Econometrics in R

Grant V. Farnsworth[*]

Initial Version: April, 2004
This Version: March 24, 2014

[*]Althogh I originally wrote this document as an introduction to R for econometrics students, it is now primarily a personal reference. It's easier for me to search here for things I've picked up over the years than to figure them out again. Since I learned R largely by trial-and-error, there could be errors, inefficiencies, or omissions here. If you find something here suboptimal or have suggestions, please let me know. I can be contacted at my gmail email account. My username on that system is gvfarns.

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 Statistical Software?

R is much 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 packages online for use in many disciplines. As researchers develop new algorithms and processes, the corresponding packages 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 packages[1]obtained from the R website. To obtain a package 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 packages 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 packages 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* packages. 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.

[1]In R speak, a package is a collection of code—what we are typically interested in—and a library is the place (e.g., on your system) where packages are stored. Nevertheless we load packages using the `library()` command.

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
data.table	much faster version of data.frame operations (like rbind)
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
foreach	Loop-level parallel processing
foreign*	Loading and saving data from other programs
ggplot2	Graphics and plotting
gplm	Fixed and random effects panel regressions
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
plm	Fixed and random effects panel regressions
parallel*	Parallel processing
quadprog	Quadratic Programming Optimization
quantreg	Quantile Regressions
R.matlab	Read matlab data files
RMySQL	Interact with SQL databases
RODBC	Interact with SQL databases
rootSolve	find roots of vector-valued functions
ROracle	Interact with SQL databases
RPostgreSQL	Interact with SQL databases
RSQLite	Interact with SQL databases
sampleSelection	Heckman-type selection models
sandwich (and zoo)	Heteroskedasticity and autocorrelation robust covariance
sem	Two stage least squares
sqldf	operate on R dataframes using sql queries
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 packages, we will often have to load packages before working. Many of the common functions are in the *base*package, which is loaded by default. For access to any other function, however, we have to load the appropriate package.

```
> library(foreign)
```

will load the package 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.* Instead of `library()` we could also have used `require()`. The major difference between these functions is that `library()` returns an error if the requested package is not installed, wheras `require()` returns a warning. The latter function is generally used by functions that need to load packages, while the former is generally called by the user.

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 1x3 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 2.4.2).

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* package, 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 index using a vector of boolean (TRUE/FALSE) values of the same length as the dimentions of our R object in order to select only the TRUE elements. A common case would be the use of a comparison operator to extract certain columns or rows.

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

compares each entry in the first column of foo to 1 and inserts the row corresponding to each match into smallfoo. To check a set of values (a vector) against the values of another, we can use the intuitively named `%in%` operator. To repeat the above, pulling all rows of `foo` that have 2, 4, or 6 in the first column we could use

```
> otherfoo <- foo[foo[,1] %in% c(2,4,6),]
```

The `%in%` operator is often easier to use than `==` when indexing because the `==` operator returns `NA` when either side of the comparison is `NA`, and `NA` is not a valid index.

If we want to pull the first match of a particular value, rather than all of them, we can use the `match()` function. The first argument is the value to be looked up and the second is the object (typically a vector) to be searched. It returns the index of the first match or `NA` if it is not found.

```
> match(4,c(0,9,4,3,1))
[1] 3
```

Thus if we wanted to find the first row of `foo` that has 10 in the first column we could use

```
> firstfoo <- foo[match(foo[,1]<10,]
```

The `match()` function can also look up multiple values

```
> match(c(4,8,0),c(0,9,4,3,1))
[1] 3  NA  1
```

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 online. There are a lot of them.

2.2 Sorting Data

We can reorder data by one or more columns. If `wealth` is a dataframe with columns `year`, `country`, `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).

Notice that we can also sort by multiple columns using `order()`.

```
> wealth <- wealth[order(wealth$country,-wealth$year),]
```

Sorts the `wealth` dataset by `country` and then by `year`. We put a minus sign before `year` in order to sort in descending order. Notice that after a sort, the row names of the data frame contain the original row number of each row. We can access these row numbers or delete them if we want to using these two commands:

```
> originalorder <- row.names(wealth)
> row.names(wealth) <- NULL
```

2.3 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 9). 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.

2.4 Important Data Types

2.4.1 Atomic Data Types

R has the capability to have an unlimited number of data types. However, the most common data objects are aggregations of a few "atomic" types. These building block data elements include the following:

- *Logical.* Logical values can be `TRUE` or `FALSE`. We often use the abbreviated form `T` and `F` instead. If a logical variable is coerced into a numeric, `TRUE` will become 1 and `FALSE` will become 0. Thus `sum(c(T,F,T))` will evaluate to 2.

- *Integer.* Integers in R are most frequently used for indexing datasets or as counters in loops. That is, a number used in square brackets to reference elements of a vector or dataframe will be converted to an integer. Also using the colon operator to create a sequence of numbers will generate integers. However, assignment of a number to a variable by default creates a variable of type *numeric.* For example

  ```
  > j <- 0
  > for (i in 1:30){
  >   j <- j+1
  > }
  ```

  will result in an integer `i` with a value of 30 and a numeric `j` with a value of 30.0. Conversion from integer to numeric and back happens pretty transparently, so it doesn't get noticed by most users.

- *Numeric.* This is a double precision floating point number. Most numbers we deal with are of this type. Note that the function `as.numeric()` is actually a general test for whether a variable is numeric. That is, `as.numeric(j)` and `as.numeric(i)` from the example above will both return TRUE even though `i` is technically an *integer*, not a *numeric.*

- *Character.* A character string is of type *character.* Character strings can be compared using `==`, `>`, and `<`. The greater than and less than signs evaluate the strings alphabetically. For example `"cat"<"hat"` evaluates to `TRUE` but `"cat"<"bat"` evaluates to `FALSE`. Strings beginning with numbers and symbols come alphabetically before those beginning with letters. Also, if the strings are alphabetically the same except for case, the lower case letters are considered to come earlier. Actually, the most useful comparison operator is `==`. Notice also that to concatenate strings, we don't use the addition operator. Instead we use `cat()` or `paste()`.

- *Factor.* Factors are similar to character strings in that they look like strings of characters. However, internally they are represented by integers, with an accompanying table matching each integer to a value. For large vectors with comparatively few possible values, factors are much more efficient, both in speed and space. They are also more useful in many statistical operations. For example, you can use a vector of factors as an explanatory variable in a model and R will understand how to use them. Not so with vectors of character strings. **For this reason, R will convert character vectors to factors in many cases, including when we read in a text file and when we create a dataframe. Understanding this fact will clear up a good deal of confusion for the new R user.** There are options in many functions to override this behavior, and we can explicitly convert to and from factors using `as.factor()` and `as.character()`, respectively.

2.4.2 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`.

2.4.3 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.

2.4.4 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.

2.4.5 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 `$` symbol or by number using brackets. Referencing a single member of a list is generally done using double, not single, brackets (see section 2.1). We can think of the `[[` operator as being a slightly more flexible version of `$` (more flexible in that it works even if the elements are not named and can optionally perform partial matching on names). Both operators pull a single item from the list. To get a new list that is a subset of the original list, we use single brackets `[`. **Remember not to use double brackets to try and pull multiple elements from a list. Also remember that double brackets return an item from inside the list and single brackets return a list containing the desired element(s).**

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.

2.4.6 Functions

Yes, a function is a data object. Functions are created using the method described in section (10.1). The function, then, can be passed to other functions, copied, and modified, just like other data items. And the scope rules (whether the function is visible from a particular location in the code) are the same as for other data items.

2.4.7 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
```

2.4.8 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
```

2.5 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()`.

An alternative method of storing dates is using the `yearmon` format from the *zoo* package. a `yearmon` object works much like a `Date` but it does not have a day portion. I have also used `yearmon` to change monthly values to month-end values, since when we convert from `yearmon` to `Date` we specify where in the month we wish the date to occur, using the `frac` argument. For example

```
> NewDate<-as.Date(as.yearmon(myvariable,format="%b-%Y"),frac=1)
```

will create a date variable occuring at the end of the month represented by a string of the form `‘‘Jun-1999’’`. Setting `frac=0` would have used the first day of the month. We can perform a similar operation using `yearqtr()` to get the beginning or end of a calendar quarter.

2.6 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.

2.6.1 Using SQL Commands Directly on R Dataframes

In my experience, econometrics work can involve a lot of complex merging, and `merge()` is somewhat limited in its design. For example, you can only merge two datasets at a time, the merged database contains all columns from both inputs, and you can't merge on conditionals (only equality), among many others. In cases like these I do much of my data work in an SQL database and then inport relatively complete dataframes into R. However, it is often the case that a complex data join is desired on dataframes that are already in R and not worth sending into a database. For these type of merges and data manipuations, I use SQL commands directly on R dataframes using the package *sqldf.* Since the whole databases are retained in memeory using this method, merges are sometimes much faster than when they are done in a database.

The use of *sqldf* does not move the data from R into a database, so it affords no advantage specific to maintaining larger datasets than R can hold in the computer's RAM, but some data operations are much simpler and more intuitive in SQL than they would be in R[2]

Suppose we want to add the column `MktCap` from table `crsp` to the table `compustat` where the merge is on matching `permno` we want to match every compustat observation with every crsp observation that occurred before it.

```
> library(sqldf)
> sqldf("select a.*,b.MktCap from compustat left join crsp as b on a.permno=b.permno
         and a.gmonth>b.gmonth")
```

[2]Actually if the size of the intermediate tables is a concern for RAM, *sqldf* actually does provide an advantage: we can pass the parameter `dbname=tempfile()` to `sqldf()` and it will copy the input tables to a temporary database on disk and use that database–instead of precious RAM–for intermediate tables.

This would be quite a hassle using `merge()` but is trivial in SQL.

In this example I used gmonth, which is a variable I created to represent the number of months since January, 0000. It is an integer variable of my own creation that bypasses R and sqldf's date conventions.

Notice that *sqldf* tries to work well with R dates but sometimes is not successful. R dates internally are integers (representing the number of seconds since midnight on Jan 1, 1970). If you are modifying an R Date variable in *sqldf*, it will continue to be interpreted as a Date. If you create a new variable from a date variable, it will be numeric. You can convert back to a date variable in R using R's `as.Date()` function.

```
> firstdates <- sqldf('select FundID, min(Dates) as firstdate from mydata')
> firstdates$firstdate <- as.Date(as.numeric(firstdates$firstdate),"1970-01-01")
```

The use of `as.numeric()` is necessary because in this case sqldf converts the integer representation of the date to a string.

Simialarly, *sqldf* does not handle factors particularly intelligently. That is, a factor in R is an integer, where the integer can be looked up in the attributes to find the value. Factors are treated as integers in *sqldf*. As in other areas of R, I find factors to be a nuisance and resolve this problem by converting factors to character vectors before use in sqldf.

Another warning: sqldf is set up by default to use sqlite as its database. That works quite well. However, if you are also using *RMySQL* in the same R session, it will likely try to use MySQL for its operations. In my experience this doesn't work as well with *sqldf*. To force `sqldf()` to use sqlite, pass the parameter `drv="sqlite"` or set `options(sqldf.driver="SQLite")`. Alternatively, you can detach *RMySQL* by running `detach(package:RMySQL)` before the `sqldf()` command.

A final note about *sqldf*: any variable containing a period (.) in it will have that period translated to an underscore in *sqldf*. This is because the period is part of standard SQL syntax. Unfortunately the common convention in R (using period to separate words in a variable name) and SQL (using underscores to separate words in a variable name) are incompatible, so when working with *sqldf* we can just expect that some translation is necessary, or we can avoid both of these separators.

2.7 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.

2.8 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)
```

2.9 File Operations

Using the `system()` command is frequently not the best course because it limits the transportability of your code to machines running the same operating system and sometimes having the same third party software as you do.

R comes with some functions that manipulate files directly. `file.access()` tests whether a file exists and whether it can be read from and written to. `file.choose()` prompts the user for a file name to open or create. Also there are file operations called
tt file.copy(), `file.remove()`, and `file.rename()` which do what we expect them to. There are a few other `file.`-type commands as well. In addition to these, the `unlink()` command removes either files or directories, possibly recursively.

2.10 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")
```

2.11 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 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[3]. 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.

[3]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".

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()`.

3.1 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 14.4 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`.

3.2 Utilizing Unix Tools

If you are using R on a linux/unix machine[4] 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.

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.

[4]Thanks to Wayne Folta for these suggestions

3.3 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.

The functions in *filehash* allow us to create a database on disk and the interact with it as if it were an R environment. We can read, write, and modify data in this disk environment in the usual R way. The usual way of referencing R objects within an environment is using the `$` operator, similar to the way we interact with vectors in a dataframe.

```
> dbCreate("dump1.db")
> dbhandle<-dbInit("dump1.db")
> db1<-db2env(dbhandle)
> db1$myfirsttable <- read.table("large1.txt",header=T)
> db1$mysecondtable <- read.table("large2.txt",header=T)
> mergeddata <- merge(db1$myfirsttable, db2$mysecondtable, by=c("date","id"))
```

Now we have two dataframes inside the `db1` environment and we have the merged version of the two in RAM. If there were insufficient RAM for the merged data, we could have placed it inside the `db1` environment as well.

3.4 Using SQL Databases

A more elegant solution to the ones already mentioned is to keep huge datasets somewhere outside of R to do preliminary data work, then to use R only for the statistical analysis on the final (smaller) data. Large databases may be kept on an SQL server and queries may be sent to it from within R, and then data, merged, cleaned, and fully subsetted, may be imported into R. This sounds like a fairly complex process, but I have actually found it to be surprisingly simple. This is my preferred way of dealing with large data files[5].

Several implementations of SQL are supported by R. There is a direct interface to SQLite, PostgreSQL, Oracle, and MySQL. Microsoft SQL server and others may be reached using an ODBC connection available through the *RODBC* package, which is also the package used to import excel files directly, if you absolutely must do that.

On my system (Fedora Linux) a local MySQL server can be installed an started using the commands (at the Linux command prompt)

```
$  yum install mysql mysql-server
$  sudo service mysqld start
```

Since I am using this from my desktop I don't feel the need to set up accounts and passwords on my SQL server. I use the package *RMySQL* to connect to the MySQL server using the package *DBI*. I create a connection to a database which I will call "test" using

```
> dr<-dbDriver("MySQL")
> con<-dbConnect(dr,"test")
```

Now we can create a dataset (just an example) and save it to SQL

```
> A<-data.frame(a=rnorm(5000),b=runif(5000))
> dbWriteTable(con,"D1",A)
> dbGetQuery(con,"create table D2 as select * from D1 where a>0")
```

[5]Many econometricians perform data cleaning and basic operations using a language like SAS, export the data, and then read the cleaned data into R for the statistical analysis. This used to be my approach but I find the SQL approach cheaper and more convenient. Pretty much anything you might do with a datastep in SAS is faster and better implemented in SQL.

The *dbWriteTable* connection writes dataframe A as a table D1 in the databse. The next query creates a new dataset in the database called D2, which contains only the rows from D1 where the generated random number was positive. It returns *NULL*. Notice that if we had not specified "create table as" then this command would have returned a dataframe with the selected rows and columns. If we just want to read an entire SQL dataset into R, we could have used the `dbReadTable()` function instead.

Another thing to notice is that you can load data directly into the SQL server without creating and R table using `dbWriteTable()`. Just provide the csv file instead of a dataframe.

```
> dbWriteTable(con,"D1","./mybigfile.csv")
```

We can see what tables are available and what varaibles are in a table using (respectively)

```
> dbListTables(con)
[1] "D1" "D2"
> dbListFields(con,"D2")
[1] "row_names" "a"         "b"
```

When we are done using the SQL connection, we close it using

```
> dbDisconnect(con)
```

Of course this could have been done with a remote SQL server as well as one on this workstation, and the only difference would have been the arguments in the `dbConnect()` command.

Using an SQL server to keep, merge, filter, and perform basic manipulations on data before using the finished data in R can save a lot of time and headache. R is not really the right software for these functions when the datasets are very large, and the SQL interfaces allows us to do all this on the server where the data resides, before we get to the analysis part. Of course, there is some effort required in learning to use SQL well, but it's a comparatively simple and accessable language, and in my opinion the right tool for the job.

One thing to watch for when interacting with an SQL server from R is that variable types are typically not preserved. For example, a datetime variable in the SQL database will be imported as a vector of character strings in R, not as a vector of R Dates.

Notice, if it is the SQL functionlity (complex joins, etc.) that we seek and not its ability to maintain large databases, we can easily interact with R datasets directly in R using SQL commands by passing them to `sqldf`. See section (2.6.1).

There are a few gotchas when using MySQL from within R. For example, if you are loading string variables into the sql server and some of them contain tab characters, the operation will likely interpret those tabs at the end of the field and the field ordering will be off. It is a good idea to remove tabs and other special characters that MySQL may interpret as something other than text, if that is possible without compromising the data.

3.5 Changing Data from Wide to Long Format

3.5.1 Going from Long Format to Wide

R is generally set up to work with data in wide format. That is, if we have panel data, most R functions and programmers expect columns to represent the cross section and rows to represent the time dimension of the data. For example, we may have returns on 5 assets over time. In wide format there will be 6 columns (one indicating the date and 5 the return to each asset).

However, large databases, especially those coming from the SAS or SQL world, will typically store data in long format. The same data mentioned above would consist of three columns: the date, the asset in question, and the associated return. Changing from wide format to long and back is a fairly common operation with econometric and especially financial data.

The most convenient way to change from wide to long format or back in R is to use the `reshape()` function. Unfortunately, the writers of this function gave the arguments names that assume a format that is the opposite of what we usually want, and gave the arguments descriptive names, which normally serve only to confuse an econometrician. Specifically, they assume each row is an individual and the columns represent observations over time–normal economic data has a row for each time and a column for each individual.

Recognizing this naming issue and correcting for it, we can use reshape to quickly produce a wide version of long data. Suppose we start with a dataframe `h` in long format

```
> h
        date id      return
1 1990-10-01  a -0.14838438
2 1990-10-01  b -0.08231648
3 1990-10-01  c  0.26355277
4 1990-10-01  d -1.52749429
5 1990-10-01  e -0.11889179
6 1990-10-02  a  2.39816839
...
```

We would change to wide format using

```
> reshape(h,idvar='date',timevar='id',direction='wide')
         date   return.a    return.b   return.c    return.d   return.e
1  1990-10-01 -0.1483844 -0.08231648  0.26355277 -1.52749429 -0.1188918
6  1990-10-02  2.3981684 -0.56833820 -0.16736626  0.64116497  0.7249889
11 1990-10-03 -1.6494030 -0.86961966 -0.40232451 -0.85386200  1.8626587
...
```

Notice that `idvar` is the variable that designates each ROW in wide format, `timevar` is the variable that will define the new COLUMN names. Applying the argument names to the meaning of economic data is one of the biggest pitfalls in the use of `reshape()`. I have spent a lot of time debugging `reshape()` commands in which I have not been careful reverse the meaning of reshape argument names.

Notice that extraneous columns from the long format can be dropped by specifying their names in a vector of strings passed to the `drop` parameter.

When we create a dataframe using `reshape()` it also contains information about what the original dataframe looked like so we can take the dataframe back to its original shape by passing it to `reshape()` again with no arguments.

3.5.2 Going from Wide Format to Long

If we were starting with the wide format (and had not previously used `reshape()` with this dataset), we could use `reshape()` to change it to long format. We must again be careful about ignoring argument names. If we have a data set `w` of the form

```
      date          a           b           c           d          e
1990-10-01 -0.1483844 -0.08231648  0.26355277 -1.52749429 -0.1188918
1990-10-02  2.3981684 -0.56833820 -0.16736626  0.64116497  0.7249889
1990-10-03 -1.6494030 -0.86961966 -0.40232451 -0.85386200  1.8626587
...
```

we could create the long version of it using

```
> longh<-reshape(w,idvar='date',varying=names(w)[2:6],
+ v.names='return',timevar='firm',times=names(w)[2:6],direction='long')
> longh
                   date firm      return
1990-10-01.a 1990-10-01    a  -0.14838438
1990-10-02.a 1990-10-02    a   2.39816839
1990-10-03.a 1990-10-03    a  -1.64940297
1990-10-04.a 1990-10-04    a  -0.72549914
1990-10-05.a 1990-10-05    a  -0.69929495
...
```

The first column printed here consists of the row names, which we can ignore in this case. Notice that the `varying` parameter specifies which columns to use by column number (not the number IN the column name). Alternatively, one could pass a vector of column names. The function attempts to be clever about parsing the names but there are additional parameters that can be passed to help it if necessary. In particular we pass the column names again to the argument `times` to avoid getting numeric id's instead of names. Since the authors make assumptions about what the dimensions mean, we use the parameters `v.names` and `timevar` to give names to our columns. The default values (at least for `timevar`) are likely to be misleading. I typically strip out the row names (i.e., "1990-10-01.a") created by `reshape()`.

A side note: if the data comes in long format, the programmer will often want to apply functions to all the observations, by groups. The R method for doing this is described in section 10.3.3.

3.6 A Faster Reshape

Reshaping using the default package can be quite slow on large datasets and the syntax is not very intuitive. The *reshape2* package provides an alternative approach that is an order of magnitude faster and has a convenient (though I wouldn't describe it as intuitive) interface. In the context of *reshape2*, the word *melt* means to transform the data into long format. On the other hand *cast* means to transform long format data into wide format. When casting, variables are can be specified using formula notation. Our long-to-wide example from above (section 3.5.1) could have been written

```
> # Long to wide transformation
> dcast(h,date ~ id, value.var="Return")
```

Notice that we specified the variable that will define our rows on the left hand side of the formula and the one that will define our columns on the right hand side. If instead of `date` we had two identifying variables (say, `year` and `month`) we could have specified both using the addition sign

```
> # Long to wide transformation with two identifiers
> dcast(h,year + month ~ id, value.var="Return")
```

to get two idenfiying columns. There are a few versions of `cast()`, depending on the type of output we want. The function `dcast()` returns a dataframe and the help file for `cast.data.frame()` is the helpful one.

Our wide-to-long example above (section 3.5.2) could have been

```
> # Wide to long transformation
> melt(w,id = "date",variable.name="firm",value.name="return")
```

Note that `id` can be a vector of identifying variables. We could also specify which columns to use as values using the `measure.vars` parameter. Otherwise all available columns besides the id are used.

I have only illustrated the most basic and common uses of these functions. They can aggregate and cut the data in many powerful ways once you get the hang of them.

4 Cross Sectional Regression

4.1 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`salary` on various forms of `age` and `exper`. 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 functions such as `log()` and `sqrt()`. Interaction terms are specified using the colon between the two interacted data members. Another sytax for interaction terms uses the asterisk, but this has the additional meaning of including each term separately as well. Thus a synonym for the above regression would be

```
> result <- lm(salary ~ age*exper,data=byu)
```

The power term has a special meaning when specifying a regression model as well. It includes all base terms and (up to the power indicated) cross terms. Thus

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

includes `age`,`exper`, `ranking`, and the interaction terms between each two of these. If the power were 3, we would add the interaction between all three. In order to remove terms implied by one of these expansions, we can use the minus operator.

This syntax for model specification is potentially very useful but it causes confusion for many new users of R who would like to include power terms explicitly in their calculations. 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 (it just omits power terms whose first power is already included in the regression). There is no warning associated with this behavior so the programmer should 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)
```

4.2 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

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

4.3 Heteroskedasticity and Friends

4.3.1 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`.

4.3.2 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[6] we use the `hccm()` function (from the *car* package) 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()`.

4.4 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

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

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(0,0,0,1,1))
> 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

```
> 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"`.

4.5 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* package. 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.

4.6 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

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

```
> 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.

5 Special Regressions

5.1 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.

5.1.1 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 `i` identifies the individuals in the sample, then

```
> lm(y~factor(i)+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 by default when doing fixed effects estimation. Notice that if we were to manually compute these variables and use a standard linear model, the reported standard errors 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.

These calculations can be performed by hand in R, but it is easier to use the *plm* package, which does both fixed and random effects estimation and reports the appropriate standard errors. As the input it takes a standard dataframe. If you do not specify which variables are the individual and time indexes, it assumes the first two variables in the dataframe are these indexes. It also assumes that the dataframe is grouped by individual and sorted by time. Personally I always specify the index variables to avoid incorrect grouping. When specifying indexes, the individual identifier is the first index and the time identifier is the second. For example

```
> plm(y~x,data=mydataframe,index=c("i","t"),model="within")
```

performs a "within" estimation. We could instead use `model=‘‘fd’’` to calculate the first differences estimator[7]

Notice that we can also do panel data regression for generalized models (for example, a logistic or probit model) using the `pglm` package. The syntax for *pglm()* is an amalgam of the usage of *glm()* and *plm()* and is therefore an easy extension from either.

The authors of `plm` have created a very good writeup describing the various panel estimators (both fixed and random effects) and how to use them in a `plm` context, which can be accessed by typing

```
> vignette("plm",package="plm")
```

[7]For additional description of the `plm` package and fixed/random effects models, see *Croissant, Yves and Millo G. "Panel Data Econometrics in R: The plm Package." Journal of Statistical Software. July 2008. Vol. 27 (2).*

5.1.2 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

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

We can also use the `plm()` with `model=‘‘random’’` to do a random effects estimation.

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

5.2 Qualitative Response

5.2.1 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)
```

One thing to be aware of when using `glm()` and a link function is the way the `predict()` function works. If you use the default parameters, you get the predicted value in terms of the input to the link function. If you want a predicted value in terms of probabilities (the scale of the right hand variable) then you need to pass the parameter `type="response"` to `predict()`. Alternately, if we are using the probit link function, we can get the predicted values using the default method and then use `pnorm()` to translte them into estimated probabilities.

Measures of fit for logit and probit models are viewed with a skeptical eye in the statistics community. Nevertheless many reviewers want to see pseudo R-squared measures of these types of models. There are several possible measures we may call "pseudo R-squared." Probably the most popular and best justified is the McFadden measure

$$R^2_{McFadden} = 1 - \frac{\ln\hat{L}(M_{full})}{\ln\hat{L}(M_{null})}.$$

It is one minus the ratio of the log likelihood of the full model and the log likelihood of the null model (intercept only). There are several packages that compute this and other measures[8], but I find it simplest to note that in the usual case that the explanatory variable is always 0 or 1, the deviance reported by `glm()` is -2 times the log likelihood. Thus we can write a one-line function to return the McFadden pseudo R-squared

[8]Perhaps start with `pR2` from the *pscl* package

```
> R2 <- function(m) 1 - m$deviance/m$null.deviance
> mymodel <- glm(y~x1+x2,family=binomial(link=logit))
> R2(mymodel)
[1] 0.01469101
```

The second most common measure is the Cox and Snell measure

$$R^2_{CS} = 1 - \left(\frac{\hat{L}(M_{null})}{\hat{L}(M_{full})} \right)^{\frac{2}{N}},$$

which is similarly easy to compute by hand.

5.2.2 Multinomial Logit

There is a function for performing a multinomial logit estimation in the *nnet* package 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* package.

5.2.3 Ordered Logit/Probit

The *MASS* package 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")
```

5.3 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* package. 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.

5.4 Heckman-Type Selection Models

It is frequently the case that we have a dataset in which the dependent variable is endogenously missing. The classic example of this type of situation has `wage` as the dependent variable and a set of individual characteristics on the right hand. Because some people do not work, their wage is missing. However, we can take the full sample (employed and unemployed) to do a first stage probit regression where the dependent variable is `employed`. We can then take the fitted values (the stuff inside the probit function, not the fitted probabilities) and create the inverse mills ratio. We can then add the inverse mills ratio for individuals with jobs to the second stage regression, thereby correcting for the sample selection bias. This Heckman selection-bias correction is quite common in academic literature. There is also a one-step maximum likelihood method for solving it that has some better statistical properties.

The two-step procedure is easy enough to do manually (do a probit regression, get predicted values, create your own inverse mills ratio, etc.) but it is easy to get wrong. The *sampleSelection* package does this type of corrected regression as well as others of a similar type. The first argument is the function for the first stage probit (the dependent variable should be T/F where T means the observation will be included in the second regression). The second argument is the equation of interest. For example

```
> library(sampleSelection)
> corrected <- heckit(employed ~ age + educ + race, wage ~ educ + race + prev,data=G)
> corrected2 <- selection(employed ~ age + educ + race, wage ~ educ + race + prev,data=G,method="2step"
```

These two are equivalent. `Heckit()` does the most common case and defaults to the two-stage method while `selection()` has many other possibilities and defaults to the maximum likelihood estimator.

The manual equivalent of the above could be

```
> firststage <- glm(employed ~ age + educ + race,data=G,family=binomial(link=probit))
> predicted1 <- predict(firststage, G, type="link")
> G$invmills <- dnorm(predicted1)/pnorm(predicted1)
> secondstage <- lm(employed ~ educ + race + prev + invmills,data=G[G$empolyed,])
```

I often find it convenient to do this computation manually.

5.5 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.

5.6 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 package. 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.

5.7 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.

$$\begin{aligned} Y &= \frac{\epsilon}{1+e^{\beta_1 X_1+\beta_2 X_2}} && (5) \\ \log(Y) &= -\log(1+e^{\beta_1 X_1+\beta_2 X_2})+\log(\epsilon) && (6) \end{aligned}$$

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 package.

5.8 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* package. 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.

5.9 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* package. 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")
```

5.9.1 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)
```

5.9.2 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.

6 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.

6.1 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 $w_t = v_t - v_{t-2}$ 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 $\mathtt{y}_\mathtt{t}$, $\mathtt{x}_\mathtt{t}$, and $\mathtt{x}_{\mathtt{t}-3}$. 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 $\mathtt{y_t}$ on $\mathtt{x_{t-3}}$.

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.

6.2 Filters

6.2.1 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")
```

6.2.2 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.**

6.2.3 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.

The package *mFilter* contains `hpfilter()` which performs this filtration automatically. To pass a lambda value, we can specify `type="lambda"` and then pass `freq=1600` or whatever lambda value we would like to use.

6.2.4 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.

6.3 ARIMA/ARFIMA

The `arima()` command from the `ts()` package 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* package 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.

6.4 ARCH/GARCH

6.4.1 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* package. 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 p, 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)
```

6.4.2 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`.

6.4.3 Miscellaneous GARCH–Ox G@RCH

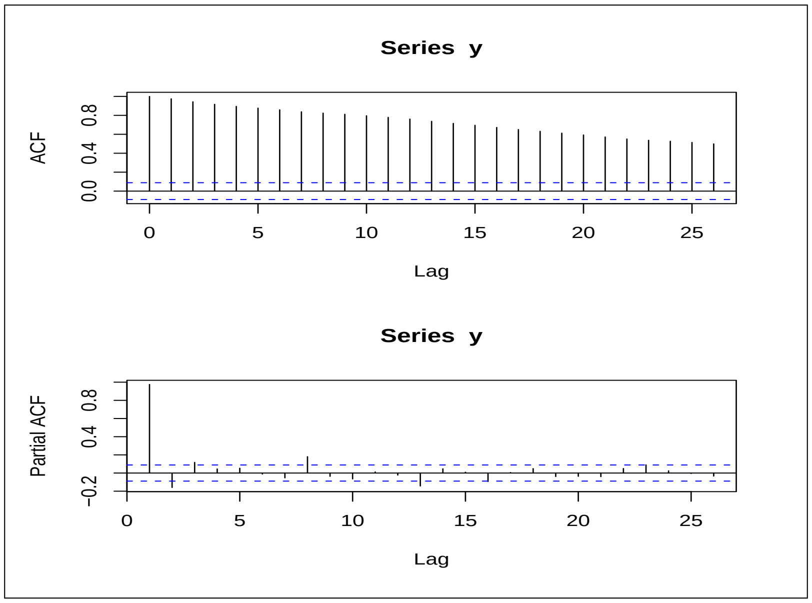
fSeries also provides an interface to some functions from the `Ox G@RCH`[9] software, which is not free in the same sense as R is, although as of this writing it was free for academic use. That interface provides routines for estimation of EGARCH, GJR, APARCH, IGARCH, FIGARCH, FIEGARCH, FIAPARCH and HYGARCH models, according to the documentation.

6.5 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* package 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 7.6.1 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`.



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

To plot confidence intervals that are based on Bartlett's standard errors, pass the keyword `ci.type="ma"` to `acf()`. This may be useful, for example, at the identification stage of a Box-Jenkins analysis when inspecting the ACF in order to identify what type of ARIMA model we might want to use. This keyword is documented in the help file for `plot.acf()`[10].

6.6 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.

6.7 Time Series Tests

6.7.1 Durbin-Watson Test for Autocorrelation

The Durbin-Watson test for autocorrelation can be administered using the `durbin.watson()` function from the *car* package. 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)
```

6.7.2 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* package.

```
> 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()` package as `bgtest()`. It takes a fitted *lm* object instead of a vector of data as an argument.

6.7.3 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* package for this test.

[10]Thanks to Graeme Walsh for pointing this issue out.

```
> 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
```

6.8 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* package 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.

7 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-like, 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)
```

7.1 Plotting Empirical Distributions

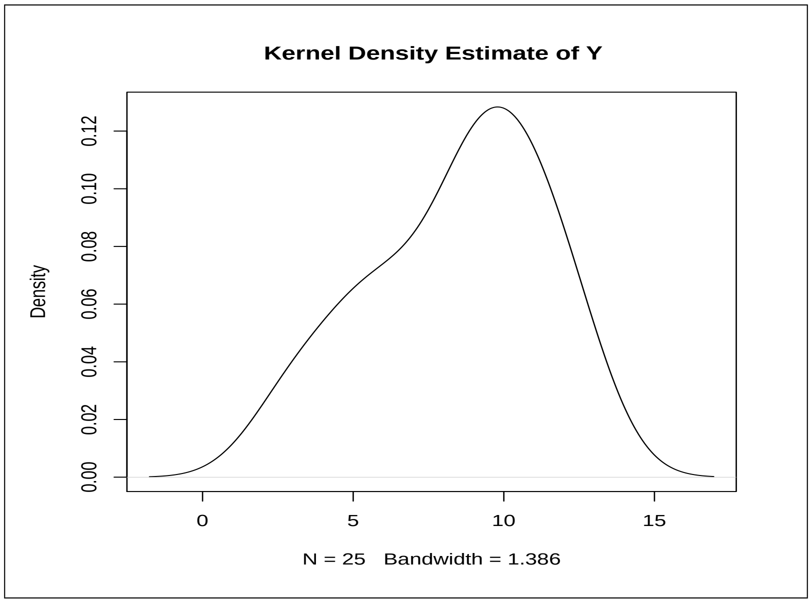
7.1.1 Histograms

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. This takes the raw data as an input and draws a histogram with minimal fuss. However, I find that it is not very flexible and, to my eyes, is not very pretty. For higher quality histograms, I frequently generate my own bins and counts and use the `barplot()` function. It is very flexible and makes beautiful bar graphs of many different styles

7.1.2 Kernel Density Estimates

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* package, 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")
```

7.2 Contour Plots

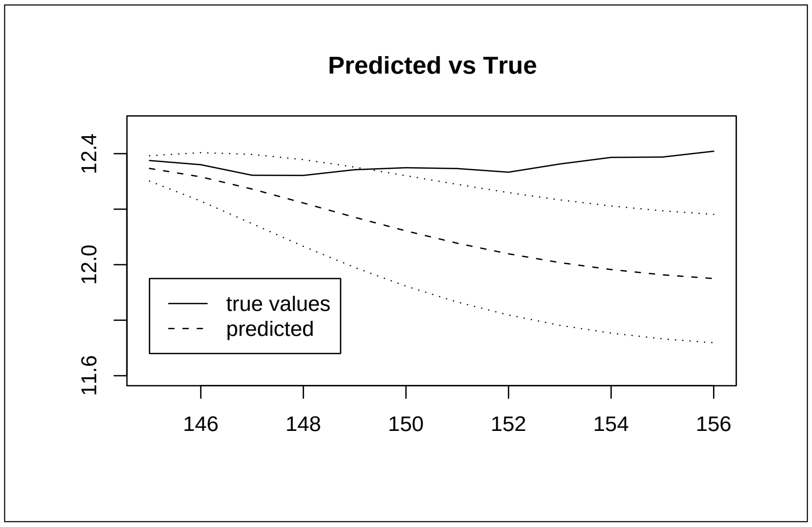
The command `contour()` from the *graphics* package takes a grid of function values and optionally two vectors indicating the x and y values of the grid and draws the contour lines. Contour lines can be added to another plot using the `contourLines()` function in a similar manner. The *lattice* package 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 14.5.

7.3 Adding a Legend

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))
```



To avoid the hassle of finding the right location in graph coordinate space we can use more descriptive words about where we want it, like `x='topleft'`. That would put the legend right in the topleft (with no space between it and the plot border). To make it more visually appealing we can bring it in a little by specifying something like `inset=.02`.

7.4 Adding Arrows, Text, and Markers

After drawing a plot of some type, we can add arrows using the `arrows()` function from the *graphics* package. 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 14.5.

7.5 Changing the Tick Marks

When you do a plot, by default R chooses tick marks and labels that it thinks are reasonable. In my view they generally are, although sometimes they are more spaced out than I think they should be. The easiest way to customize the tick marks is to do tell R not to put tick marks on with the plot command by passing `xaxt='n'` (or `yaxt='n'` if it is the y axis we wish to replace) and then afterwards calling the *axis()* command. For example, with 15 years of monthly data, R is likely not to put a tick mark by each year, so we use the *axis()* command

```
> ticktimes <- seq(from=as.Date('1995-01-01'),to=as.Date('2011-01-01'),by='year')
> plot(dates,y,xaxt='n')
> axis(side=1,at=ticktimes,labels=format(ticktimes,'%Y'))
```

7.6 Multiple Plots

7.6.1 Simple Grids

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 $\mathtt{c}(5, 4, 4, 2) + 0.1$. 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.

7.6.2 More Advanced Layouts

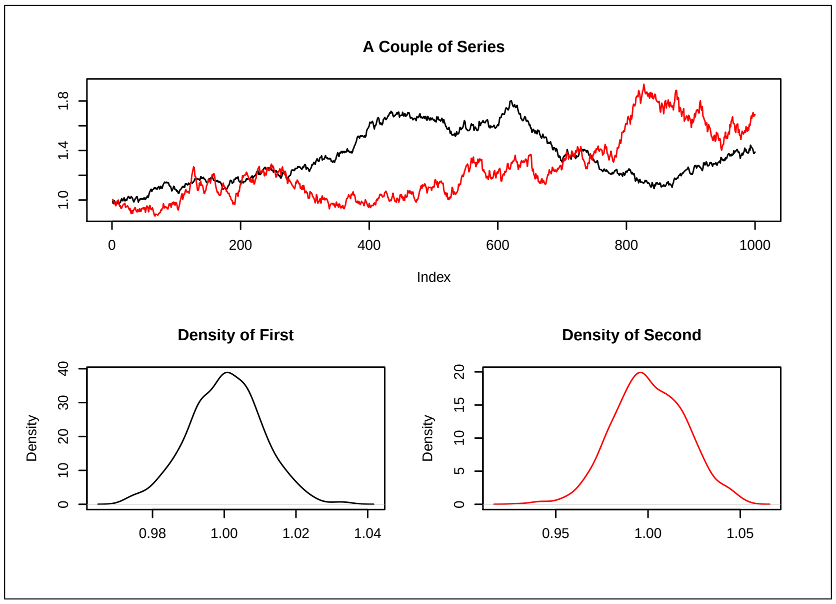
The *layout()* command treats the plotting area as a grid and each plot may span multiple columns or rows. The columns and rows also may optionally be different widths. To use it, first set up a matrix where each entry represents a location in the plotting grid and the number in the entry represents the plot that will be in that location. For example, to have a plot that spans horizonally and then two smaller plots side by side below it we could use

```
> layout(mat=rbind(c(1,1),c(2,3)))
```

and then execute each plot statement just as we did above

```
> data1 <- exp(rnorm(1000,sd=.01))
> data2 <- exp(rnorm(1000,sd=.02))
> range <- c(min(cumprod(data1),cumprod(data2)),max(cumprod(data1),cumprod(data2)))
> plot(cumprod(data1),type="l",main="A Couple of Series",ylab="",ylim=range)
> lines(cumprod(data2),col="red")
> plot(density(data1),main="Density of First",xlab="")
> plot(density(data2),main="Density of Second",xlab="",col="red")
```

Yielding the graph



In addition to controlling the width and height of plots by indicating which elements of the grid each plot contains, we can explicitly define the width and height of each column and row by passing the *width* and *height* parameters to *layout()*.

7.6.3 Overplotting: Multiple Different Plots on the Same Graph

It is simple to add two sets of data to the same plot using `plot()` and then, for example, `lines()`. In that case the X and Y axes of the plot are determined by the first `plot()` command and then subsequent additions are added on the same scale. Occasionally, though, we want to plot two data series with the same X values, but completely different Y scales. We may want to put tick marks for one Y scale on the right axis and tick marks for the other on the left axis. To do this, we do one complete plot and then do a second plot in the same space but with different Y limits. The `par()` command with key `new` allows us to do this. First we generate some example data:

```
> data1<-data.frame(date=seq(from=as.Date("1980-01-01"),
+  length.out=200,by="month"),Y1=cumprod(exp(rnorm(200)/100)))
> data2<-data.frame(date=seq(from=as.Date("1975-01-01"),
+  length.out=205,by="month"),Y2=100*cumprod(exp(rnorm(205)/100)))
```

Now we plot it

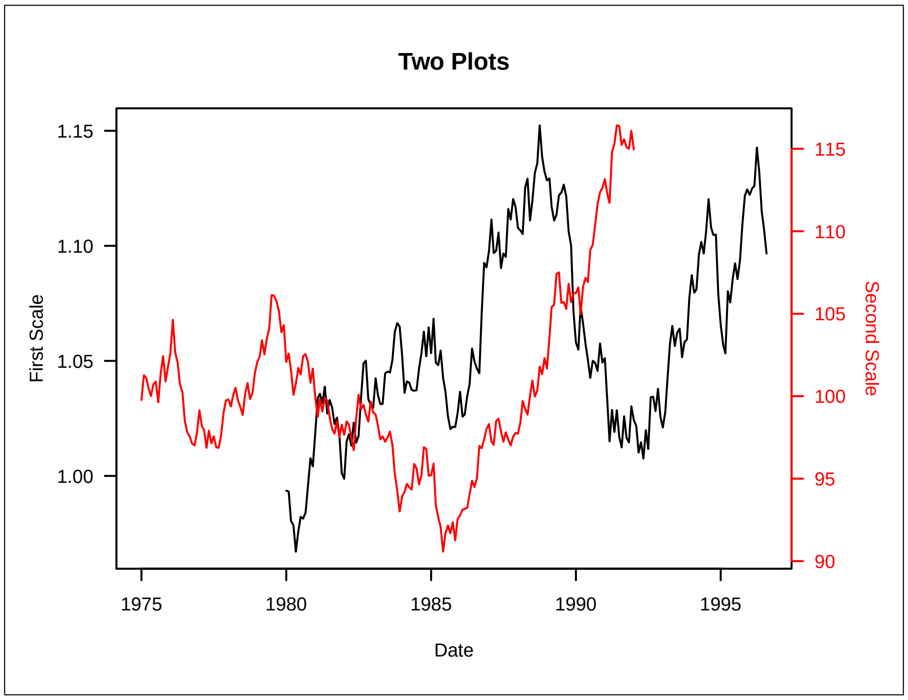
```
> mindate<-min(c(data1$date,data2$date))
> maxdate<-max(c(data1$date,data2$date))
> par(mar=c(5,4,4,4)+.1)
> par(las=1)
> plot(data1$date,data1$Y1,type="l",ylab="First Scale",xlab="Date",
+  xlim=c(mindate,maxdate),col="black",main="Two Plots")
> par(new=TRUE)
> plot(data2$date,data2$Y2,type="l",axes=F,ylab="",xlab="",main="",
+  xlim=c(mindate,maxdate),col="red")
> axis(side=4,col.axis="red",col="red")
> mtext("Second Scale",4,line=3,col="red")
```

Notice that we determined the maximum and minimum X values at the beginning so we could use it on both plots. Usually we are plotting one on top of the other, we want the X values to line up. If both dataset have exactly the same range of X this will probably work without manually setting up X limits, but if the dates differ, we need to set the X range ourselves. The first `par()` command modifies the margins so the right margin gets the same amount of space as the left. The second ensures that all tick labels are horizontal (otherwise the right axis labels read from bottom to top). The third `par()` command specifies that we will be writing the next plot, with a different scale, on top of the first. I disabled drawing of the axis and labels in the second plot so they did not draw on top of the first plot's labels. Then we manually add the right hand axis[11]. `mtext()` writes the right Y axis label on the right side. The `line` parameter specifies the placement (in number of text lines away from the axis line).

I have often wanted to rotate the right margin text 180 degrees so that it reads from top to bottom, but `mtext()`, unfortunately, does not support this. We can get around this limitation by using the `text()` command instead, with keywords `xpd=TRUE` and `srt=-90`. The former allows placement outside of the plot area and the latter rotates the text so it reads from top to bottom. In this case we must determine the correct plotting location ourselves. We can use the `par()` command to get the location of the axis. For example

```
> dims<-par("usr")
> text(dims[2]+(dims[2]-dims[1])*.1,(dims[3]+dims[4])/2,srt=-90,
+  labels="second scale",xpd=TRUE,col="red")
```

Here we computed a location that would be in the middle of the plot in the vertical direction and 10% to the right of the edge. The second ratio may need to change if the font size changes via the `cex` option in `par()`, which I often use when saving a plot to a file. It's a bit of manual work, but we can get the desired objective: a right hand legend that reads from top to bottom, located right where we want it.



Using this method we can put any type of plot on top of any other type of plot and tweak the placement and appearance of every detail of the resulting composite plot. This is an example of the flexibility and power of R's plotting capability, but also an example of how R makes you read the help pages and do a little more work by hand than some other environments.

[11]Notice that in R each axis is numbered: 1 is bottom, 2 is left, 3 is top, 4 is right. This convention is used in many different plotting functions.

7.7 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()`. Another thing to notice is that when outputting to pdf, the default behavior is to generate a square graph and center it on the page. With letterpaper the usual default size is 7 inches square. The *width* and *height* parmeters override that behavior, which is why we include them even though the dimensions of "letter" paper are already known.

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.

7.8 Fixing Font and Symbol Size in Pdfs

When R draws a plot to the screen the fonts, symbols, and other plot characteristics are typically nicely sized. Yet occasionally when drawing to a device that has a different resolution than the screen, such as a pdf file, the sizes are not so nice. In my experience, drawing a full page pdf corresponding to a graph that looks nice when drawn to the screen will result in very large fonts and symbols. The general size of these marks is scaled by the `cex` parameter, using the `par()` command. The commands

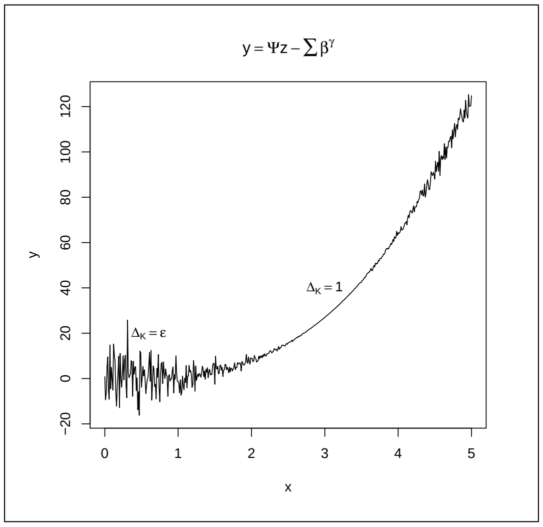
```
pdf("myplot.pdf",paper="letter")
par(cex=.75)
plot(x,y,main="Sample Plot")
dev.off()
```

are more likely to give pleasing results in my experience. The smaller the `cex` parameter the smaller the fonts and symbols in the resulting graph. Again, remember that parameters set by `par()` should be set **after** the device (pdf in this case) is initialized. Otherwise they apply to whatever device was being drawn to before that.

7.9 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])))
```

7.10 Changing the Font in Plots

The default font used for the title, labels, and any other words on a plot is a fairly plain sans-serif font. I like to use a font that looks a little more like my body text. You can change the font family by passing the keyword *family="serif"* or *family="mono"*. Alternately, the font family can be specified using the command *par(family="serif")*. Similarly, they keyword **font** to either the *plot()* or *par()* functions changes the face from normal to bold or italics.

A more flexible font family is the Hershey family. Hershey fonts are built into R as a system of vectors that R can draw easily in any size or rotation. This font also has a lot of special characters (such as math and greek symbols) built in. Run

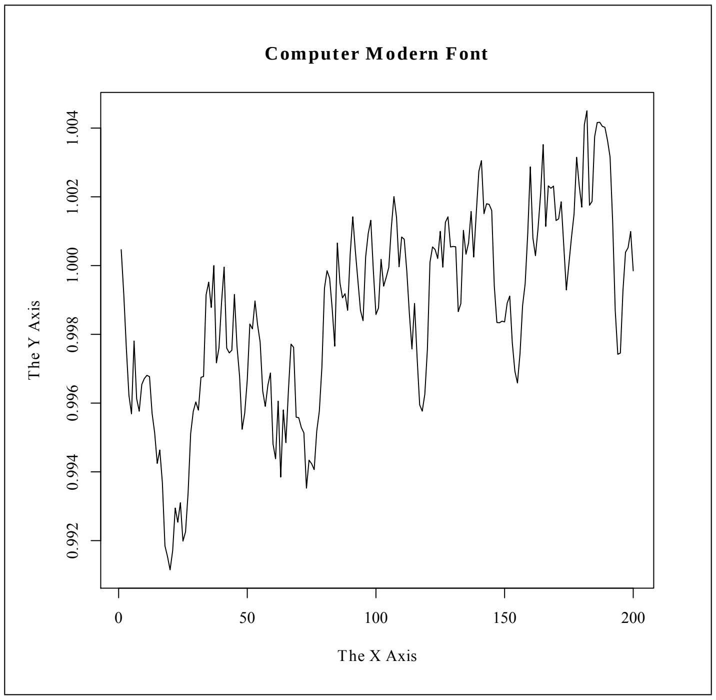
```
> demo(Hershey)
```

to see some examples.

R can also load up arbitrary fonts, such as TrueType fonts, but the procedure is a little more involved. One has to create the font within R. I use this functionality only for using the Computer Modern font, which is the default font in the LaTeXdocuments I write (like the one you are reading right now). One thing to be aware of is that some fonts work only with certain devices. Computer Modern, for example, works with the pdf and postscript drivers only—not to the screen. I obtained the five necessary font files online[12], copied them into my R directory, and executed

```
> CM <- Type1Font("CM", c("./fcmr8a.afm", "./fcmb8a.afm",
+       "./fcmri8a.afm", "./fcmbi8a.afm", "./cmsyase.afm"))
> pdf("testpdf.pdf", family=CM)
> plot(1:200,cumprod(1+rnorm(200,sd=.001)),main="Computer Modern Font",
+       xlab="The X Axis",ylab="The Y Axis",type='l')
> dev.off()
> embedFonts("testpdf.pdf",out="fixedpdf.pdf")
```

Notice that I needed to use *embedFonts()* because R does not embed fonts in the pdf. Not a problem if it's one of the standard 14 fonts that all pdf viewers should support, but Computer Modern is not one of those.



It is a little unusual to use Computer Modern in a big bold font as you see in this title of this plot, but the axis labels should match a normal Computer Modern text.

7.11 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 packages in R which might prove useful, including *grid*, *lattice*, *and ggplot2*.

[12]I learned how to do this at https://www.stat.auckland.ac.nz/~paul/RGraphics/rgraphics.html

8 Statistics

R has extensive statistical functionality. The functions `mean()`, `sd()`, `min()`, `max()`, and `var()` operate on data as we would expect[13]. 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()`.

8.1 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.

```
> 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

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

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*.

8.2 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.

8.3 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.

9 Math in R

9.1 Matrix Operations

9.1.1 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()`.

9.1.2 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.

9.2 Numerical Optimization

9.2.1 General 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.

9.2.2 General 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. Notice that U_i and C_i have a number of rows equal to the number of constraints you will use and U_i has a number of columns equal to the number of parameters to be estimated. Equality constraints should be built into the objective function.

A simple example: T o 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. Notice that in this example m is a 5-vector. Actually `ui` and `ci` should be appropriately sized data members of type *matrix*.

9.3 Quadratic Programming

A large number of constrained optimization problems that we might seek to solve can be written as the minimization of a quadratic function of several variables subject to linear (equality or inequality) constraints. If the optimization problem can be written this way, it is much more efficient and reliable to use an optimizer that specializes in this class of problems. These 'quadratic programming' optimizers are available in the *quadprog* package.

The package assumes that the problem can be written

$$\min -d^T b + \frac{1}{2} b^T D b$$

subject to

$$A^T b >= b_0.$$

The choice variable (the vector of parameters the optimizer will find) here is b. When we set up the problem, we should specify equality constraints at the top of the A^T matrix and inequalities after. The parameter `meq` specifies how many of the constraints are equalities.

As an example, we solve a portfolio optimization problem. If `SIGMA` is a covariance matrix of available assets and `ER` is the vector of expected returns, we can solve for the optimal portfolio weights for a portfolio with expected return .04 using the following code

```
> w <- solve.QP(Dmat=SIGMA, dvec=rep(0,NROW(SIGMA)), Amat=cbind(ER,1),
                bvec=c(.04,1), meq=2)$solution
```

Notice that we set d to be a vector of zeros because we are just minimizing the variance. The A matrix has the expected returns as the first column and a vector of ones as the second. Notice that the A matrix is transposed in the description of the problem. For b_0 the first entry is our expected return constraint and the second is the constraint that the sum of the weights is unity.

A second example disallows short sales. Here we set up a constraint for each variable requiring positivity.

```
> eye <- diag(1,NROW(SIGMA))
> w2 <- solve.QP(Dmat=SIGMA, dvec=rep(0,NROW(SIGMA)), Amat=cbind(ER,1,eye),
                bvec=c(.04,1,rep(0,NROW(SIGMA))), meq=2)$solution
```

Notice that `meq` is still 2 since the positivity constraints are all inequalities, whereas the first two constraints are equalities.

Notice that we have actually minimized half the variance here. This gives the correct solution for the portfolio weights. However, the output from the optimizer includes a variable called `value`, which in this case is only half the variance.

9.4 Root Finding

I have often had opportunity to need to find the root of a nonlinear function of one or more variables. One method for accomplishing this is to minimize the function squared (or the norm of the function, in the multivariable case), but it is more efficient to search for a zero (known value) than a minimum (unknown value). For this purpose, R provides `uniroot` in the univariate case. We pass the function and a set of starting values and it returns the parameters at which the root occurs. For the vector-valued function case, the package *rootSolve* provide `multiroot`, which works the same way.

9.5 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* package, which does not allow infinite bounds.

10 Programming

10.1 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. Section 10.12 has ideas on how to deal with situations where you might be tempted to use it.

10.2 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).

10.3 Avoiding Loops

10.3.1 Using Vector Math (Implicit Loops)

R is a vector-oriented language and it is therefore much faster to use the implicit loop of vectors than to write an explicit loop. For example if `A`, `B`, and `C` are large vectors, the following is fast and clear

```
> Z <- A*B/sqrt(C)
```

while the explicit version

```
> Z <- numeric(length(A)
> for (i in 1:length(A)){
>   Z[i] <- A[i]*B[i]/sqrt(C[i])
> }
```

is much slower, longer, and less clear. One of the skills an experienced R programmer must learn is to write functions that use the implicit loop as much as possible. When the datasets get large this speedup effect is very pronounced.

10.3.2 Applying a Function to an Array, List, or Vector

To help the programmer avoid unsightly or unclear loops[14], 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()`. There are several other apply-like functions in R that the interested programmer can learn to use.

Function	Input	Output
`apply()`	matrix	vector
`lapply()`	list or vector	list
`sapply()`	list or vector	vector
`vapply()`	list or vector	vector
`mapply()`	multiple equal-sized objects	vector
`tapply()`	subsets of a vector	vector
`by()`	subsets of a dataframe	vector

Note: In most cirumstances calling code from one of these apply-like functions doesn't generally save much time over explicit looping. These functions faster than loops but not by an order of magnitude. They should be called for parsimony or clarity. By a small margin, the fastest of these functions is `lapply()`—the others typically call it and then simplify the results. It's also the most general. Actually technically `vapply()` is even faster on equivalent problems because you specify the return type, which speeds things up. Apply-like functions differ from loops primarily in that one iteration cannot affect the others (wheras loops allow things like counters and cumulations).

A very, very good reason to use these functions is that if you call the time-consuming bits of your code with these functions, you can easily replace them with the equivalent parallel-processing versions (e.g., `mclapply()` or `parLapply()`) and save some real time on a multiprocessor machine. Section 10.11 has more details. The limitations of these functions correspond very nicely to those imposed by simple parallel processing problems, so they are a natural candidate for simple parallelization.

10.3.3 Applying a Function to By-Groups

Programmers coming from the SAS or SQL world will find themselves looking for a convenient way to apply a function to all the observations, by group, as is often done when working with data in long format (see section 3.5). A very simple example would be applying the `prod()` function to a set of daily returns to get

[14] but not necessarily the performance penalty loops impose. Read on.

the monthly returns. Suppose we have a dataset tracker with columns date and r where date is the date of the observation and r is the daily return for that date (gross returns, meaning 1+return). We would like to generate a dataset of monthly returns without using a cumbersome loop structure. In this case it is very simple

```
> dmonthly<-aggregate(tracker$r,by=list(ym=format(tracker$date,'%Y%m')),FUN=prod)
```

Notice that the arguments are the data to which we will apply the function, the grouping list (it must be specified as a list, and may contain more than one level of by group), and the function we wish to apply. For this example I chose to create a new variable ym, which is a string that serves to index each month. If we have a variable already created for this purpose, it may be clearer to the reader.

The output will be a dataset containing two columns: ym, our monthly index, and x, which contains the corresponding monthly return.

This construction may seem like a very specific or uncommon case to the inexperienced programmer, but to anyone coming from the record-oriented programming world (which much of finance is, for example) it comes up rather frequently.

Notice that aggregate() produces a vector equal in length to the number of by-groups there are. In some cases, however, we want the result to have the the same number of observations as the original vector. For example, if we want to find the rank of one variable when grouped by another. For these cases I use ave(). To rank returns (r)to various funds by month (ym) we could use

```
> FundReturns$MonthRank <- ave(FundReturns$r,FundReturns$ym,FUN=rank)
```

10.3.4 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.

10.4 Conditionals

10.4.1 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[15].

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)
`any(x)`	TRUE if any element of x is true
`all(x)`	TRUE if all elements of x are true

10.4.2 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.

10.5 The Ternary Operator

Since code segments of the form

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

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.

10.6 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

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

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. Notice that if one of the arguments to the `paste()` command is a vector of strings, then by default the result will be a vector of strings in which each element of the output string is the whole argument to `paste()`, with one element of the input vector inserted. To make a single string out of a vector of strings, use the `collapse` keyword instead of the `sep` keyword when defining how you want `paste` to separate its inputs.

10.7 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()`.

10.8 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
    user   system  elapsed
   0.152    0.016 1091.586
```

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)[3]
elapsed
1091.586
```

The *proc.time()* interface actually gives us information on five types of time, although the associated *print()* routine only shows three. In general I find it most convenient to look at the third number produced by proc time, which is the amount of actual time that has elapsed. The amount of CPU time used, for example, can be deceptive if R sends queries to a database that takes a long time to run. If we wanted to know how much computation time was actually used by R, we would look at the sum of the first and fourth entries (user time plus user time of child processes).

10.9 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[16]. 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.

10.9.1 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.

10.9.2 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
```

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
```

[16]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.

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()`.

10.10 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.

10.11 Parallel Computing

As desktops and workstations become more capable, they have an increasing ability for parallel processing: multiple processors, multiple cores, hyperthreading, and clustered computers. R is natively a single-threaded language and so that during a long computation, one processor is likely to be near 100% utilized while the others are idle. There are a number of packages that allow R to spread a computation across multiple processors. Since the release of R 2.14.0, R comes with a new base package called *parallel.* This package includes most of the functionality from the *multicore*, which provides simple functionality for parallelizing on a single machine, and *snow*, which provides for parallelization on a cluster—possibly over multiple machines.

In my experience, the most useful kind of multiprocessing for interpreted languages (also the one most available) is high-level parallelization for calling a single time-consuming function many times with different arguments that do not depend on the output of the other function calls. This is an example of what computer scientists call an “embarrassingly parallel” problem.

One of the easiest ways to implement parallelization is to write code in such a way that the parallelizable part is in a function of its own and is called from within one of the `apply()` functions. Specifically, I use `lapply()`, which takes a list as its argument and returns a list containing the results from function call with no dependence between one function call and the next[17]. When the code works, we can then replace `lapply()` with `mclapply()`, specifying how many processors to use and we are done. The catch is that `mclapply()` works only on POSIX-type computer (Linux, Unix, Mac), not windows. On windows it simply calls `lapply()`.

Besides `mc.cores`, which specifies how many processors to use at a time, there is one important `mclapply()` option: `mc.preschedule`. By default this is false, which means one thread is created for each processor and the work is split up between the threads before any computations are done. This works well if there are many elements in the list and each takes about the same amount of time to run. The alternative is to set `mc.prescedule` to true, which means each item in the list to be executed is placed in its own thread, one at a time, and given to the next available processor. This is called load-balancing and works well if there are relatively few elements of the list (creating these threads has significant overhead) and if the function calls vary widely in how long they take to run. In other words, setting this to true avoids the situation where all processors except one finish early and then you have to wait for the one to do a long job, but it incurs greater overhead overall. Notice that functions called by `mclapply()` can see upper-level variables just as functions called in a normal context can and are generally able to print to the screen as well. When using other packages for parallelization this may not be the case.

A simple example:

```
> M <- 5
> a <- 1:10
> f <- function(x){cat(x+M,' '); return(x+M)}
> out <- mclapply(a,FUN=f,mc.cores=4)
6  10  14  7  11  15  8  12  9  13
> unlist(out)
 [1]  6  7  8  9 10 11 12 13 14 15
```

Notice that `cat()` output shows the order in which the computations were done (out of order because of the way it was spread across processors) but they are in order in the output list. Also notice that `M` was read by each child thread from the higher-level environment without a problem. Beacuse I did not explicitly set `mc.preschedule` to false, no load balancing was done, which is a good idea for this case.

When working in windows (or across multiple machines), `parallel` provides the functionality that was in `snow`, which is slightly more cumbersome to write and slower to execute. Basically one has to call `makeCluster()` before the parallel code and `stopCluster()` afterward. Instead of `mclapply()` we use `parLapply()`. Any variables not passed to the function called by `parLapply()` must be exported into it using `clusterExport()`.

Within *parallel* and in several other packages there are other methods for parallelizing R code (for example, the `foreach` package has convenient loop-level parallelization).

When writing parallel programming we must be careful about the scoping of our variables. We must treat each function evaluation as if it was executed on a different thread, so it should not modify any top-level variables. Modifying variables outside the scope of a function is poor programming practice in general anyway.

Another point to remember is that when using parallel processing for any code that uses random number generation, it is important to seed the random number generator for each thread with a different value. Otherwise all threads may compute the same “random” values. There are functions within `parallel` and other packages to do this.

[17]If we need to parallelize in such a way that we must pass multiple varying arguments to our function each time it is called, we can create a list of lists. Each interior list contains the named arguments we would pass our function in one iteration. Then we write a wrapper that calls our function using `do.call()`. The function `do.call()` passes the elements of a list as arguments to a function.

Note: I frequently find myself using `mclapply()` and getting back a list of dataframes (with the same column names) which I then want to row-bind into a single dataframe. The usual way to do this is

```
> MyDataFrame <- do.call(rbind,MyList)
```

but this calls `rbind()` once for each element of the list, making copies of the data over and over, so it is very slow—sometimes impossibly slow. To do the same thing quickly we can use `rbindlist()` from the *data.table* package.

```
> MyDataFrame <- rbindlist(MyList)
```

The latter example is several orders of magnitude faster than the former. Since parallel processing is typically used when there is a lot of data to consider, I find myself needing to use `rbindlist()` almost every time.

10.12 Environments and Scope

Environments are a very complex subject in R so I will stick to what I have found useful and leave more in-depth investigation to the interested reader. Though there is some disagreement on the right terminology, I'll use "environment" to represent the R container that holds the set of variables that are available for reading *and* writing at a particular point in the execution of an R program.

When the program starts, variables created in open code are stored in the global environment. When a function is called a new, empty environment is created (a "child" of the global environment) and any variables created within the function are stored in that environment. If the function calls another function, yet another environment is created. Since essentially everything that happens in R is a function call, environments are constantly being created. When a function terminates, the environment created for it disappears along with any variables created within it. This is common in functional programming languages.

Creation and modification of variables can only happen within the current environment. However, data in parent (and grandparent, etc.) environments is available for reading without further effort. The rules governing this are referred to as *lexical scope*, meaning if R can't find a referenced variable it looks for that variable in the parent environment, then the parent above that, etc. For example,

```
> addone <- function(){
+  return(a + 1)
+ }
> a <- 3
> addone()
[1]  4
```

Notice that this works even though `a` is not being passed to `addone()`. R searches the local environment and doesn't find it, so it searches the parent, finds it, does the addition and returns the expected result. This works if we are just reading from the variable `a`, but we can't modify it directly. Notice that if we modify `a` within the function, a new variable is created in the local environment, which is a modified version of the `a` above it. But the modified version is not retained after the function ends.

```
> addoneprint <- function(){
+  a <- a+1
+  cat(a,"\n")
+ }
> a <- 3
> addoneprint()
[1] 4
> a
[1] 3
```

Remembering that R traverses parent environments to read but that you can't write to them is a key habit for the programmer to keep in mind. There are ways around this restriction. For example, one could use the superassignment operator, `<<-`.

```
> addoneprint <- function(){
+  a <<- a+1
+  cat(a,"\n")
+ }
> a <- 3
> addoneprint()
[1] 4
> a
[1] 4
```

Use of the superassignment operator can easily lead to subtle unexpected results and is almost universally discouraged. Instead, create a copy of the data in the local environment and return it at the end of the function, then overwrite the version in the parent environment. If the data is very unwieldy, consider a strategy to pass by reference.

10.12.1 Passing by Reference

In many programs there is a large dataset that the programmer wants to modify but copying it over and over each time a function is called can impose a large performance or memory penalty. I have seen (and used) a number of ways around this restriction, some of which are very awkward. The solution we need is a method for passing by reference: allowing a function to directly modify data from a parent environment without making a copy.

In R we can create an empty environment, store variables within it, and pass it to functions (which does not make copies of the contained variables). This is how I deal with data that I do not want to make a copy of.

```
> fiveit <- function(thisenv){
>   thisenv$a[3] <- 5
> }
> myenv <- new.env()
> myenv$a <- c(1,4,7)
> fiveit(myenv)
> myenv$a
[1] 1 4 5
```

There's a little bit of hassle associated with referring to the environment every time we use data stored inside it, but it's easier and safer then most alternatives. If the dataset is of moderate size, passing by value and overwriting is often easier. Also note that you can pass large datasets to R functions by value without the overhead of making a copy as long as you do not modify these datasets within the function. That is, R treats passed data as copy-on-write.

11 Changing Configurations

11.1 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)
```

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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.

12 Saving Your Work

12.1 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")
```

The `save()` command preserves the same of the object or objects that you save. When you load them back the names are kept. This is convenient when saving all the objects associated with a project, but it can cause problems (like overwriting existing objects of the same name). As an alternative, we can save a single object (but not its name) using `saveRDS()`. Then we load it back using `readRDS()`.

```
> saveRDS(BYU,"BYU.rds")
> BYU2 <- readRDS("BYU.rds")
```

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 2.7.

12.1.1 A Note About Scientific Notation

Note that sometimes R will print numbers in scientific notation, either to the screen, or into a .csv file when using `write.table()`, which can sometimes be a problem. There are several ways to control this behavior. The easiest is to set the `scipen` option. When set to 0 (the default) R decides when to use scientific notation—it chooses scientific notation if that is shorter than fixed notation. If we set `scipen` to a positive number, scientific notation will only be chosen if it is shorter by that many digits.

```
> options(scipen=10)  # discourage scientific notation
> write.table(BYU,file='BYUINFO.csv',sep=',',row.names=F)
> options(scipen=0)   # return to default setting
```

12.2 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()[18].

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()`.

```
> 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 7.7.

12.3 Saving as LaTeX

R objects can also be saved as LaTeX tables using the `latex()` command from the *Hmisc* package. 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[19]

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	$\Pr(>\lvert t \rvert)$
(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* package.

13 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.

[18]Thanks to Trevor Davis for this observation.

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

14 Appendix: Code Examples

14.1 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)
```

14.2 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.

14.3 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)
```

14.4 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()`.

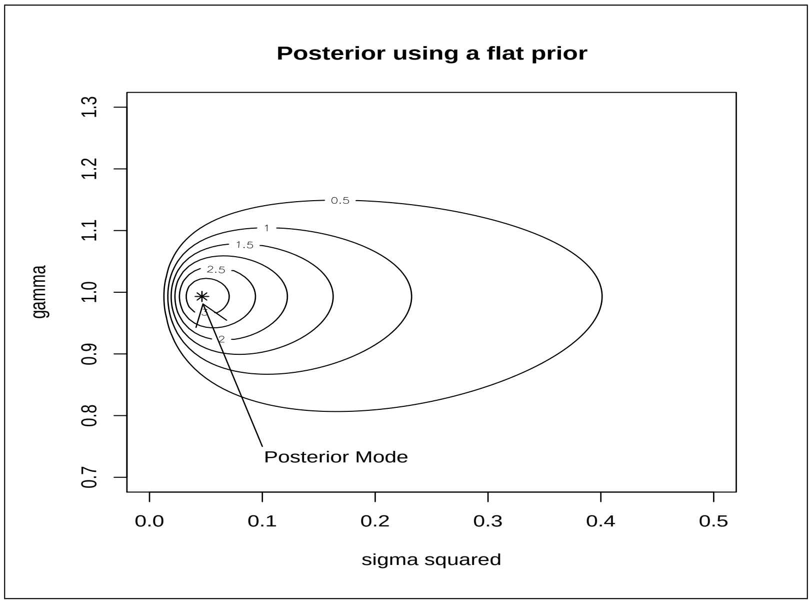
```
> LENGTH<-600000
> myformat<-list(date="",exdate="",cp="",strike=0,bid=0,ask=0,
+                volume=0,impvolat=0,id=0,cjadj=0,ss=0)
> date=character(LENGTH)
> exdate=character(LENGTH)
> price=numeric(LENGTH)
> strike=numeric(LENGTH)
> f<-file("301328226.csv")
> open(f,open="r")
> titles<-readLines(f,n=1) # 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 2.5). I welcome suggestions on improvements to this example.

14.5 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)
```