,4,6,2)
> y<-c(3,2.4,1,9)
> dyn.load("mycode.so")
> product<-.C("gdot",myx=as.double(x),myy=as.double(y),myn=as.integer(NROW(x)),myoutput=numeric(1))
> product$myoutput
[1] 36.6
```

Notice that when `.C()` was called, names were given to the arguments only for convenience (so the resulting list would have names too). The names are not passed to C. It is good practice (and often necessary) to use `as.double()` or `as.integer()` around each parameter passed to `.C()`. If compiled code does not work or works incorrectly, this should be checked first.

It is important to create any vectors from within R that will be passed to `.C()` before calling them. If the data being passed to `.C()` is large and making a copy for passing is not desirable, we can instruct `.C()` to edit the data in place by passing the parameter `DUP=FALSE`. The programmer should be very wary when doing this, because any variable changed in the C code will be changed in R also and there are subtle caveats associated with this. The help file for `.C()` or online documentation give more information.

There is also a `.Fortran()` function. Notice that `.C()` and `.Fortran()` are the simple ways to call functions from these languages, they do not handle `NA` values or complicated R objects. A more flexible and powerful way of calling compiled functions is `.Call()`, which handles many more types of R objects but adds significantly to the complexity of the programming. The `.Call()` function is a relatively recent addition to R, so most of the language was written using the simple but inflexible `.C()`.

9.10 Calling R Functions from C

Compiled C code that is called from R can also call certain R functions (fortran can not). In particular, the functions relating to drawing from and evaluating statistical distributions are available. To access these functions, the header file `Rmath.h` must be included. Unfortunately these C functions are not well documented, so the programmer may have to look up their definitions in `Rmath.h` on the local system. Before calling these functions, `GetRNGstate()` must be called, and `PutRNGstate()` must be called afterward. Below is a C function that generates an AR(1) series with N(0,1) errors and a supplied coefficient.

```
#include<Rmath.h>
void ar1(double *y,double *rho,double *N){
  int i;
  GetRNGstate();
  for (i=1;i<N[0];i++){
    y[i]=rho[0]*y[i-1]+rnorm(0.0,1.0);
  }
  PutRNGstate();
}
```

which could be called (as usual) from within R using

```
> dyn.load("ar1.so")
> X<-.C("ar1",x=double(len=5000),rho=as.double(.9),n=as.integer(5000))$x
```

Most common mathematical operations, such as `sqrt()` are also available through the C interface.

Actual R expressions can also be called from within C, but this is not recommended since it invokes a new instance of R and is slower than terminating the C code, doing a computation in R, and calling another C function. The method for doing it is the (now depreciated) `call_R()` function.

10 Changing Configurations

10.1 Default Options

A number of runtime options relating to R's behavior are governed by the `options()` function. Running this function with no arguments returns a list of the current options. One can change the value of a single option by passing the option name and a new value. For temporary changes, the option list may be saved and then reused.

```
> oldops <- options()
> options(verbose=true)
...
> options(oldops)
```

10.1.1 Significant Digits

Mathematical operations in R are generally done to full possible precision, but the format in which, for example, numbers are saved to a file when using a write command depends on the option `digits`.

```
> options(digits=10)
```

increases this from the default 7 to 10.

10.1.2 What to do with NAs

The behavior of most R functions when they run across missing values is governed by the option `na.action`. By default it is set to `na.omit`, meaning that the corresponding observation will be ignored. Other possibilities are `na.fail`, `na.exclude`, and `na.pass`. The value `na.exclude` differs from `na.omit` only in the type of data it returns, so they can usually be used interchangeably.

These NA handling routines can also be used directly on the data. Suppose we wish to remove all missing values from an object `d`.

```
> cleand <- na.omit(d)
```

Notice that `na.omit()` adds an extra attribute to `d` called `na.action` which contains the row names that were removed. We can remove this attribute using `attr()`.

```
> attr(cleand,"na.action")<-NULL
```

This is the general way to change attributes of an R object. We view all attributes using the `attribute()` command.

10.1.3 How to Handle Errors

When an error occurs in a function or script more information may be needed than the type of error that occurs. In this case, we can change the default behavior of error handling. This is set via the `error` option, which is by default set to `NULL` or `stop`. Setting this option to `recover` we enter debug mode on error. First R gives a list of "frames" or program locations to start from. After selecting one, the user can type commands as if in interactive mode there. In the example below, one of the indices in my loop was beyond the dimension of the matrix it was referencing. First I check `i`, then `j`.

```
> options(error=recover)
> source("log.R")
Error: subscript out of bounds

Enter a frame number, or 0 to exit

1: source("log.R")
```

```
2: eval.with.vis(ei, envir)
3: eval.with.vis(expr, envir, enclos)
4: mypredict(v12, newdata = newdata)

Selection: 4
Called from: eval(expr, envir, enclos)
Browse[1]> i
[1] 1
Browse[1]> j
[1] 301
```

Pressing enter while in browse mode takes the user back to the menu. After debugging, we can set `error` to `NULL` again.

10.1.4 Suppressing Warnings

Sometimes non-fatal warnings issued by code annoyingly uglifies output. In order to suppress these warnings, we use `options()` to set `warn` to a negative number. If `warn` is one, warnings are printed are printed as they are issued by the code. By default warnings are saved until function completion `warn=0`. Higher numbers cause warnings to be treated as errors.

11 Saving Your Work

11.1 Saving the Data

When we choose to exit, R asks whether we would like to save our workspace image. This saves our variables, history, and environment. You manually can save R's state at any time using the command

```
> save.image()
```

You can save one or several data objects to a specified file using the `save()` command.

```
> save(BYU,x,y,file="BYUINFO.Rdata")
```

saves the variables `BYU`, `x`, and `y` in the default R format in a file named "BYUINFO.Rdata". They can be loaded again using the command

```
> load("BYUINFO.Rdata")
```

R can save to a number of other formats as well. Use `write.table()` to write a data frame as a space-delimited text file with headers, for example. Some other formats are listed in section 2.6.

11.2 Saving the Session Output

We may also wish to write the output of our commands to a file. This is done using the `sink()` command.

```
> sink("myoutput.txt")
> a
> sink()
```

The output of executing the command `a` (that is, echoing whatever a is) is written to "myoutput.txt". Using `sink()` with no arguments starts output echoing to the screen again. By default, `sink()` hides the output from us as we are interacting with R, so we can get a transcript of our session and still see the output by passing the `split=T` keyword to tt sink()[10].

If we are using a script file, a nice way to get a transcript of our work and output is to use `sink()` in connection with `source()`.

[10] Thanks to Trevor Davis for this observation.

```
> sink("myoutput.txt")
> source("rcode.R",echo=T)
> sink()
```

R can save plots and graphs as image files as well. Under windows, simply click once on the graph so that it is in the foreground and then go to *file/Save as* and save it as jpeg or png. There are also ways to save as an image or postscript file from the command line, as described in section 6.6.

11.3 Saving as LaTeX

R objects can also be saved as LaTeX tables using the `latex()` command from the *Hmisc* library. The most common use we have had for this command is to save a table of the coefficients and estimates of a regression.

```
> reg <- lm(educ~exper+south,data=d)
> latex(summary(reg)$coef)
```

produces a file named "summary.tex" that produces the following when included in a LaTeX source file[11]

summary	Estimate	Std. Error	t value	Pr(¿—t—)
(Intercept)	17.2043926	0.088618337	194.140323	$0.00000e+00$
exper	−0.4126387	0.008851445	−46.618227	$0.00000e+00$
south	−0.7098870	0.074707431	−9.502228	$4.05227e-21$

which we see is pretty much what we want. The table lacks a title and the math symbols in the p-value column are not contained in $ characters. Fixing these by hand we get

OLS regression of `educ` on `exper` and `south`

summary	Estimate	Std. Error	t value	Pr(> |t|)
(Intercept)	17.2043926	0.088618337	194.140323	$0.00000e+00$
exper	−0.4126387	0.008851445	−46.618227	$0.00000e+00$
south	−0.7098870	0.074707431	−9.502228	$4.05227e-21$

Notice that the `latex()` command takes matrices, summaries, regression output, dataframes, and many other data types. Another option, which may be more flexible, is the `xtable()` function from the *xtable* library.

12 Final Comments

It is my opinion that R provides an effective platform for econometric computation and research. It has built-in functionality sufficiently advanced for professional research, is highly extensible, and has a large community of users. On the other hand, it takes some time to become familiar with the syntax and reasoning of the language, which is the problem I seek to address here.

I hope this paper has been helpful and easy to use. If a common econometric problem is not addressed here, I would like to hear about it so I can add it and future econometricians need not suffer through the process of figuring it out themselves. Please let me know by email (if the date today is before, say, May 2009) what the problem is that you think should be addressed. My email is `g-farnsworth@kellogg.northwestern.edu`. If possible, please include the solution as well.

[11] Under linux, at least, the `latex()` command also pops up a window showing how the output will look.

13 Appendix: Code Examples

13.1 Monte Carlo Simulation

The following block of code creates a vector of randomly distributed data X with 25 members. It then creates a y vector that is conditionally distributed as

$$y = 2 + 3x + \epsilon. \tag{13}$$

It then does a regression of x on y and stores the slope coefficient. The generation of y and calculation of the slope coefficient are repeated 500 times. The mean and sample variance of the slope coefficient are then calculated. A comparison of the sample variance of the estimated coefficient with the analytic solution for the variance of the slope coefficient is then possible.

```
>A <- array(0, dim=c(500,1))
>x <- rnorm(25,mean=2,sd=1)
>for(i in 1:500){
+ y <- rnorm(25, mean=(3*x+2), sd=1)
+ beta <- lm(y~x)
+ A[i] <- beta$coef[2]
+ }
>Abar <- mean(A)
>varA <- var(A)
```

13.2 The Haar Wavelet

The following code defines a function that returns the value of the Haar wavelet, defined by

$$\psi^{(H)}(u) = \begin{cases} -1/\sqrt{2} & -1 < u \leq 0 \\ 1/\sqrt{2} & 0 < u \leq 1 \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

of the scalar or vector passed to it. Notice that a better version of this code would use a vectorized comparison, but this is an example of conditionals, including the `else` statement. The interested student could rewrite this function without using a loop.

```
> haar <- function(x){
+ y <- x*0
+ for(i in 1:NROW(y)){
+   if(x[i]<0 && x[i]>-1){
+      y[i]=-1/sqrt(2)
+   } else if (x[i]>0 && x[i]<1){
+      y[i]=1/sqrt(2)
+   }
+ }
+ y
+ }
```

Notice also the use of the logical 'and' operator, &&, in the `if` statement. The logical 'or' operator is the double vertical bar, ||. These logical operators compare the entire object before and after them. For example, two vectors that differ in only one place will return FALSE under the && operator. For elementwise comparisons, use the single & and | operators.

13.3 Maximum Likelihood Estimation

Now we consider code to find the likelihood estimator of the coefficients in a nonlinear model. Let us assume a normal distribution on the additive errors

$$y = aL^b K^c + \epsilon \tag{15}$$

Notice that the best way to solve this problem is a nonlinear least squares regression using `nls()`. We do the maximum likelihood estimation anyway. First we write a function that returns the log likelihood value (actually the negative of it, since minimization is more convenient) then we optimize using `nlm()`. Notice that `Y`, `L`, and `K` are vectors of data and `a`, `b`, and `c` are the parameters we wish to estimate.

```
> mloglik <- function(beta,Y,L,K){
+  n <- length(Y)
+  sum( (log(Y)-beta[1]-beta[2]*log(L)-beta[3]*log(K))^2 )/(2*beta[4]^2) + \
+  n/2*log(2*pi) + n*log(beta[4])
+ }
> mlem <- nlm(mloglik,c(1,.75,.25,.03),Y=Y,L=L,K=K)
```

13.4 Extracting Info From a Large File

Let `301328226.csv` be a large file (my test case was about 80 megabytes with 1.5 million lines). We want to extract the lines corresponding to put options and save information on price, strike price, date, and maturity date. The first few lines are as follows (data has been altered to protect the innocent)

```
date,exdate,cp_flag,strike_price,best_bid,best_offer,volume,impl_volatility,optionid,cfadj,ss_flag
04JAN1997,20JAN1997,C,500000,215.125,216.125,0,,12225289,1,0
04JAN1997,20JAN1997,P,500000,0,0.0625,0,,11080707,1,0
04JAN1997,20JAN1997,C,400000,115.375,116.375,0,,11858328,1,0
```

Reading this file on my (relatively slow) computer is all but impossible using `read.csv()`.

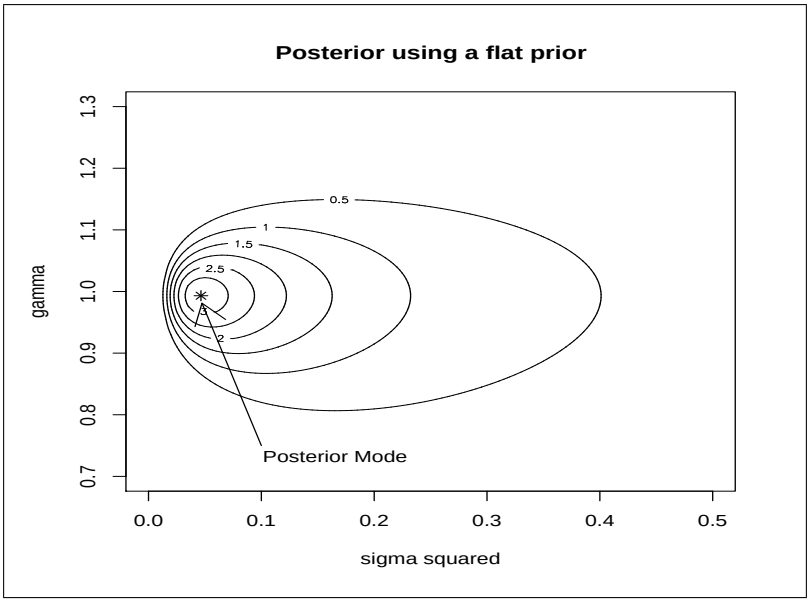
```
> LENGTH<-600000
> myformat<-list(date="",exdate="",cp="",strike=0,bid=0,ask=0,
+                volume=0,impvolat=0,id=0,cjadj=0,ss=0)
> date=character(LENGTH)
> exdate=character(LENGTH)
> price=numeric(LENGTH)
> strike=numeric(LENGTH)
> f<-file("301328226.csv")
> open(f,open="r")
> titles<-readLines(f,n=1) # skip the first line
> i<-1
> repeat{
+   b<-scan(f,what=myformat,sep=",",nlines=1,quiet=T)
+   if (length(b$date)==0) break
+   if (b$cp=="P"){
+     date[i]<-b$date
+     exdate[i]<-b$exdate
+     price[i]<-(b$bid+b$ask)/2
+     strike[i]<-b$strike
+     i<-i+1
+   }
+ }
> close(f)
```

This read took about 5 minutes. Notice that I created the vectors ahead of time in order to prevent having to reallocate every time we do a read. I had previously determined that there were fewer than 600000 puts in the file. The variable `i` tells me how many were actually used. If there were more than 600000, the program would still run, but it would reallocate the vectors at every iteration (which is very slow).

This probably could have been much speeded up by reading many rows at a time, and memory could have been saved by converting the date strings to dates using `as.Date()` at each iteration (see section 2.4). I welcome suggestions on improvements to this example.

13.5 Contour Plot

This code produces a contour plot of the function `posterior()`, which I had defined elsewhere.




```
> x <- seq(0,.5,.005)
> y <- seq(0.7,1.3,.005)
> output <- matrix(nrow=length(x),ncol=length(y))
> for(i in 1:length(x)) {
+   for(j in 1:length(y)) {
+     output[i,j] <- posterior(c(x[i],y[j]))
+   }
+ }
> contour(output,x=x,y=y,xlab="sigma squared",ylab="gamma",main="Posterior using a flat prior")
> points( 0.04647009 , 0.993137,pch=8)
> arrows(.1,.75,0.04647009,0.993137)
> text(.09,.73,"Posterior Mode",pos=4)
```