

Econometrics in R

Grant V. Farnsworth*

March 19, 2005

Abstract.

This document introduces the student to the syntax and principles necessary to use R for basic econometric analysis. It focuses on answering the types of questions teaching assistants get all the time, rather than trying to be a self-contained tutorial or reference. The information here is geared for typical problems faced by undergraduate and first year graduate students in economics and is by no means an exhaustive introduction to R. Where commands are merely mentioned, the student is expected to read the corresponding help file to learn how to use the various options. For more thorough and general documentation, consult the R website.

*This paper was originally written as part of a teaching assistantship and has subsequently become a personal reference. I learned most of this stuff by trial and error, so it may contain inefficiencies, inaccuracies, or incomplete explanations. If you find something here suboptimal or have suggestions, please let me know. Until at least 2008 I can be contacted at *g dash farnsworth at kellogg.northwestern.edu*.

Contents

1	Are You Ready for R?	3
1.1	What is R?	3
1.2	How is R Better Than Other Packages?	3
1.3	Obtaining R	3
1.4	How R Works	4
1.5	Getting Help	5
2	Data in R	5
2.1	The Basics	5
2.1.1	Dataframes	7
2.2	Opening a data file	7
3	Cross Sectional Regression	8
3.1	Ordinary Least Squares	8
3.2	Extracting What You Need	9
3.3	Heteroskedasticity and Friends	9
3.3.1	Heteroskedasticity Consistent Covariance Matrix	9
3.3.2	Generalized Least Squares	9
4	Special Regression	9
4.1	Models With Factors/Groups	9
4.2	Logit/Probit	10
4.2.1	Multinomial Logit	11
4.2.2	Ordered Logit/Probit	11
4.3	Tobit and Censored Regression	11
4.4	Robust Regression - M Estimators	11
4.5	Nonlinear Least Squares	12
4.6	Two Stage Least Squares on a Single Structural Equation	12
4.7	Systems of Equations	12
4.7.1	Seemingly Unrelated Regression	13
4.7.2	Two Stage Least Squares on a System	13
5	Time Series Regression	13
5.1	ARIMA/ARFIMA	13
5.2	ARCH/GARCH	14
5.3	Correlograms	14
5.4	Predicted Values	15
5.5	Durbin-Watson Test	15
5.6	Dickey-Fuller Test	16
5.7	Box-Pierce Test	16
6	Plotting	16
6.1	Plotting Empirical Distributions	17
6.2	Adding Stuff to Plots	18
7	Statistics	18
7.1	Working with Common Statistical Distributions	18
7.2	P-values	18

8	Math in R	19
8.1	Matrix Operations	19
8.2	Writing Functions	19
8.3	Numerical Optimization	20
9	Programming	21
9.1	Looping	21
9.2	Conditionals	21
10	Saving Your Work	22
10.1	Saving the Data	22
10.2	Saving the Session Output	22
10.3	Saving as L ^A T _E X	23
11	Conclusion	23
12	Appendix: Code Examples	24
12.1	Monte Carlo Simulation	24
12.2	The Haar Wavelet	24
12.3	Maximum Likelihood Estimation	25

1 Are You Ready for R?

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, covered by the GNU General Public License. Syntactically and functionally it is very similar (if not identical) to S+, the popular statistics package.

1.2 How is R Better Than Other Packages?

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

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

1.3 Obtaining R

R can be obtained as the installation program SetupR, which can be downloaded from <http://www.r-project.org>. Click on the CRAN link, under “downloads” then click “windows 95” followed by “base” and “SetupR.exe”. This obtains the executable and standard libraries. Under linux or OSX, the procedure is similar.

To obtain a library that does not come with the standard installation¹, follow the above procedure, but click on “contrib” instead of “base”. What you will find is a list of compressed libraries ready for download. Click on the one you need and save it somewhere you can find it later. Then start R and click “install package from local directory” under the “package” menu. Then select the file that you downloaded. Now the package will be available for use in the future. If you are using R under linux, install new libraries by issuing the following command at the command prompt: “R CMD INSTALL *packagename*”

¹Packages in the table that are indicated by a * are included by default in recent versions of R but may not be installed by default in older versions.

Contributed Packages Mentioned in this Article and Why

car	Regression tests and robust standard errors
sem	Two stage least squares
MASS	Robust regression, ordered logit/probit
tseries	Garch, ARIMA, and other time series functions
MNP	Multinomial probit via MCMC
Hmisc	L ^A T _E X export
xtable	Alternative L ^A T _E X export
systemfit	SUR and 2SLS on systems of equations
ts*	Time series manipulation functions
nls*	Nonlinear least squares
foreign*	Loading and saving data from other programs
fracdiff	Fractionally integrated ARIMA models
survival	Tobit and censored regression
nlme	Nonlinear fixed and random effects models
nnet	Multinomial logit/probit

1.4 How R Works

In addition to being an interactive program, R is a mathematical language. We can issue commands directly or load script files that we have previously written, which may include new functions. Under windows the prompt and whatever input we give R are in blue while the output it returns is red.

Script files generally have the extension “.R”. These files contain commands as you would enter them at the prompt, and I recommend that you use them if you write anything longer than a few lines. In order to load a script file named “mcmc.R” we would use the command

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

Commands executed from a script file may not print as much output to the screen as we might expect. If we want interactive-level output, we can use the *echo* argument to source

```
> 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 by using the menu item “change working directory”. Also note that under windows the slashes should 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 BATCH *scriptname.R*” The output will be saved in a file named *scriptname.Rout*. Batch mode is also available under windows using Rcmd.exe instead of Rgui.exe.

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

```
> library(foreign)
```

will load the library that contains the functions for reading and writing data that is formatted for other programs, such as SAS and Stata. Alternately (under windows), we can pull down the “package” menu and select “library” in order to select from a list of installed libraries.

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+.

2 Data in R

2.1 The Basics

R allows you to create many types of data storage objects, such as numbers, vectors, matrices, 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. Typing the name of the object typically echos its data to the screen.

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.

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

uses the *c()* (concatenate) command to create a COLUMN 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 make a row vector. Notice that R is case sensitive. 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 into rows.

To view the contents of any data object type the object name.

```
> 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.

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 rows

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

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

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

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

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

```
> firstten <- wealth[1:10,]
```

```
> eighty <- wealth[wealth$year==1980,]
```

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

2.1.1 Dataframes

Most of our data will be in the form of a dataframe. A dataframe is a collection of columns containing data, which need not all be of the same type. Each column has a title by which the whole column 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 element-wise. 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*. Most mathematical operations behave as one would expect.

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.

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

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

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

2.2 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. The other format we will occasionally use is the binary format used by Stata, which will generally come with the extension “.dta”.

If *blah.dat* is a delineated text file, as described above, and was 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 excel). These files are very similar to those mentioned above, but use punctuation to delimit columns and rows. instead of *read.table()*, use *read.csv()*.

3 Cross Sectional Regression

3.1 Ordinary Least Squares

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

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

or alternately:

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

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

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

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

$$salary = \beta_0 + \beta_1 age + \beta_2 experience + \epsilon \quad (1)$$

Using *lm()* results in an abbreviated summary being sent to the screen, giving only the Beta coefficient estimates. For more careful analysis, save the results in as a data member:

```
> result <- lm(salary ~ age + experience + age*experience,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, such as this one with the interaction term. Functions such as *log()* and *sqrt()* are also allowed in the model description. Unfortunately, in order to include a power term, such as *age* squared, we must either first compute the values, then run the regression, or use the *I()* operator, which forces computation of its argument before evaluation of the formula

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

or

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

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

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

3.2 Extracting What You Need

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
```

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()`), and other statistics are as named.

3.3 Heteroskedasticity and Friends

3.3.1 Heteroskedasticity Consistent 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 matrix² use the `hccm()` function (from the `car` library) instead of `vcov()`. The diagonal entries are variances and off diagonals are covariance terms.

3.3.2 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)
```

4 Special Regression

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

4.1 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 you 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), you could simply run the regression

²obtaining 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 I removed some of the coefficients for brevity's sake.

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 you wanted to do the same regression, but by a numeric code specifying an area, you 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.

The package `nlme` contains functions for doing fixed and random effects models in a linear or nonlinear framework. I haven't actually gotten around to figuring out how to use it yet, since the above method has always worked for me.

4.2 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)
```

There is also a special package for binary dependent variable regressions called *boolean*. The *boolean* framework generally requires that a boolean data object be prepared using *boolprep()* and passed to *boolean()*. It also includes functions to plot and do tests. I haven't had to use it.

4.2.1 Multinomial Logit

There is a great function for performing a multinomial logit calculation in the *nnet* library called *multinom()*. To use it, simply transform your dependent variable to a vector of factors (including all cases) and use syntax like a normal regression. If your factors are stored as vectors of dummy variables, you 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 is characteristically ill conditioned. A method that uses markov chain monte carlo simulations, *mnp()*, is available in the *MNP* library.

4.2.2 Ordered Logit/Probit

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

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

4.3 Tobit and Censored Regression

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

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

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

4.4 Robust Regression - M Estimators

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

4.5 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

$$Y = F(X; \beta) + \epsilon \quad (2)$$

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

$$Y = \frac{\epsilon}{1 + e^{\beta_1 X_1 + \beta_2 X_2}} \quad (3)$$

$$\log(Y) = -\log(1 + e^{\beta_1 X_1 + \beta_2 X_2}) + \log(\epsilon) \quad (4)$$

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, you may have to load the *nls* library.

4.6 Two Stage Least Squares on a Single Structural Equation

For single equation two stage least squares, I use the *tsls()* function in the *sem* library. If you want to find the effect of education on wage while controlling for marital status but think educ is endogenous, you 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 you want to estimate and the second is a tilde followed by all the instruments and exogenous variables from the structural equation (everything you need for the *Z* matrix in the 2sls estimator $\hat{\beta} = \{X'Z(Z'Z)^{-1}Z'X\}^{-1}X'Z(Z'Z)^{-1}Z'y$).

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 you were to do this using an actual two stage approach, the resulting standard errors would be bogus.

4.7 Systems of Equations

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

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

4.7.1 Seemingly Unrelated Regression

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

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

4.7.2 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
> result2spls <- systemfit("2SLS",system,labels,inst)
```

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

5 Time Series Regression

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

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

Most time-series related functions automatically coerce the data into *ts* format, so this command is often not necessary. We can compute differences of a time series object using the *diff()* operator, which takes as an optional arguments, the number of lags and differences to use. If we want to lag the data directly, we simply 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.

```
> lagy <- y[-1]
> lag4y <- y[-(1:4)]
> ysmall <- y[-NROW(y)]
> ydif <- diff(y)
```

Produces *lagy*, a once lagged version of *y*, *lag4y*, a four times lagged version of *y*, and *ysmall*, a version of *y* with the last data entry removed so as to be conformable with *lagy*. The variable *ydif* contains the first differenced version of *y*.

5.1 ARIMA/ARFIMA

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

$$\Delta^d y_t = \mu + \gamma_1 \Delta^d y_{t-1} + \dots + \gamma_p \Delta^d y_{t-p} + \epsilon_t + \theta_1 \epsilon_{t-1} + \dots + \theta_q \epsilon_{t-q} \quad (5)$$

where

$$\Delta y_t = y_t - y_{t-1} \quad (6)$$

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.

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

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

fits an AR(3) model, for example.

The function *fracdiff()*, from the *fracdiff* library fits a specified ARMA(p,q) model to your 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.

5.2 ARCH/GARCH

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

$$\sigma_t^2 = \alpha_0 + \delta_1 \sigma_{t-1}^2 + \dots + \delta_p \sigma_{t-p}^2 + \alpha_1 \epsilon_t^2 + \dots + \alpha_q \epsilon_{t-q}^2 \quad (7)$$

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

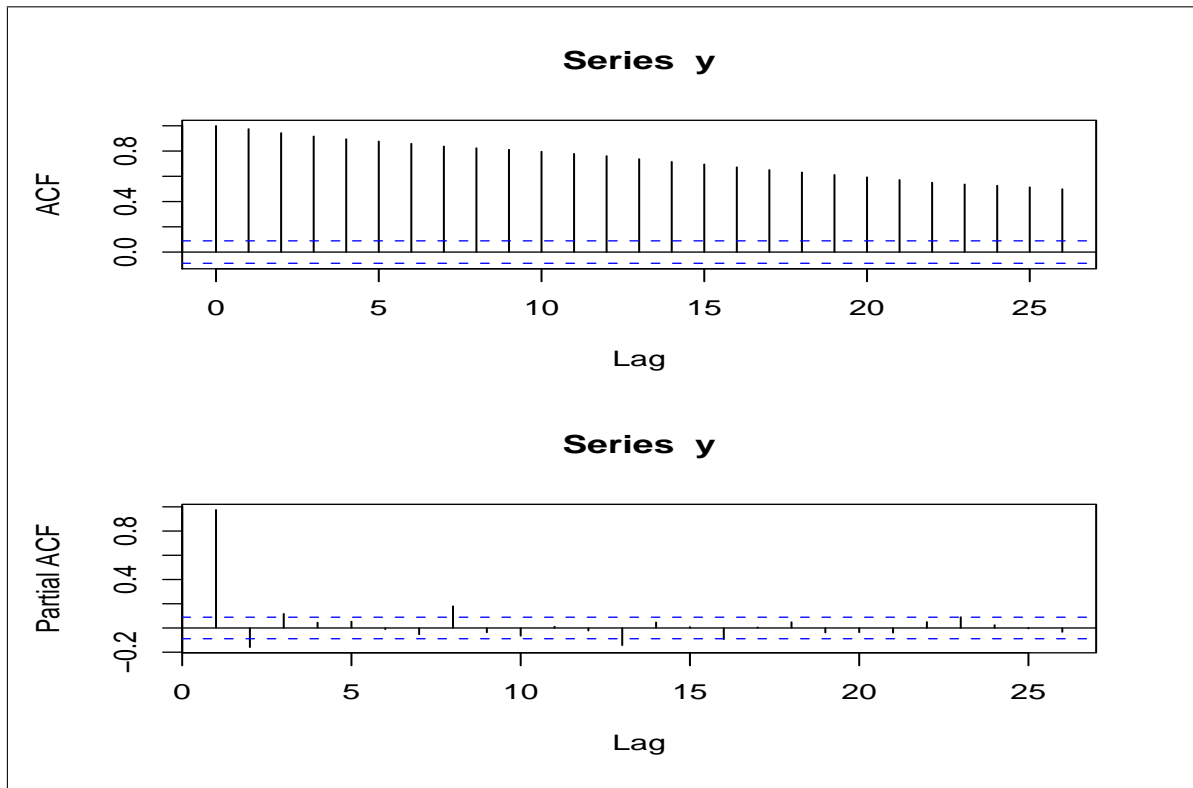
```
> archoutput <- garch(y,order=c(0,3))
> garchoutput <- garch(y,order=c(2,3))
```

so that *archoutput* is the result of modeling an ARCH(3) model and *garchoutput* is the result of modeling a GARCH(2,3). Notice that the first value in the *order* argument is *p*, the number of alphas, and the second argument is the number of delta parameters. The resulting coefficient estimates will be named *a0*, *a1*, etc. for the alpha and *b1*, *b2*, etc. for the delta parameters.

5.3 Correlograms

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

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



5.4 Predicted Values

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

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

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

5.5 Durbin-Watson Test

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

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

5.6 Dickey-Fuller Test

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

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

Augmented Dickey-Fuller Test

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

5.7 Box-Pierce Test

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

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

Box-Pierce test

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

would lead us to believe that the model may not be correctly specified, since we soundly reject the Box-Pierce null.

6 Plotting

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

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

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

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

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, for drawing connected lines or scatter plots, respectively. Many of the same options that apply to `plot()` apply to `lines()` and many other graphical functions.

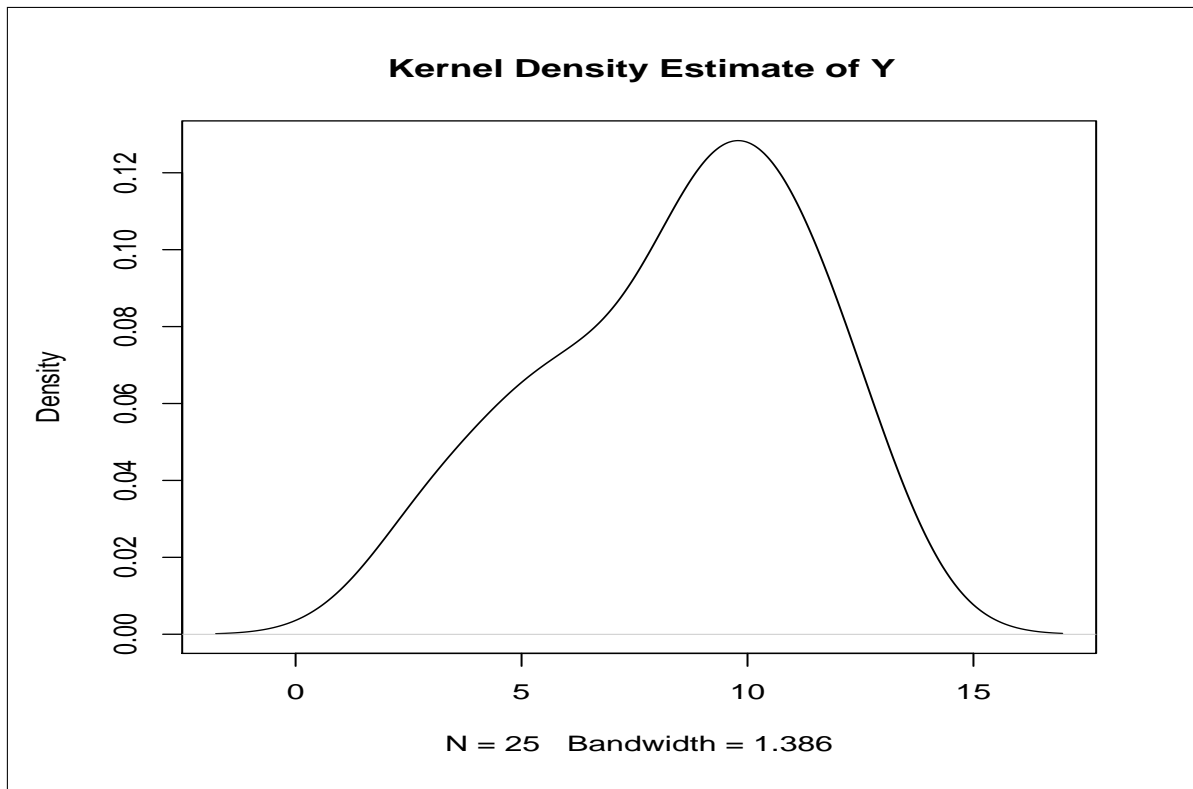
We can plot a line, given its coefficients, using the `abline()` command. This is often useful in visualizing the placement of a regression line after a bivariate regression

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

6.1 Plotting Empirical Distributions

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

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



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

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

6.2 Adding Stuff to Plots

After plotting we often wish to add annotations or other graphics that should really be placed exactly. Functions like *text()* 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.

7 Statistics

R has extensive statistical functionality. The functions *mean()*, *sd()*, *min()*, *max()*, and *var()* operate on data as we would expect³.

7.1 Working with Common Statistical Distributions

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

```
> 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. Note that we could replace *norm* with *binom*, *nbinom*, *chisq,t,f* or other distribution names if appropriate.

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 $[0,1]$

7.2 P-values

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

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

³note: the functions *pmax()* and *pmin()* function like max and min but elementwise on vectors or matrices.

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, you 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 Math in R

8.1 Matrix Operations

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. 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

$$\beta = (X'X)^{-1}X'Y \quad (8)$$

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 `%x%` operator.

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

calculates the Kronecker product of g with h .

8.2 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 the “+” signs are prompts R provides to remind you that you are inside brackets. They are not anything you type.

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),A1=3)
> f(c(2,4,1),Alpha=3)

```

Variables that are not passed in as arguments are not available within functions and variables defined within functions are unavailable outside of the function. Changing the value of a passed-in argument within a function does not change its value outside of the function. In other words, R passes arguments by value and variable scoping applies.

8.3 Numerical Optimization

R can numerically minimize an arbitrary function using the command `nlm()`, which takes as its argument a function and a starting vector at which to evaluate the function. The first 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. In order to maximize a function, multiply the function by `-1` and minimize it.

```

> 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

```

This function 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)

```

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 Programming

9.1 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 you 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 your command is, when you 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");}
```

9.2 Conditionals

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 than 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 good programming practice to avoid loops and *if* statements whenever possible when writing in any scripting language.

The Boolean Operators

<code>! x</code>	NOT x
<code>x & y</code>	x and y elementwise
<code>x && y</code>	x and y total object
<code>x y</code>	x or y elementwise
<code>x y</code>	x or y total object
<code>xor(x, y)</code>	x xor y (true if one and only one argument is true)

10 Saving Your Work

10.1 Saving the Data

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

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

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

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

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

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

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

10.2 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. Of course, `sink()` hides the output from you as you are interacting with R, so many times the easiest way to get a transcript of your session is to copy and paste using the mouse.

If you are using a script file, a nice way to get a transcript of your 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. From the command line we can also use the commands

```
> png("myplot.png")
> plot(x,y)
> dev.off()
```

to write a png file.

10.3 Saving as L^AT_EX

R objects can also be saved as L^AT_EX tables using the *latex()* command from the *Hmisc* library. The most common use I 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 L^AT_EX source file⁴

summary	Estimate	Std. Error	t value	Pr($\hat{\beta}_j - t$)
(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 <i>educ</i> on <i>exper</i> and <i>south</i>				
summary	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	17.2043926	0.088618337	194.140323	0.00000e + 00
exper	-0.4126387	0.008851445	-46.618227	0.00000e + 00
south	-0.7098870	0.074707431	-9.502228	4.05227e - 21

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

11 Conclusion

R provides an effective platform for econometric computation and research. It has built in functionality sufficiently advanced for professional research and has a reasonably steep learning curve (if you put knowledge on the y axis and effort on the x). Because R is a programming language as well as an econometrics program, it allows for more complex, tailored computations and simulations than one would get in a prepackaged system. On the other hand, it takes some time to become familiar with the syntax and reasoning of the language. I hope that this guide eases the burden of learning to program and do standard data analysis in the finest statistical environment available. Don't forget to let me know if you feel like I didn't do this or have a suggestion about how I could do it better.

⁴Under linux, at least, the *latex()* command also pops up a window showing how the output will look.

12 Appendix: Code Examples

12.1 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

$$y = 2 + 3x + \epsilon. \quad (9)$$

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)
```

12.2 The Haar Wavelet

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

$$\psi^{(H)}(u) = \begin{cases} -1/\sqrt{2} & -1 < u \leq 0 \\ 1/\sqrt{2} & 0 < u \leq 1 \\ 0 & \text{otherwise} \end{cases} \quad (10)$$

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.

12.3 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

$$y = aL^bK^c + \epsilon \quad (11)$$

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)
```