R Studio Tutorial

WRITTEN FOR ECON*3740

Prof. Ross McKitrick

August 25 2020 edition

TABLE OF CONTENTS

INTRODUCTION: R AND R STUDIO

This manual will help you get started with the free statistical programming package R, which is also available in a more user-friendly version known as R Studio. It is specifically written for students taking ECON*3740 (Introductory Econometrics) using the Koop text (Introduction to Econometrics) but students in other courses should not have any problem working through it. There are four data sets needed: income.xls, orange.xls, compute1.xls and rts.xls; they should be available in the same place you got this file from.

There are many popular econometric packages out there, such as SAS, SPSS, Shazam, Stata, E-views and more. I've used almost all of them over the years. Each one has pluses and minuses, but I now use R exclusively. This is partly because so many other economists and data analysts use it, so sharing code is easier. The fact that it is free is also a big help: with the other packages you need to purchase upgrades every few years and they can be expensive. And once you're used to it, R's structure makes programming faster and its ability to handle and combine data sets is superior to Stata and other packages I've used.

R has some downsides. It is more complicated to get started in, and the online help documentation is often not much help. This booklet presents a 6-step sequence for learning how to use R Studio, for both Windows and Mac.

NOTE: Sections 1—3 are the longest. Sections 4—6 will take less time, in fact you can skip parts of Sections 5 and 6 until later in the course (I have indicated where).

ALSO NOTE: Material in Section 7 and beyond is for later in the course.

Troubleshooting
This manual covers 6 introductory steps plus some other commands and programming steps you will need to do basic econometrics in R. As long as you work through the steps you should pick up everything you need to know to do the programming required. There will inevitably still be glitches and problems along the way. At any point if you run into trouble, here are some general suggestions.

- Try to figure out if the problem is *syntax* (did you spell the commands correctly), a *read-write* problem (can R find the file you're trying to read and extract the data from it?), or something to do with how your computer's operating system is managing R Studio.

- If it's syntax, double check how you've written the command. If it's read-write, check that you are in the right working directly (by typing `getwd()`, see Section 1.3.1) and go over the material in Section 2.

- If you are writing data to a spreadsheet file and you have the file open at the same time, R will give you an error message that looks like this:

```
Error in file(file, ifelse(append, "a", "w")) :
  cannot open the connection
In addition: Warning message:
In file(file, ifelse(append, "a", "w")) :
  cannot open file 'econ/reg.results.csv': Permission denied
```

  Close the file you are writing to and it should now work.

- You might run into a problem with "smartquotes". If you copy and paste commands from an MSWord document into the R command prompt or workspace, beware that the character " in Word is not the same as " in a text editor. R can only recognize plain, unformatted quotation marks. MSWord and other word processors (including the text editor on a Mac) replace them with "smart quotes"—namely quotation marks formatted either as open quotes or close quotes. The R workspace editor doesn't know what these mean, so try erasing the quotation mark characters and re-typing them in in the R editor itself.

- If you are having trouble installing a package (or if R is telling you the package does not exist) you probably need to upgrade to a newer version of your operating system and then install a newer version of R. Sometimes when packages are updated they are not backwards-compatible with older versions of R.

- If you are getting an error message and you want to email me or the TA, include the code so we can look at it.

- If you want to do something and you can't remember the command name, check the list at the end of this booklet or Google it. For instance if you want to save a data file into a .CSV spreadsheet, Google "save data in csv file in R" and several pages will come up with the instructions written out.

- If you know the command name but you can't remember how exactly to use it, type `?commandname` (using the command name, e.g. `?print`) at the R-Studio command prompt and the instructions will come up in a side window.

- You can search Youtube and you'll find lots of helpful tutorial videos. Even if the video is for R rather than R Studio, it will likely give you the information you need.

- If all else fails, try re-booting your computer.

Time to get started.

1 DOWNLOADING AND INSTALLING R STUDIO

1.1 WINDOWS USERS

On your desktop or in any convenient location where you will easily be able to find it, create a folder called ECON3740. Inside that folder create a subfolder called "prelim". This is the folder we will use throughout Sections 1 to 6 of this document so you should copy the data files that I refer to (on Courselink) into this location.

You first need to install the core R software. Go to http://cran.rstudio.com and click on Download R for Windows. Next click on "base", which takes you to https://cran.rstudio.com/bin/windows/base/. Now click on "Download R 4.0.1 for Windows" and you will see an executable file download. In Chrome the file appears along the bottom of the window. Double click on it and give the usual permissions. R will install and put an icon on your desktop. R runs well on Windows machines so the current version should not give you any problems, but if you have a very old PC and you get an error message you may need a version of R from the archive here: https://cran.r-project.org/bin/windows/base/old/

Once R is installed, you next need to install R Studio, which is an interface on top of R. Go to www.rstudio.com, click on Download and select the Windows 10/8/7 (64 bit) option.

Once the file has downloaded click on the filename in the downloads bar along the bottom of your browser (or wherever your browser shows downloads) and when prompted, allow the app to proceed. Once the installation is done you can run the program by going to the Start menu and clicking the Tile labeled R Studio.

If you get an error message during the R Studio install it may be that your version of Windows is not up to date, so you may have to click on this link to get an older version of R Studio. Version 1.1.463 will probably work.

1.2 MAC USERS

On your desktop or in any convenient location where you will easily be able to find it, create a folder called ECON3740. Inside that folder create a subfolder called "prelim". This is the folder we will use throughout Sections 1 to 6 of this document so you should copy the data files that I refer to (on Courselink) into this location.

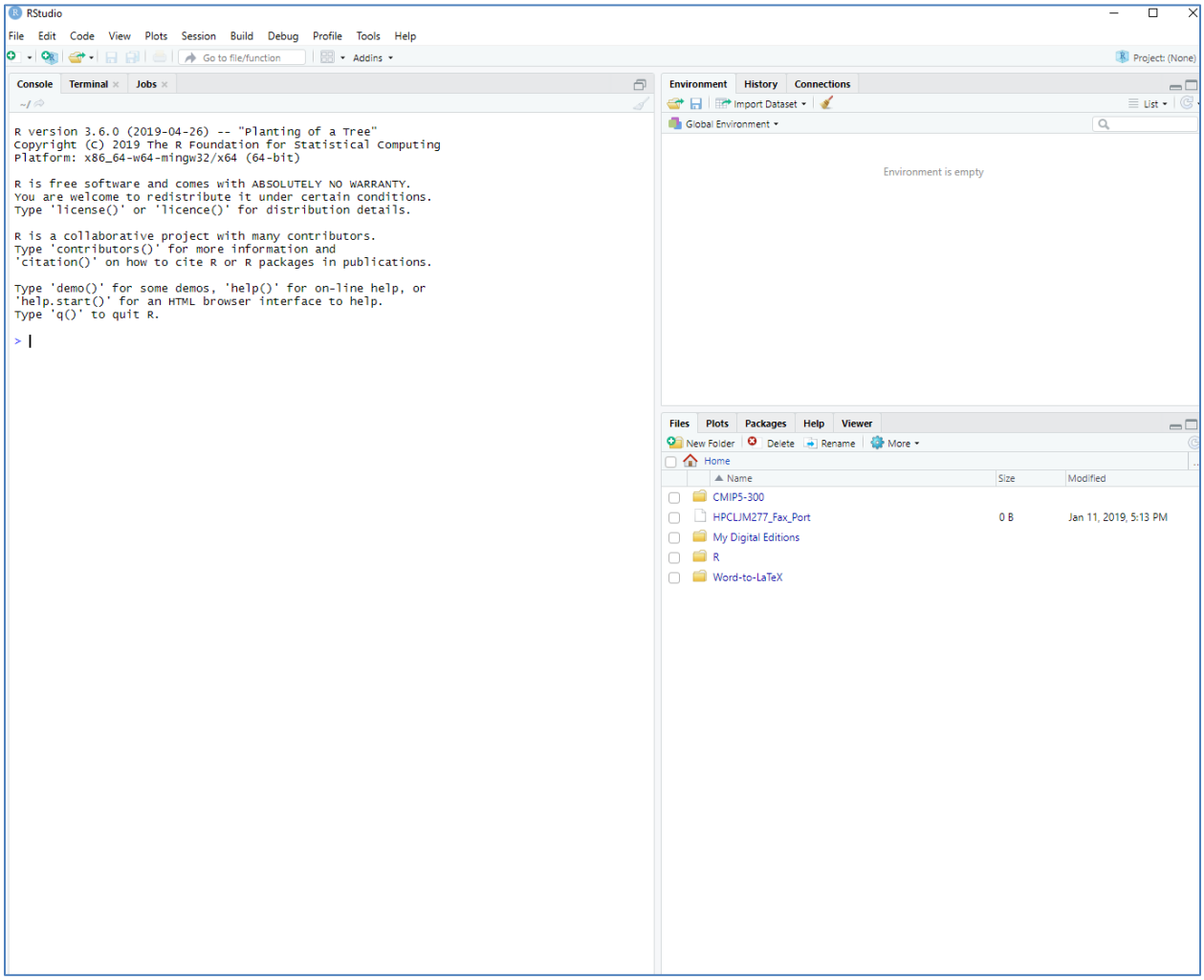
You first need to install the R core package. Check which version of the Mac Operating System you have by looking under the apple menu and selecting About This Mac. Take note of the number (it will be something like 10.6 or 10.9 or 10.11.) Then go to https://cran.rstudio.com/bin/macosx/. If your system is OS X 10.13 or higher you need the R 4.0.1 version. If it's 10.11 or higher you will need the "El Capitan" version of R. If your system is older than that, look on the page for a suitable version compatible with your operating system. For instance if you have OS X 10.6 you should use the "Snow Leopard" version. Bear in mind that you can probably get a free OS system upgrade from Mac and if you can it is better than using an old version of R.

When you have downloaded the appropriate file go to the downloads folder (the icon in Safari is in the top right, in Chrome you will probably see the file along the bottom). Click the usual approvals and then the program will install itself.

Now you need to install R Studio, which is an interface on top of R. Go here and look for the MacOS option. If you have a Mac with OS X 10.13 or higher, click on the link to download the installation file. If your Mac OS is older (has a lower OS number), click here for an older version of RStudio. If your system is 10.9 or higher you can use the Desktop 1.1.463 version, otherwise you'll need to try an older version (or, preferably, update your Mac OS). As before, click on the relevant link and download the file, then run it. You'll see a folder open with the RStudio icon. If a scrolling bar appears for a while saying "verifying R Studio" and then an error message comes up, you need to go back and get an earlier version of the program. Once you have a compatible version, R Studio will install quickly and open up.

1.3 Starting R Studio

After installing R Studio, start the program and a window will open up that looks something like this:



The pane on the left is the console with the **command prompt** (>) where you can enter commands directly. The one on the lower right will initially have a list of folders in your Documents folder since that's where it points by default. The one above it shows the contents of the active memory as R runs.

At the console prompt, to see R at work, try typing `>2+2 <enter>`. You'll see

```
> 2+2
[1] 4
>
```

That means R Studio is installed and working as it should.

1.4 WORKING IN THE R STUDIO ENVIRONMENT

1.4.1 Finding and Changing the Working Directory

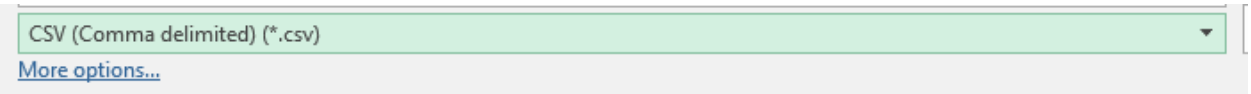
Now let's find your ECON3740 folder and make it the working directory so you can keep all your files in one place. Under the heading **Session** select **Set Working Directory > Choose Directory** … and you'll see a window open up that lets you browse folders. Find the ECON3740/prelim folder and select it. When you do, you'll see the lower right pane goes blank, because it's now looking at the empty ECON*3740/prelim folder.

You can confirm it's the working directory using the `getwd` command. At the command prompt (left pane) type `getwd()` which means "show me the current working directory" (which is useful if R can't seem to find your data or command files). You'll see the file path to your ECON3740 folder pop up.

Note that with R commands you almost always include an `()` after them because sometimes you enter parameters inside the brackets, as we will see.

Permanently set the working directory
If you stop R Studio and restart it, the working directory will go back to the default (Documents folder most likely). To make the change permanent go to **Tools > Global Options** and click the Browse button beside the box labeled Default Working Directory (when not in a project). Find the ECON3740 folder and select it.

Get some data files
On the Courselink page for this class download the files income.xls and orange.xls. Open them and resave them as CSV files. Be sure to choose the right CSV format. The option looks like:



Now copy those files into the ECON3740/prelim folder. We will use the data in those files throughout this set of lessons.

1.4.2 Writing output to the screen and to a file

Let's write a simple program. Under **File** select **New File > R Script**. The left pane will split into two parts, with the command console down below and a blank **workspace** above. Make sure your cursor is in the workspace above. We will create two variables, x and y. Each will consist of 5 data points. Type the following:

```
x = c(1,2,5,3,6)
y = c(1,2,3,4,5)
```

The `c()` format is very important in R. It tells the program to treat whatever is inside the brackets as a list.

print() command

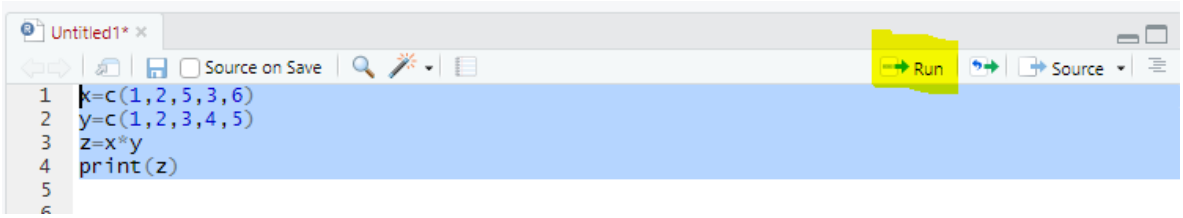
Now type

```
z = x*y
print(z)
```

The first line asks R to multiply each element of x by the corresponding element of y and call the result z. The next line tells R to print the result to the console.

Note that when you insert an open-bracket R Studio will automatically insert the close-bracket for you. But you need to cursor past it to confirm that you want it inserted. The editor built-in to R Studio has a number of convenient automatic formatting features like this.

Now highlight the lines you've written and click Run in the top line:



In the console below you'll see the result:

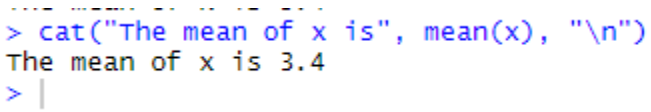
```
> x=c(1,2,5,3,6)
> y=c(1,2,3,4,5)
> z=x*y
> print(z)
[1]  1  4 15 12 30
```

You can verify that R did the multiplication correctly. You'll also see in the top right window the variables listed under the heading "Global Environment". This shows you what R is currently maintaining in its memory.

cat() command – creating more elaborate print statements
in the workspace add the command

```
cat("The mean of x is", mean(x),"\n")
```

This tells R to concatenate ("cat") or string together the phrase "The mean of x is", the calculated value mean(x) and a carriage return indicated by "\n". Highlight and run it and you'll see in the R console the output



sink() – direct your output to a file
Add a new line to the start of your program and the end, as follows.

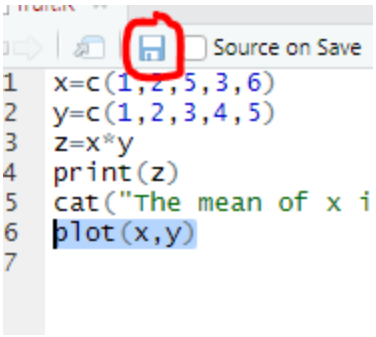
```
sink("output.txt")
x = c(1,2,5,3,6)
y = c(1,2,3,4,5)
z = x*y
print(z)
cat("The mean of x is", mean(x), "\n")
sink()
```

The first sink() command tells R to write the output to a file called "output.txt" instead of writing it to the console. Then the second sink() command returns the output to the console.

Highlight and run the commands. You'll notice output.txt appears in your list of files in the working directory. Open it either by going to the folder and double-clicking on it or just by double-clicking on the name in the Files window and you'll see it open in the top left R Studio pane. You'll see the lines that used to appear in the console window are now in the text file. This is very handy when you want to preserve the output of your calculations.

plot() – draw a picture of your data
Now in the top window, below the `cat(…)` command type `plot(x,y)`, then run it. In the bottom right pane you'll see the scatter plot of y against x appear. You can click the tab for Files to go back to the view of the working directory if you want. On the Plots tab, when you have made a graph you can click the Export header to save the plot as a PDF or an Image. Save the file as a PDF and give it a simple name like xy_plot. When you go to the Files header you'll see the file is now there.

We will learn more about plotting in Step 4. For now you can save your command file by clicking the save icon above the command lines



When prompted, give the file the name first_program and click Save. In the files window you'll see it appear as an .R file. Now close R Studio. You'll be asked if you want to save the workspace image, but you can usually click Don't Save since you have the command file that recreates everything if you need it.

Now restart R Studio. You'll see that it automatically loads your most recent program. You can stop it from doing that if you want (under **Tools > Global Options**). If you highlight the command lines and click Run, you'll see the output, values and plot all reappear.

For the rest of these introductory steps I will assume you have a working directory that you can find and use. I suggest you call it ECON3740/prelim. You'll then create separate folders for your assignment materials through the semester.

In what follows, if I am referring to typing an individual command at the command prompt (lower window) I will prefix it with the > symbol. If I refer to typing something in the workspace, I am talking about writing lines in the program file.

2 READING, WRITING AND MANAGING DATA

2.1 READING DATA FROM A CSV FILE

Most of the data we will use comes in the form of spreadsheets. Find the file **income.xls** that comes with your textbook (I will post it on Courselink for convenience). This is an excel spreadsheet with 3 columns: date, log of income and log of consumption. Re-save it as a CSV file in your /prelim folder. Using the CSV format makes it easier to read into R, though R can read XLSX files if we add a special package to it. Make sure to choose the option "CSV (Comma delimited)".

Open R Studio and verify that your working directory is ECON3740/prelim. Open a new program file and put the following commands in it

```
sink("output.txt")
cat("Start of program", "\n")
rm(list=ls(all=TRUE))
inc = read.table("income.csv", sep=",", header=TRUE)
cat("End of program", "\n")
sink()
```

rm() – remove items from the working environment

The third line is a command to empty the memory of all previous data. Note that R commands are always of the form ***command*(…)** where the first word is the function and the part in brackets is the list of variables and parameter(s) to use in the function. In this case the command says "Remove everything on a list which consists of everything". This is a handy line to include in files when you are running and re-running programs and you might reuse variable names, so you need to empty the memory first. If you just want to remove a single variable, such as z, you could write `rm(z)`.

read.table() – get data from a spreadsheet

The line starting "inc =" tells R to read a table from a file called income.csv which has the values in rows separated by a comma, and a header row with variable names. If we left out the "inc =" part, R would read the file and just dump the contents to the screen (try it). By assigning the command to a variable name (inc) we are putting the output of the read.table function into a *data frame*, which is one of the storage systems in R.

Now if you run this program it will create a data frame called inc and an output file called output.txt. See if you can guess what will be in the output file, then check to see if you were correct.

2.2 DATA AND DATA FRAMES

Run the commands then type `ls()` at the command prompt (which asks R to list all the entries in the Global Environment) and you'll see the following:

```
"inc"
```

That's it. Also if you look at the Environment tab in the top right pane all you'll see is "inc". In particular, the 3 individual columns aren't listed separately, though if you click the little arrow it

will expand the menu and show you they are there. They are bundled in the "inc" data frame. Type `inc` at the console and you'll see all three columns print out.

attach() - index a variable directly without referring to its data frame
To access an individual variable in the inc data frame, refer to it with the prefix "inc$". For instance, type in

```
>inc$Income
```

and you'll see the Income variable listed out. If you want to make the variables in 'inc' directly accessible without having to use the inc$ prefix, you can "attach" them to the top directory by typing

```
>attach(inc)
```

and then if you type `>Income` the variable contents will be displayed.

2.3 FILES WITH HEADERS – THIS IS IMPORTANT, DO NOT SKIP!

Some data files have more than one header line. This can create problems and we have to explain to R how to read the data. In the Koop data archive find the file orange.xls, then copy it into your setup folder, resaving it as a CSV file (if you haven't done so previously). Note that this file has two header rows, "organic/regular" and "pence/lb./pence/lb." This requires an extra step to work around.

Use the R editor to create a file called fruit.R. Enter the following commands:

```
rm(list=ls(all=TRUE))
fruit = read.table("orange.csv", sep=",")
```

Now run the file. This tells R to discard everything in its memory, read the file "orange.csv" using commas to separate columns, and assign the contents to the data frame "fruit".

Now if you type `>fruit` at the command prompt and hit return you will see the following:

```
           V1        V2
1     organic   regular
2   pence/lb. pence/lb.
3     83.0273   66.5331
4     91.1854   66.4581
5     82.2809   66.3976
6     91.6732   66.3488
7     93.1171   67.5421
8     88.1942   70.5665
9     97.4186   70.2617
…
```

But it's not what it appears. R read the file as if it contained 2 columns (V1 and V2) of *text* not numbers. It assumed the data are text because the first entry in each column was text, and R assumes that the type of content in a variable is determined by the class of data in the first row

(text, number, etc.) So the 3rd row of column V1 is not the number 83.0273, it is the word "82.0273". If you try to compute the mean of the organic orange prices you will get an error.

```
> mean(fruit$V1)
[1] NA
Warning message:
In mean.default(fruit$V1) :
  argument is not numeric or logical: returning NA
```

If there is only one header row we can tell R to assume the top row is the variable name as follows:

```
>fruit=read.table("orange.csv", sep=",", header=TRUE)
```

Now type `>fruit` and hit return and you get:

```
      organic   regular
1   pence/lb. pence/lb.
2     83.0273   66.5331
3     91.1854   66.4581
4     82.2809   66.3976
5     91.6732   66.3488
6     93.1171   67.5421
7     88.1942   70.5665
8     97.4186   70.2617
….
```

The V1 and V2 were replaced with the variable names. But we still have the same problem that the data are all treated as text, because the second rows contain text. So we need to get R to skip the first 2 rows. We do this using the following command:

```
fruit=read.table("orange.csv", sep=",", skip=2)
```

Now fruit contains numerical data:

```
>fruit
          V1       V2
1    83.0273  66.5331
2    91.1854  66.4581
3    82.2809  66.3976
4    91.6732  66.3488
5    93.1171  67.5421
6    88.1942  70.5665
7    97.4186  70.2617
…
```

If you type `> mean(fruit$V1)` you will get the answer 163.8089.

Incidentally, once you have typed `> mean(fruit$V1)`, if you need to enter the same command again, instead of re-typing it you can use the up-arrow button on your keyboard to scroll through all the commands you typed before. This can be a time saver.

Note that we now don't have variable names, so we make new ones by adding these commands to the file:

```
organic = fruit$V1
regular = fruit$V2
```

Now if you type `mean(organic)` you'll see the answer appear (163.8089).

IT IS IMPORTANT TO LOOK AT YOUR DATA FILE BEFORE READING IT IN, so you can see if you need to skip one or more lines of text to find the numbers.

3 ELEMENTARY DATA HANDLING

3.1 WORKING WITH DATA FRAMES

A data frame is a collection of variables that are associated with each other in a data set, each of which has the same number of rows. This is the default method R uses to store data. It is not the same as a matrix, since R would not be able to perform matrix operations on a data frame (although it is easy to convert a data frame into a matrix). But R can do useful operations on a data frame.

Copy and run the following program

```
rm(list=ls(all=TRUE))
fruit = read.table("orange.csv", sep=",", skip=2)
organic = fruit$V1
regular = fruit$V2
new = log(organic) + sqrt(regular)
```

By now you should be able to look at the program and recognize what it does. It clears the environment, reads the orange price data into a data frame fruit, creates new variables using the columns of the data frame fruit, and creates another new variable called new using the log of organic price plus the square root (`sqrt()`) of the regular price. This formula doesn't mean anything; it is just a way to generate some new numbers.

head() and tail() – inspect the first or last 5 rows of a data frame

If you want to view just the first 5 rows of a data frame use

```
>head(fruit)
```

and similarly `>tail(fruit)` would show you the last 5 rows. This is sometimes handy when you are trying to figure out what variables are in a data frame.

colnames() – rename the columns in a data frame

When you type `>head(fruit)` you will see that the variables in it are still named V1 and V2 rather than organic and regular. You can change that by using `colnames()`, as in:

```
>colnames(fruit) = c("organic","regular")
```

To use these variables you still need to refer to them in their data frame, i.e. fruit$organic and fruit$regular.

cbind() – add or bind another column to a data frame
Since we created a new data series called "new" we can bind it to the existing data frame using `cbind()`:

```
>fruit = cbind(fruit, new)
```

This tells R to replace the data frame fruit with a new one consisting of the old one plus a new column consisting of the variable new. For this to work you need to ensure that the new variable has the same number of rows as the data frame it is being added to.

Accessing specific rows and columns using [ ]
Now suppose you want to see just the first 10 rows of the first 2 columns of fruit. You index locations in a data frame using a pair of numbers separated by a colon : where the pair indicates the start and finish. The first pair indicates the rows and the second pair indicates the columns, and the pair are separated by a comma and enclosed in [square brackets] right after the data frame name.

For example: type

```
>fruit[1:10 , 1:2]
```

This tells R to show you rows 1 to 10 and columns 1 to 2. You should see the following:

```
> fruit[1:10,1:2]
    organic regular
1   83.0273 66.5331
2   91.1854 66.4581
3   82.2809 66.3976
4   91.6732 66.3488
5   93.1171 67.5421
6   88.1942 70.5665
7   97.4186 70.2617
8   90.5336 72.9029
9  103.4476 74.9074
10  89.5046 74.1933
```

If you leave out the start:finish list (but still include the comma), R assumes you want everything in that category (rows or columns). So let's say you write

```
>fruit[1:5 ,]
```

This asks R to show you rows 1:5 and all the columns. It will give you the same printout as if you had typed in `head(fruit)`. If you just want to see the values in the new variable, type

```
>fruit[,3]
```

This ask R to print all the rows in column 3. It's the same output as if you had typed `>new`.

To check the dimensions of your data frame use `dim(fruit)`, to check the number of rows use `nrow(fruit)` and to check the number of columns use `ncol(fruit)`.

Data types and transformations
In econometrics we usually use one of two types of data: time series or cross-sectional. The first denotes data collected over time, such as the annual inflation rate. The second denotes data collected at one point in time across different locations, such as the population per US state. If data consists of time series collected at different locations we call it *panel* data; but we won't use it in this course.

Sometimes the data in a data file is not in the form we want to use, so we apply a transformation. The previous step involved two simple data transformations: the log function and the square root function. We will often use transformations of our data. For instance, suppose we have data on national Gross Domestic Product or GDP, and national Population (P). Then we can compute GDP per capita by dividing one by the other: GDP / P. Or suppose we have time series data on, say, total population at time t, which we denote P_t. Then we can compute the percentage change at time t using

$$\frac{P_t - P_{t-1}}{P_{t-1}} \times 100$$

If your price variable is called P, to compute the numerator (called the first difference) use `diff(P)`. Computing the denominator in R is a bit tricky without downloading a special package, but there's a simple trick which we'll learn in Section 8.4.

Before going on, please read pages 1—6 in the textbook.

3.2 SUMMARY STATISTICS

We will now use R Studio to compute some summary statistics. Before going on please read pages 11—13 in the textbook, to remind yourself about the concepts of *mean* and *variance*. Then read pages 16 and 17 to remind yourself of the concept of *correlation*.

To get some summary statistics of the variables in fruit, use the `summary()` command.

```
>summary(fruit)
```

This will print out, for each variable, the minimum, 1st quartile, median, mean, 3rd quartile and maximum. It doesn't give you the standard deviations or variances. To get the variances use

```
>var(fruit)
```

This gives you a 3x3 matrix with the variances down the diagonal and the covariances off the diagonal. If you have forgotten what a covariance is, don't worry we will cover it in class soon. What if you only want the variances? You can get that by asking R to show you the entries down the diagonal of the matrix produced by the var command:

```
>diag(var(fruit))
```

This yields

```
> diag(var(fruit))
    organic     regular         new
2655.224507 2663.395480    6.243442
```

This is an example of an important feature of the R programming language: you can often embed one command inside another. Sometimes this makes for neater and faster code, although you need to be careful to keep track of orders of operation. If we want the standard deviations of the columns of fruit, that means we need to compute the square roots of the variances. We can add another layer to the command above:

```
>sqrt(diag(var(fruit)))
```

This seems rather cumbersome. There is a function in R called `sd()` that computes a standard deviation, but only of one variable at a time. Type

```
>sd(organic)
```

and you'll see the same answer as the previous command. But if you type

```
>sd(fruit)
```

you will get an error message. In R, to apply a mathematical function to every row or column of a data frame, you need to use the `apply()` command. You probably won't need it in this course, but if you do, the syntax is

```
apply(dataframe, direction, function)
```

where the first entry is the name of your data frame, the second entry indicates if you want to apply the operation across all rows (=1) or columns (=2), and the third entry indicates the function. So if you want to compute the standard deviation of every column in fruit, type

```
apply(fruit, 2, sd)
```

This will give you the same output as the previous command.

3.3 WRITE.CSV() -- SAVING DATA TO A CSV FILE

There are several ways to save your data in R. For now we will look at the most basic one. If you have done a calculation and created a variable like "new" or a revised data frame like "fruit" you can save it to a file using the **write.csv()** command. Enter the following in a program or at the command line:

```
write.csv(fruit, file="fruit.csv", sep=",", row.names=FALSE)
```

This tells R to write the contents of the variable "fruit" to a CSV file fruit.csv. If the file does not exist R will create it. If it already exists, R will empty it then write the new contents in it. The `sep=","`

option tells R to separate values with a comma (hence CSV) and the `row.names=FALSE` option tells R not to put row names in the file (otherwise R will add an unnecessary column of row numbers as the row names).

TIP: if you get an error message check to make sure that the file you are writing to isn't currently open on your computer (see troubleshooting note in the introduction).

4 COMMANDS AND OBJECT ASSIGNMENTS

4.1 THE GENERAL FORMAT OF R COMMANDS

In this step we will learn about the general structure of R commands. Let's start by getting some data. We will use the fruit data from the last step. Enter the following program in the workspace and run it.

```
rm(list=ls(all=TRUE))
fruit=read.table("orange.csv", sep=",", skip=2)
organic = fruit$V1
regular = fruit$V2
new = log(organic) + sqrt(regular)
fruit = cbind(fruit, new)
```

You should now have the variables new, organic and regular in your active environment.

We can display the mean of the organic fruit prices by typing in

```
> mean(organic)
```

The answer is 163.8089.

As noted above, most R commands have the form *`command(var)`* where *command* is the instruction about what function to apply, and *var* is a variable indicating what to apply the function to. However, there are also optional parameters that most functions can use, so the more general syntax is

`command(var, options)`

For instance, suppose we want to compute the mean of organic fruit prices but we want to drop the first 10% and last 10% of the sample. Then we would use the *trim* option as follows:

```
> mean(organic, trim = 0.1)
```

The answer now is 161.9268.

Some options require us to indicate if an option is TRUE or FALSE. Suppose we want the list of regular fruit prices, but we want them sorted from largest to smallest. Type

```
> sort(regular, decreasing = TRUE)
```

and you'll see the prices starting at the highest and ending with the lowest. In the case of `sort`, the default is lowest to highest, so if you re-enter that command but leave out the decreasing option, the numbers will appear in the opposite order.

Even for simple R commands there may be available options that are useful when programming. Use the `?commandname` step at the command prompt to get information on available options. For instance if you enter > `?sort` at the command prompt you'll see a brief description in the Help window of the structure of the command and the available options.

4.2 Subsets of Objects

Now that you have learned to create a sorted list, what if you wanted to know the top ten entries in a list? In the previous section we saw that using a pair of numbers in square brackets separated by a colon tells R to select only those entries. You can usually add a [:] entry right onto a calculation if you only want a subset of the object being created. Type in the following command:

```
> sort(regular, decreasing = TRUE)[1:10]
```

It's the same command as before with [1:10] added on, so instead of re-typing everything, just use the up cursor arrow button then type the bracketed term on the end and hit return. You will see that the output is the first 10 entries of the list you saw previously.

4.3 Creating R Objects

In the previous example we printed the sorted prices to the screen, but did not save them for later use. Suppose we want to use the sorted values. We can assign the output of the `sort()` command to a variable. In the workspace add the line

```
reg.sorted = sort(regular)
```

When you run it, it will add a new variable to the environment, namely `reg.sorted`.

Incidentally R used to use the symbol <– for value assignments. It still works, so you could write

```
reg.sorted <- sort(regular)
```

instead. When looking up R tips online you'll probably encounter this in a lot of places because it was the format for so long.

The result of the `sort()` function is a list of the same length as the original variable. If we assign the output of the `mean()` function to a variable the result is a single number, or a *scalar*. You can verify this by entering

```
z = mean(regular)
```

and you'll see the number 143.7594… appears in the Environment tab.

In all these cases R is executing a function and returning an *object* which is assigned to a variable name. In the case of `sort()` the object returned is a list. In the case of `mean()` the object is a scalar. Many of the commands we will use will return more complex objects which may contain multiple lists and matrices of information.

Enter the following lines in the workspace and run them.

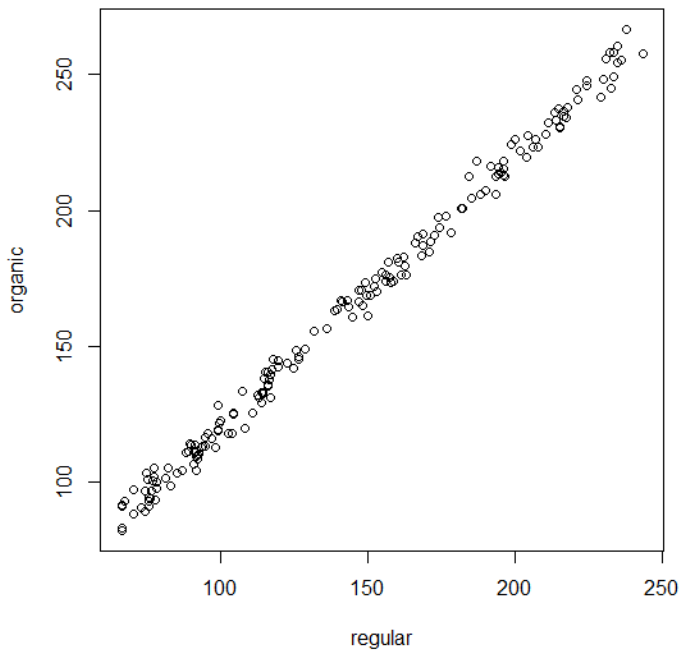
```
lreg = log(regular)
x = data.frame(organic, regular, lreg)
cov.x = cov(x)
```

The first line takes the log of regular prices, and the second line puts the three variables into a data frame called x. The third line computes the covariance matrix of x. The object returned (`cov.x`) is a matrix which you can see by typing `>cov.x` at the command prompt.

```
> cov.x
          organic    regular       lreg
organic 2655.22451 2651.60601 19.1020244
regular 2651.60601 2663.39548 19.1971273
lreg      19.10202   19.19713  0.1419593
>
```

The top left number is the variance of organic fruit, the next number along is the covariance of organic and regular, and the third number in the top row is the covariance of organic and the log of regular. The other rows have similar interpretations. (You might have noticed that this is identical to the output of the `var(x)` command). If you want the correlation coefficients instead you can use the `cor()` command.

Now we will look at an example of a more complex R object. We will begin by plotting the two price series against each other. Type `>plot(regular,organic)` at the command prompt and you should see a graph like this appear:



One of the most important tools we will learn in this course is how to construct a line of best fit between two variables. This is called *linear regression* and we will study the methods in great detail. For now, all you need to know is that R has a tool for running a straight line through a set of data like the one shown above, with the intercept and slope selected so it provides the best possible fit to the observations (in a sense to be defined later on). We can access this function using the *linear model* command, or `lm()`. At the command prompt type

```
> lm(organic ~ regular)
```

The "~" tilde character is important. It tells R that the variable on the left is a function of the variable on the right, so we want an equation of the form $organic = a + b \times regular$ where a is the intercept and b is the slope coefficient.

The following will appear

```
Call:
lm(formula = organic ~ regular)

Coefficients:
(Intercept)      regular
    20.6858       0.9956
```

This tells us that the function

$$organic = 20.6858 + 0.9956 \times regular$$

would fit the above data very nicely. But what would we get if we assigned the `lm()` function output to a variable? Let's see. At the command prompt type

```
> w = lm(organic ~ regular)
```

which assigns the output to an object called w. Now type `> w` and hit enter. You'll see the same output as before. But if you look at the Environment pane (top right) you'll see that w actually consists of a list of 12 items. To see more of them we need to use the `summary()` function. Type the following:

```
> summary(w)
```

You'll see the following output:

```
> summary(w)

Call:
lm(formula = organic ~ regular)

Residuals:
    Min      1Q  Median      3Q     Max
-8.7439 -2.6528 -0.4993  2.8314 11.2923

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 20.685806   0.866590   23.87   <2e-16 ***
regular      0.995574   0.005675  175.42   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.93 on 179 degrees of freedom
Multiple R-squared:  0.9942,    Adjusted R-squared:  0.9942
F-statistic: 3.077e+04 on 1 and 179 DF,  p-value: < 2.2e-16
```

The coefficients are there (under the heading Estimate) but there is a lot more information as well. We will soon learn what all that information means. If you click on `w` in the Environment pane you'll see a window open up beside it with a list of even more elements inside `w`. These will be useful when we are writing programs that make use of the outputs of functions like `lm()`.

5 CREATING PLOTS AND CHARTS

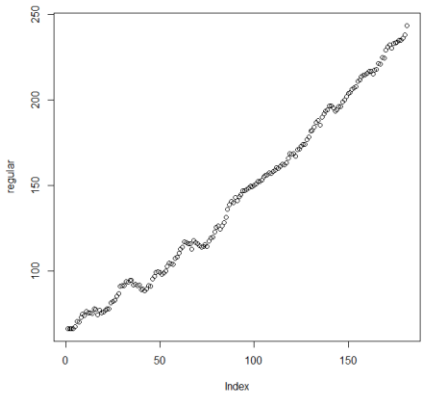
5.1 PLOTS

Before doing most analyses it is a good idea to look at your data. The plot and histogram commands are helpful in R for this purpose. Also you will need to generate graphs for some assignments. Although it is possible to tell R to save a graph to a PDF file (or other formats), for the purposes of this course it is acceptable to make a graph in the console window and then either copy it using the File menu or the right-mouse-click option, or use the Windows snipping tool, and then paste it into your assignments.

Continuing with the example of the fruit data, let's look at the chart of each price series over time. Consult the last Step for instructions how to read the fruit data into R Studio and assign it the "regular" and "organic" variable names. At the command prompt type

```
> plot(regular)
```

and you should see a window open up showing:

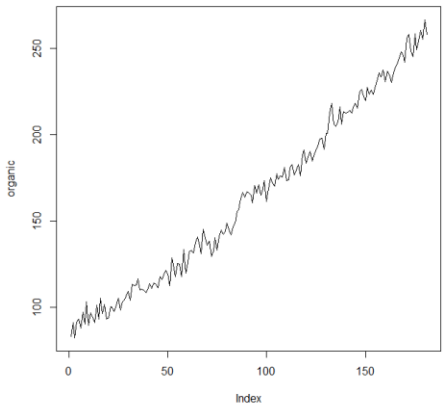


Notice that we didn't ask R t plot the series against another variable, we just asked for a plot of the data in the order of the data set. So the x-axis is just labeled (Index) which means the observation number.

The default format in R is to represent each data point as a dot. If you'd prefer a line chart, we can do that for the organic data using

```
> plot(organic, type="l")
```

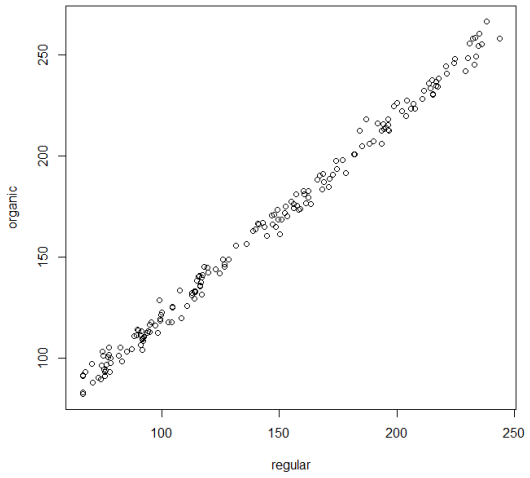
which yields



These are univariate (single-variable) plots that show how each series changes across the sample. We already saw how the series correlate with each other by when we graphed an X-Y scatter plot of one against the other. Type

```
>plot(regular,organic)
```

which puts regular on the horizontal axis and organic on the vertical axis. The result looks like

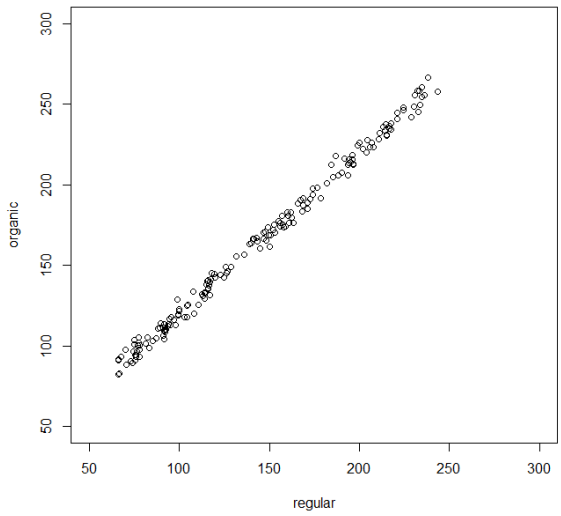


which indicates that they are quite highly correlated.

Notice that the axes tick marks are not quite equally spaced. Sometimes we will want to see exactly how the data are positioned against each other. To force R to use specific x and y axis limits, such as 50 to 300, use the following:

```
>plot(regular,organic, xlim=c(50,300), ylim=c(50,300))
```

Notice that in the xlim and ylim options we are passing a pair of numbers to R so we need to group them in a c() format. The result is

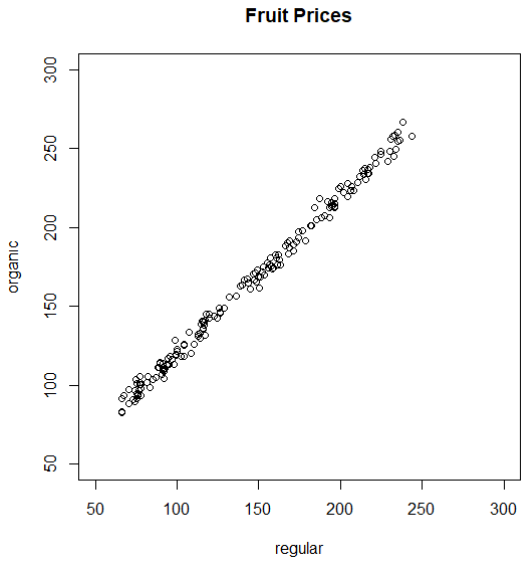


Now the axes are equalized. This way we can see that the vertical intercept is positive, meaning that organic oranges always cost a bit more than regular oranges.

Now let's give the graph a title. Type in

```
>plot(regular,organic, xlim=c(50,300), ylim=c(50,300), main="Fruit
Prices")
```

This yields



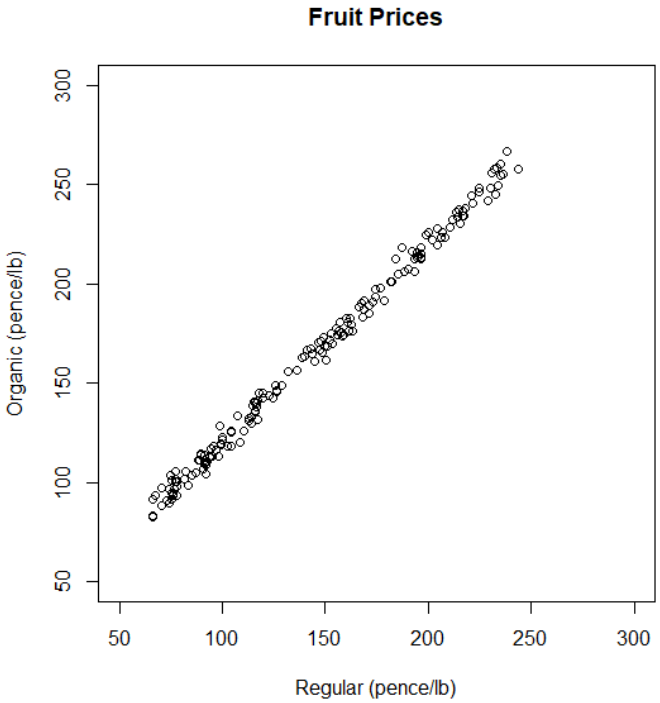
TIP: If you are getting strange error messages, it might be because of the "smartquote" issue – see the note in the Introduction.

Since our command line is getting a bit long, but we will want to use more options, let's switch to the work space. Copy the above command line into the workspace, and enter a line break before "main =". You'll see that R indents the second line, which is helpful for keeping track of the fact that it is part of the same command. Until R encounters the final ")" it knows that the command isn't fully entered, even if it goes onto another line.

Now let's improve the x- and y-axis labels. Change the set of command lines to

```
plot(regular,organic, xlim=c(50,300), ylim=c(50,300),
     main="Fruit Prices",
     xlab="Regular (pence/lb)",
     ylab="Organic (pence/lb)")
```

The result should be



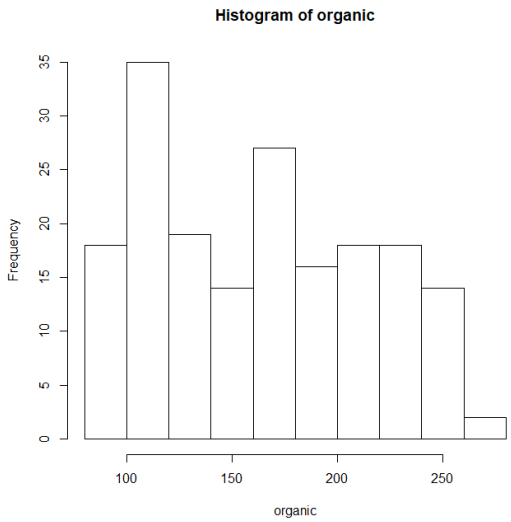
If you are getting some error messages like "Unexpected ')' in: …, check that you have included the commas after each option, and try deleting and re-typing the quotation marks and brackets.

5.2 HISTOGRAMS

To examine how data are distributed we can use histograms. Type

```
>hist(organic)
```

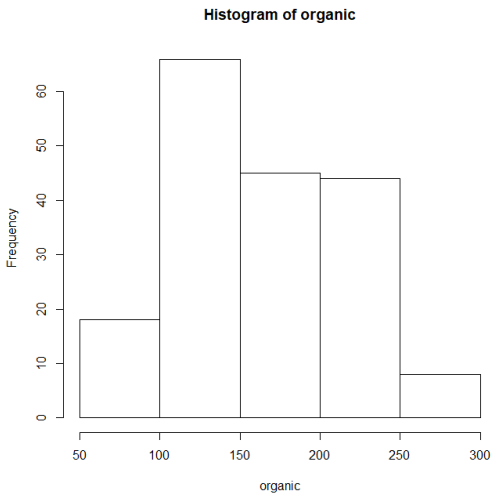
and you'll see



Here is what the graph shows. In the lowest group, up to 100 pence, there are 18 data points. In the next bin, from 100 to 120, there are 35, and so forth. The data are pretty evenly split among bins. Only the last one has relatively few data points. We can use more or fewer bins by applying the "breaks" option. For instance

```
hist(organic, breaks=5)
```

yields fewer bins:



Now the data are divided into 5 bins, each 50 pence wide. Interestingly if we had used "breaks=6" we'd get the same diagram. R will override the specific breaks request in order to make nice-looking bins, and grouping the data into 6 bins would yield ugly intervals. However, if we want to force R to do that, there are ways to do so.

As before you can use options like xlb, ylab and main to change the axis and chart labels.

6 LINEAR REGRESSION IN R

6.1 THE LINEAR MODEL COMMAND

Suppose we want to build a model to explain changes in total consumption. Copy and paste the following lines into the R Studio editor:

```
rm(list=ls(all=TRUE))
inc = read.table("income.csv", sep=",", header=TRUE)
attach(inc)
N=length(Income)
trend=c(1:N)
cycle=sin(trend+Income)
```

The third line "attaches" the data frame inc to the directory so we don't need to keep writing inc$ in front of every variable. The fourth line figures out the sample size N using the `length()` function. The next line constructs a simple trend (the number sequence 1, 2, 3, …, N) and the next generates a nonsense variable using the `sin()` function.

Simple regression

Now we will regress Consumption on Income and put the result in an object called c.model1. We do this using the `lm()` command. You would normally write the equation

$$Consumption = a + b \times Income$$

but in R it is

$$Consumption \sim Income$$

In other words you use ~ instead of = and you don't include a constant or the slope coefficient. R will add them in automatically. The command is:

```
c.model1 = lm(Consumption ~ Income)
```

This estimates the linear model $Consumption = a + b \times Income$ and puts the output in the object c.model1. Add that line to the command file and run it. You will see c.model1 appear in the environment. You can view its contents by writing `summary(c.model1)` at the command line in the console (or putting it in the command file and running it). The output will be:

```
Call:
lm(formula = Consumption ~ Income)

Residuals:
     Min       1Q   Median       3Q      Max
-0.04683 -0.01173  0.00085  0.01178  0.03252

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.110026   0.025047  -4.393 2.01e-05 ***
Income       1.001918   0.003254 307.925  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.01611 on 162 degrees of freedom
Multiple R-squared:  0.9983,    Adjusted R-squared:  0.9983
F-statistic: 9.482e+04 on 1 and 162 DF,  p-value: < 2.2e-16
```

You can see that R computes the coefficient estimates($\hat{a} = -0.110$ and $\hat{b} = 1.002$ in this case), the standard errors, the t-statistics and the p-values, plus the R-squared (which it calls Multiple R-squared), the F statistic for the R^2 and the p-value for the F statistic.

Multiple regression

We will now expand the model to include the Trend and Cycle variables, and put the results in an object called c.model2. You would normally write the equation

$$Consumption = a + b_1 \times Income + b_2 Trend + b_3 Cycle$$

but in R you will write

$$Consumption \sim Income + Trend + Cycle$$

Add these lines to your command file:

```
c.model2 = lm(Consumption ~ Income + trend + cycle)
summary(c.model2)
```

When you run them you should get:

```
lm(formula = Consumption ~ Income + trend + cycle)

Residuals:
      Min        1Q    Median        3Q       Max
-0.031911 -0.006141  0.000946  0.007989  0.025729

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.5413369  0.1260233   12.23   <2e-16 ***
Income       0.7663138  0.0179507   42.69   <2e-16 ***
trend        0.0019390  0.0001466   13.23   <2e-16 ***
cycle       -0.0002837  0.0012360   -0.23    0.819
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.0112 on 160 degrees of freedom
Multiple R-squared:  0.9992,    Adjusted R-squared:  0.9992
F-statistic: 6.543e+04 on 3 and 160 DF,  p-value: < 2.2e-16
```

6.2 FINDING ELEMENTS OF REGRESSION OUTPUT USING THE $ CHARACTER

Suppose you have run a regression and assigned the output to the object "w". To extract specific parts of the regression output you use the prefix 'w$'. For instance, to see the regression coefficient matrix use

```
w$coef
```

and to get the residuals use

```
w$residuals
```

Some of the outputs you want from a regression are in the lm object itself (in this case w), and some are in the "summary" object. For example:

```
w$fitted                    (fitted values)
w$df                        (degrees of freedom)
summary(w)$r.squared
summary(w)$adj.r.squared
summary(w)$fstatistic
```

You can use these as variables too. So if you want the sum of squared residuals you would write

```
sum(w$residuals^2)
```

If you want to get the coefficient estimates, standard errors and t statistics from c.model2 above, they can be found in a matrix called 'coef' in the `summary(c.model2)` object. Type

```
> summary(c.model2)$coef
```

and you'll see the matrix appear. If want want to get R to make a nice-looking table we need to do three things: extract the specific columns we want, round the numbers off, and assign it to an object. The function `round(x,3)` takes the scalar, variable or matrix x and rounds it off to three decimal places. To get the output we want put the following line in the workspace and run it:

cm = round(`summary(c.model2)$coef` **`[1:4,1:3] , 3)`**

where I've put the new parts in **bold** so you can see them, but you don't have to do that when you type the command in R. Reading from the center outwards this command gets the $coef matrix from the summary of c.model2, takes the 4 rows and first 3 columns, rounds the numbers off to 3 decimal places and stores it as a matrix cm.

If you type `>cm` you will see the output

	Estimate	Std. Error	t value
(Intercept)	1.541	0.126	12.231
Income	0.766	0.018	42.690
trend	0.002	0.000	13.231
cycle	0.000	0.001	-0.230

Now add a line to write this matrix to a CSV file:

```
write.csv(cm, file = "cm.csv")
```

When you run it, if you look in your active folder you'll see the file cm.csv has appeared. It contains the above table which you can copy and paste into Word and then format into a neat-looking results table. Note however that Excel drops trailing zeroes so you will need to add them in again.

You'll notice in the summary() output for the lm() function that the Residual standard error is printed. You can verify the calculation using the following:

```
sqrt( sum(w$residuals^2)/w$df )
```

Reading from the center outwards, the above command takes the residuals, squares them, sums them up, divides by the degrees of freedom (*N – k – 1*), then takes the square root of the result.

str(w) – list the contents of an object
If you want a list of all the stuff you can get from an object w, type `str(w)` and `str(summary(w))`.

6.3 COMMENTING WITH THE # SYMBOL

Suppose you write the following program

```
sink("inc_out.txt")
cat("Start of program", "\n")
rm(list=ls(all=TRUE))
inc = read.table("income.csv", sep=",", header=TRUE)
attach(inc)
ci = lm(Consumption ~ Income)
print(summary(ci))
cat("End of program", "\n")
sink()
```

If you copy it and run it you will see that it reads in some data, computes an lm object (a fitted line) and writes it to an output file. You'll see a new file in your files list called inc_out.txt, which contains the summary of the lm object.

As before, if the code doesn't run, double check all the quotation marks and brackets to make sure they are not smartquotes or MSWord characters.

The code is hard to understand. If you wrote it a few weeks ago and went back to it, you would probably have trouble understanding it. So it is a good idea to make your code readable using white spaces and the # character.

You can use empty lines to space things out or group sections, and you can use the # character to insert comments, even partway through a command line, in order to make your code readable. You will find the extra time spent making your code pretty is worth it when you go back to it a day or two later and try to figure out what you were doing. I would write the code something like this.

```
############################################################
#
#      This is a program to read income data and fit a
#      consumption function
#
############################################################

cat("Start of program", "\n")

# Open the output file
       sink("inc_out.txt")

# Clear the memory
       rm(list=ls(all=TRUE))
```

```
# Read external data file and attach the variables
       inc = read.table("income.csv",sep=",",header=TRUE)
       attach(inc)

# Estimate the consumption function
       ci = lm(Consumption ~ Income)
       print(summary(ci))

# Close the output file
       sink()

cat("End of program", "\n")
############################################################
```

You can vary the style in whatever way you like best.

6.4 How to Forbid Printing in Scientific Notation

Sometimes R generates output in scientific notation, e.g. 4.667e+02 rather than 466.7. If your data are on very different numerical scales this will usually happen. If this makes the results hard to read you can turn this option off. Insert the following command at the start of your program:

```
# forbid printing in scientific notation
options(scipen=999)
```

If you do this, however, you may need to rescale your data in order to make the regression results readable.

6.5 Reporting your results

When you write assignments you will be reporting the results of calculations including regression models. It is NOT acceptable to report regression results by taking a screenshot of R output or a dump of your session history log. You have to write up results in a way that makes it easy for your readers to follow. This is more work, but it is essential to communicating the results of your data analysis in a compelling way.

Here is a sample of how I want you to report your output. Suppose you are asked to use a file called school_costs.xls and answer the following assignment question.

1. **[10]** Run a multiple regression of total operating costs per student on the number of teachers, number of students, number of special needs pupils and the rural dummy variable. Report your results. Which factors tend to raise the average cost per student of operating a school? Do rural schools have significantly higher or lower operating costs?

Here is what a good answer looks like:

I obtained the data in the file school_costs.xls, which consists of 411 observations on a sample of elementary school characteristics and operating costs. I used the following variables:

- COSTS: Total operating cost of elementary school in 2016 in thousands of dollars
- TEACHERS: Number of full time equivalent teachers on staff that year
- STUDENTS: total number of students enrolled in that school in that year
- SPECIAL: fraction of student body classified as special needs requiring an Assistant
- RURAL: dummy variable indicating if the school is located in a rural (1) or urban (0) area

I constructed the dependent variable AVGCOST = COSTS / STUDENTS, representing total operating cost per student in thousands of dollars. I then estimated the following linear regression model

$$AVGCOSTS_i = \alpha + \beta_1 TEACHERS_i + \beta_2 STUDENTS_i + \beta_3 SPECIAL_i + \beta_4 RURAL_i + e_i$$

where i denotes an individual observation and e_i is the regression residual term. The results were as follows.

Coefficient	Variable	Estimate	Std Error	t-statistic
α	Constant	10.442	1.441	7.25***
β_1	TEACHERS	1.022	0.263	3.89***
β_2	STUDENTS	-0.368	0.447	-0.82
β_3	SPECIAL	0.326	0.142	2.30**
β_4	RURAL	-0.104	0.296	-0.35

Note that ** denotes significant at 5%, *** denotes significant at 1%.

The R^2 value was 0.688 and the F statistic was 178.6, which is significant at 1%.

The coefficients on the number of teachers and the number of special needs students were both significant at the 5 percent level. The results indicate that schools with larger staffs tend to have higher costs per student, as do schools with more special needs students. Schools with larger enrolment have slightly lower average costs but the effect is not significant. Rural schools also have slightly lower average costs but again the effect is not significant.

7 INSTALLING AND RUNNING PACKAGES

R Studio comes with most of the functions you will need for this course, but not quite all. There is a large library of specialized packages you can download and run that increase the functionality of R. One particularly useful package for this course is called <car> which stands for Companion to Applied Regression. Among other things it allows us to test linear hypotheses about regression results.

To download a package, select the 'packages' heading in the bottom right console window in R Studio, then click on the Install button. Type the name of the package you want to install and hit the Install button. Alternatively, you can go to **Tools > Install Packages…** in the top menu to get to this point. Note that as you type the package name R Studio will prompt you with package names.

Once the package has installed you need to issue a command to tell R to load it and make it available. You do this using `library()`:

```
> library(car)
```

The `library()` command is needed at the start of any command file you will run if you are going to use functions contained in that package.

Now that <car> is installed and open, you can use the `linearHypothesis()` function (we will study linear hypothesis testing in Unit 6). Load and run the following program in R.

```
library(car)
cat("Start of program", "\n")
# Clear the memory
      rm(list=ls(all=TRUE))
# Read external data file and attach the variables
      inc = read.table("income.csv",sep=",",header=TRUE)
      attach(inc)
# Make the additional variables
      N=length(Income)
      trend=c(1:N)
      cycle=sin(trend+Income)
# Estimate the augmented consumption function
      c.model = lm(Consumption ~ Income+trend+cycle)
      print(summary(c.model))
cat("End of program", "\n")
```

Now we can use the `linearHypothesis()` command to test if the coefficient on Income = 1 and if the coefficient on cycle = 0. The format is

```
linearHypothesis(lm object, c("test 1", "test 2"))
```

where test 1, test 2, etc are linear expressions we want to test.

Type in:

```
> linearHypothesis(c.model, c("cycle = 0", "Income=1"))
```

and the output will be

```
Linear hypothesis test

Hypothesis:
cycle = 0
Income = 1

Model 1: restricted model
Model 2: Consumption ~ Income + trend + cycle

  Res.Df      RSS Df Sum of Sq      F    Pr(>F)
1    162 0.041320
2    160 0.020066  2  0.021254 84.737 < 2.2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The F statistic is 84.737 so the test rejects.

8 R COMMANDS FOR LATER IN THE COURSE

8.1 LINEAR HYPOTHESIS TESTS

For this step you need to install the package <car> from the R library (see Section 7 for instructions on installing and loading packages.) Suppose you put the regression results for the following model in the object "m1".

$$WAGE = a_0 + a_1 MALE + a_2 EDUCATION + a_3 TEST + a_4 SF + a_5 SM + e$$

where *WAGE* is a list of wage rates of a sample of workers, *MALE* is a dummy variable if the worker is male, *EDUCATION* is the worker's level of schooling, *TEST*is the worker's score on an aptitude test, *SF* is the level of schooling of the worker's father and *SF* is the level of schooling of the worker's mother.

Once you have installed the <car> package, you can test linear hypotheses using the command

```
linearHypothesis(m1, test expression)
```

The test expression is composed as follows. To test that the coefficient on TEST is zero you would enter

```
linearHypothesis(m1, "TEST=0")
```

To test multiple hypotheses at the same time, combine them in a c() vector. For instance, to test that the coefficient on *TEST* is zero and the coefficient on *MALE* equals 1, write

```
linearHypothesis(m1, c("TEST=0", "MALE=1"))
```

To test whether two coefficients are equal to each other, such as *SF* and *SM*, you can use

```
linearHypothesis(m1, "SF = SM")
```

The output of the `linearHypothesis` command includes the sums of squares for each version of the model (original and restricted), the *F* test of the restriction and the *p* value for the *F*. If the test can't be run as an *F* test it will be run as a Wald test and the chi-squared score will be reported, with the associated *p*-value.

8.2 HETEROSKEDASTICITY METHODS

8.2.1 Ordering data

For the purpose of heteroskedasticity testing you need to be able to order a data set according to the values of an explanatory variable. I will show you how using some simulated data. Generate a series of numbers from 1 to 100 using:

```
a = c(1:100)
```

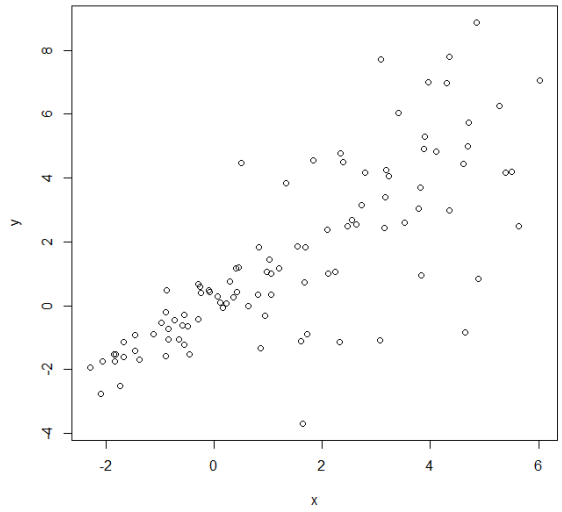
Now we generate a set of 100 random error terms. The first 50 will be N(0,0.5) and the next 50 will be N(0,2), so the variances are higher in the second half of the sample.

```
e = c( rnorm(50,0, 0.5), rnorm(50,0,2) )
```

Now we will generate an x variable with some variance in it, and then a simple y variable with random noise.

```
x = a/10 + log(a) + sin(a)
y = x + e
```

If you plot y against x using `plot(y~x)` it will look something like this:



though the specific shape will depend on the random numbers R generated for you. You can see visually that this data set is heteroskedastic, so if we run a regression using `lm(y~x)` we will get an unbiased slope coefficient (which should be close to 1) but the variance estimate will be biased.

The first step of testing for heteroskedasticity requires ordering the data. If you print out the x variable it will not be in order from smallest to largest. Type in the following command:

```
order(x)
```

This will yield something like this:

```
> order(x)
  [1]  11  17   5  10  18  12   4  24  23   6  16  30  29  36  22  19   9  25  13  31   3
 [22]   7  15  37  35  42  43   8  28  14  21  20  49  26  48  41  32  27  34   2  38  55
 [43]  44  33  50  54   1  47  61  56  40  39  62  45  60  67  46  68  53  51  57  74  52
 [64]  63  73  66  59  69  80  58  75  81  79  65  64  72  86  87  70  71  93  76  85  92
 [85]  78  82  99  77  88  94  98  91  84  83 100  89  90  97  95  96
```

Each entry tells you the rank of the corresponding element of x from lowest to highest. So the first element of x (in my data set) is 11th smallest, the second is 17th smallest, third is 5th smallest, etc. Notice that element #47 is the smallest (has the order number 1).

Now I can assign that list to a new variable h:

```
h = order(x)
```

Then if I use h to order x and y they will both be ranked according to x. To do that, I refer to each variable but add [h] inside square brackets: `x[h]` and `y[h]`. The term x[h] tells R to use the elements of x but sort them according to the elements in h. The first element of x will be put in position 11, the second element of x will be put in position 17, etc. The 47th number in x corresponds to the position where the number 1 is found in h, so it will be first in the list.

My x data, unsorted and sorted, look like the following:

```
> h = order(x)
> x
  [1]  0.94147098  0.41615025 -0.65749228 -1.74309686 -2.06836219 -1.47117497 -0.58892355
  [8] -0.29008330 -0.88510609 -1.84660620 -2.29788548 -1.82147957 -0.84478232 -0.24844997
 [15] -0.55776236 -1.46049204 -2.09461084 -1.84135900 -0.89456177 -0.08278702 -0.10786680
 [22] -0.89989376 -1.68171462 -1.68363219 -0.85122757  0.10446191  0.36053906 -0.26129872
 [29] -1.13092971 -1.38922901 -0.73802485  0.28569078  0.80340430  0.40272216 -0.48353073
 [36] -0.97529779 -0.55445605  0.45878242  1.20023374  1.05623371  0.22780526 -0.45419117
 [43] -0.29297486  0.63351229  1.54424103  1.67314695  0.97342552  0.16054433  0.05442705
 [50]  0.82560214  1.83840354  2.23538387  1.72563324  0.85222690  0.49291164  1.05309731
 [57]  2.09311349  2.73242964  2.45920056  1.60084482  1.02300837  1.33368492  2.32422097
 [64]  3.16114295  3.15244141  2.38379410  1.63978740  1.68256461  2.55110868  3.52539544
 [71]  3.78837478  3.17715724  2.33276860  2.11078865  2.79473025  3.83537430  4.35571474
 [78]  3.95726963  3.08643948  2.62408471  3.07566285  4.10650954  4.84952385  4.70237352
 [85]  3.88127312  3.22219426  3.41227404  4.35806149  5.27143304  5.39418699  4.69512801
 [92]  3.89874535  3.81911837  4.61145323  5.62938482  6.01923955  5.50489676  4.64165065
 [99]  4.30567332  4.88846417
> x[h]
  [1] -2.29788548 -2.09461084 -2.06836219 -1.84660620 -1.84135900 -1.82147957 -1.74309686
  [8] -1.68363219 -1.68171462 -1.47117497 -1.46049204 -1.38922901 -1.13092971 -0.97529779
 [15] -0.89989376 -0.89456177 -0.88510609 -0.85122757 -0.84478232 -0.73802485 -0.65749228
 [22] -0.58892355 -0.55776236 -0.55445605 -0.48353073 -0.45419117 -0.29297486 -0.29008330
 [29] -0.26129872 -0.24844997 -0.10786680 -0.08278702  0.05442705  0.10446191  0.16054433
 [36]  0.22780526  0.28569078  0.36053906  0.40272216  0.41615025  0.45878242  0.49291164
 [43]  0.63351229  0.80340430  0.82560214  0.85222690  0.94147098  0.97342552  1.02300837
 [50]  1.05309731  1.05623371  1.20023374  1.33368492  1.54424103  1.60084482  1.63978740
 [57]  1.67314695  1.68256461  1.72563324  1.83840354  2.09311349  2.11078865  2.23538387
 [64]  2.32422097  2.33276860  2.38379410  2.45920056  2.55110868  2.62408471  2.73242964
 [71]  2.79473025  3.07566285  3.08643948  3.15244141  3.16114295  3.17715724  3.22219426
 [78]  3.41227404  3.52539544  3.78837478  3.81911837  3.83537430  3.88127312  3.89874535
 [85]  3.95726963  4.10650954  4.30567332  4.35571474  4.35806149  4.61145323  4.64165065
 [92]  4.69512801  4.70237352  4.84952385  4.88846417  5.27143304  5.39418699  5.50489676
 [99]  5.62938482  6.01923955
>
```

Now suppose I want to sort the data according to x, then run the regression of y on x only for the first 40 observations. The commands and output are ("xs" means x – sorted; etc)

```
> xs = x[h]
> ys = y[h]
> dl = lm(ys[1:40] ~ xs[1:40])
> summary(dl)

Call:
lm(formula = ys[1:40] ~ xs[1:40])

Residuals:
     Min       1Q   Median       3Q      Max
-1.23037 -0.32370  0.04231  0.32373  1.24130

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.1805     0.1160   1.556    0.128
xs[1:40]      1.0564     0.1054  10.021 3.22e-12 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.5173 on 38 degrees of freedom
Multiple R-squared:  0.7255,    Adjusted R-squared:  0.7183
F-statistic: 100.4 on 1 and 38 DF,  p-value: 3.218e-12
```

Note that to get R to use the 1st 40 observations of the new variables xs and ys I refer to xs[1:40] and ys[1:40]. If I wanted to use the last 40 observations I would refer to xs[61:100] and ys[61:100].

8.2.2 Tests for Heteroskedasticity

Goldfeld-Quandt

The hard way:
Using the notation from the textbook (pp. 132-133) and the data from the above example, note that $N = 100$, $d = 20$ and $k = 1$. Regress the first 40 observations of y on x and retrieve the SSR using

```
d1 = lm(ys[1:40] ~ xs[1:40])
SSR1 = sum(d1$residuals^2)
```

then do the same for the last 40 observations using

```
d2 = lm(ys[61:100] ~ xs[61:100])
SSR2 = sum(d1$residuals^2)
```

Form the Goldfeld-Quandt F statistic and compute the degrees of freedom using

```
GQ = SSR2 / SSR1
df = 0.5*(100 - 20 - 2 -2)
```

Now find the p-value using

```
1 - pf(GQ,df,df)
```

The easy way:

Download, install and run the package <lmtest>. Run the regression of *y* on *x* and call the lm object "m1" (or any other name you choose). Run the test based on removing the middle 20% of the data using the command

```
gqtest(m1, fraction=0.2, order.by=x)
```

You can verify the two ways are equivalent by running the following code:

```
rm(list=ls(all=TRUE))

# ARTIFICIAL DATA
a = c(1:100)
e = c( rnorm(50,0, 0.5), rnorm(50,0,2) )
x = a/10 + log(a) + sin(a)
y = x + e

# GQ TEST STEP-BY-STEP
h = order(x)
xs = x[h]
ys = y[h]
d1 = lm(ys[1:40] ~ xs[1:40])
  SSR1 = sum(d1$residuals^2)
d2 = lm(ys[61:100] ~ xs[61:100])
  SSR2 = sum(d2$residuals^2)
GQ = SSR2 / SSR1
df = 0.5*(100-20-2-2)
cat("\n", "   GOLDFELD_QUANDT TEST", "\n")
cat("GQ =", GQ, ", df =", df, ", p-value =", 1 - pf(GQ,df,df), "\n" )

# GQ TEST USING LMTEST PACKAGE
library(lmtest)
m1 = lm(y~x)
print( gqtest(m1, fraction=0.2, order.by=x) )
```

Breusch-Pagan Test

The package <lmtest> includes this one, so if your lm object is called "m1" just run

```
bptest(m1)
```

Note that this is in the "studentized" form (using $N \times R^2$) which is the preferred specification nowadays. The Koop text uses $RSS/2$ which is an older formula.

White's Test

The same command will run White's test but you need to specify the variables to include as Z variables. If you have more than a couple of right-hand side variables, instead of using all the squares and cross-products it is customary to use the squared fitted values of *y* from the original regression. Generate the fitted values from m1, call them, say, y1, and run

```
y2 = y1^2
bptest(m1, ~y1 + y2)
```

This is also in the form $N \times R^2$ which is why you can use the `bptest` command in R to compute it. It will be called the "Breusch-Pagan" statistic in the printout but in this case it's the White's statistic. There doesn't seem to be a package that computes a White's statistic in R for an lm object.

If you're wondering why the BP formula works to generate the White's statistic, recall that the BP regression first involves multiplying the squared OLS residuals by the maximum likelihood estimate of the error variance. But this is a constant across the sample, and multiplying the dependent variable by a constant doesn't change the fit of the model (i.e. the R^2). Since we are only using the R^2 and the sample size, it doesn't make a difference for this particular test statistic.

The following code uses the forest.csv data set, runs a GQ test, and then constructs the BP Test and White's test step-by-step, as well as using the `bptest` command to show the equivalent results.

```
### READ IN DATA AND CONSTRUCT VARIABLES
rm(list=ls(all=TRUE))
library(lmtest)
forest=read.table("forest.csv", sep=",", skip=1)
Deforest=forest$V1
Popden=forest$V2
Cropland=forest$V3
Pasture=forest$V4

### MODEL TO BE TESTED FOR HETEROSKEDASTICITY
m1 = lm(Deforest ~ Popden + Cropland + Pasture)

### GQ TEST
gqtest(m1, fraction=0.2, order.by=Popden)

### BP TEST: Step-by-step then using <lmtest>
# COMPONENTS OF BP STAT
e2 = m1$residuals^2
N = length(e2)
sig2 = sum(e2)/N
y_bp = e2/sig2
# BP TEST REGRESSION
bp.reg = lm(y_bp ~ Popden + Cropland + Pasture)
bp.R2 = summary(bp.reg)$r.squared
# BP STATISTIC
bp.stat = N*bp.R2
cat("BP Stat = ", bp.stat, "\n")
# BP STATISTIC USING <lmtest> package
bptest(m1)

### WHITES TEST: Step-by-step then using <lmtest>
# COMPONENTS OF WHITES STAT
y_w = e2
# WHITES TEST REGRESSION
y1 = m1$fitted.values
y2 = y1^2
w.reg = lm(y_w ~ y1+y2)
```

```
w.R2 = summary(w.reg)$r.squared
# WHITES STATISTIC
w.stat = N*w.R2
cat("Whites Stat = ", w.stat, "\n")
# WHITES STATISTIC USING <lmtest> package
bptest(m1, ~ y1+y2)
```

8.2.3 Heteroskedasticity-Consistent Estimation

A popular approach for handling heteroskedastic errors is to use a variance estimator that is heteroskedasticity-consistent (HC). Two examples are the White's and Newey-West methods.

White's

Suppose you have estimated a linear model and assigned it to the variable name "m1". For instance, suppose we use the forest.csv data as in Section 8.2.2 and estimate

```
m1 = lm(Deforest ~ Popden + Cropland + Pasture)
```

Using `summary(m1)` you can see that the OLS standard error on Popden is around 0.000114.

Download the <sandwich> package and activate it using `library(sandwich)`, then type the following at the command line:

```
>vcovHC(m1)
```

to get the HC variances. Using data from the above example you will see something like:

```
> vcovHC(m1)
             (Intercept)        Popden      Cropland       Pasture
(Intercept)  8.920448e-03 -1.451178e-05 -3.410317e-04  3.820385e-04
Popden      -1.451178e-05  5.837729e-08 -1.940165e-07 -1.881777e-06
Cropland    -3.410317e-04 -1.940165e-07  6.818883e-05 -1.327951e-05
Pasture      3.820385e-04 -1.881777e-06 -1.327951e-05  2.277254e-04
>
```

This is the matrix of variances and covariances of the beta coefficients. The elements down the diagonal are the variances, so look for the one on, say, Popden and take the square root to get the standard deviation. In this example the HC standard deviation for the coefficient on Popden is the square root of 5.838e-08 which is 0.000242, which is larger than the OLS estimate of 0.000114.

You can get R to generate the new t statistics as follows.

```
vv = vcovHC(m1)
var_vv = diag(vv)
sd = sqrt(var_vv)
new_t = m1$coef / sd
cat("White's Standard Errors and t-stats:", "\n")
print(sd)
```

```
print(new_t)
```

The first line assigns the Newey-West variances to the matrix 'vv'. The second line extracts the diagonal elements of the resulting matrix and assigns them to the variable 'var_vv'.The third line computes the square roots of these numbers. The fourth line divides the slope coefficients from the 'm1' model by the White's standard deviations and calls the result new_t. The final lines print out the results. You should get

```
> vv = vcovHC(m1)
> var_vv = diag(vv)
> sd = sqrt(var_vv)
> new_t = m1$coef / sd
> cat("White's Standard Errors and t-stats:", "\n")
White's Standard Errors and t-stats:
> print(sd)
 (Intercept)       Popden     Cropland      Pasture
0.0944481236 0.0002416139 0.0082576526 0.0150905723
> print(new_t)
(Intercept)       Popden     Cropland      Pasture
  5.9892582   3.3431157  -0.4813436   1.8532098
```

Newey-West
The steps are exactly the same (including installing/activating the sandwich package) but the first line instead uses `NeweyWest()` instead of `vcovHC()`:

```
vv = NeweyWest(m1)
var_vv = diag(vv)
sd = sqrt(var_vv)
new_t = m1$coef / sd
cat("Newey-West Standard Errors and t-stats:", "\n")
print(sd)
print(new_t)
```

8.3 COMPUTING A TAIL PROBABILITY

Suppose you generate an F statistic of 2.13 on a Goldfeld-Quandt test with (30, 30) degrees of freedom. You want to know what is the *p*-value, in other words the probability of obtaining $F_{30,30} \geq 2.13$. The command

```
> pf(2.13,30,30)
```

in R means "compute the area under the F probability function up to 2.13 for 30,30 degrees of freedom". That will give you 0.9788. We want the upper tail, so instead compute

```
1 - pf(2.13,30,30)
```

which is 0.0212. This would imply the statistic is significant at a 5% level.

The corresponding function for a normal variable is $pnorm(z, \mu, s)$ where z is the test value and the other two parameters are the mean and standard deviation. To see how well you understand the function, predict what you'd get from entering

```
1 - pnorm(1.96,0,1)
```

then see how well your prediction worked out.

You can also get $\chi^2(p)$ tail probabilities using

```
1 - pchi(x,p)
```

which computes the probability of observing a value greater than or equal to x when it is distributed $\chi^2(p)$.

8.4 Autocorrelation Testing and Estimation

8.4.1 Testing for autocorrelation

We will use the data in the file **compute.xls** in the Koop archive. Read in the data and re-save it as a CSV file. I would suggest renaming the variables "ch.sales" and "ch.comp.purch" to get rid of the % signs, since computers sometimes get confused about its meaning and might change it to another character.

Likelihood-based test score (Breusch-Godfrey test)
Run the following code to read in the data:

```
rm(list=ls(all=TRUE))
library(lmtest)
inc = read.table("compute1.csv", sep=",", skip = 1)
ch.sales = inc$V2
ch.comp.purch = inc$V3
N = length(ch.sales)
```

Now you need to regress the change in sales on the change in computer purchases using OLS and get the residuals *e*.

```
m1 = lm(ch.sales ~ ch.comp.purch)
e1 = m1$residuals
```

Next we need to generate the lagged residuals. We do that in R by constructing a new variable that consists of a zero followed by the first *N*-1 values of *e*. Since *e* has a length of 98, the command is

```
lag.e1 = c(0,e1[1:97])
```

Now run the regression described on pp. 145—146 of the text and compute the test statistic $N \times R^2$:

```
m2 = lm(e1 ~ ch.comp.purch + lag.e1)
```

```
bg = N*summary(m2)$r.squared
cat("B-G Test score=", bg, "\n")
```

R can do this test automatically as long as you have the <lmtest> package installed. Simply type:

```
bgtest(m1)
```

and you'll get the same result. With the `bgtest()` command you can also test for higher-order serial correlations, using the `order=` option.

Note that we filled in the missing first lag with a zero. The `bgtest()` command does the same thing by default. But if you are testing for more than a lag-1 autocorrelation you should set the initial values to missing, so R drops those rows altogether. For an AR5 test we would write

```
bgtest(m1, order=5, fill=NA)
```

Durbin-Watson test
The function `durbinWatsonTest(m1)` will compute a DW statistic for the lm object m1. We won't be using this testing procedure however.

8.4.2 Estimating Models Robust to Serial Correlation

Cochrane-Orcutt
Continuing with the above example, install and run the package <orcutt>. Now run the Cochrane-Orcutt procedure on the regression model and assign the output to 'co'.

```
co = cochrane.orcutt(m1)
```

This runs the C-O procedure iteratively until convergence. Use `summary(co)` to get

```
> print(summary(co))
Call:
lm(formula = ch.sales ~ ch.comp.purch)

              Estimate Std. Error t value Pr(>|t|)
(Intercept)   0.025091   0.073108   0.343   0.7322
ch.comp.purch 0.681630   0.031180  21.861   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3828 on 95 degrees of freedom
Multiple R-squared:  0.8342 ,  Adjusted R-squared:  0.8324
F-statistic: 477.9 on 1 and 95 DF,  p-value: < 7.633e-39

Durbin-Watson statistic
(original):    1.20035 , p-value: 1.767e-05
(transformed): 1.61875 , p-value: 8.634e-02
```

Oddly it doesn't print out $\hat{\rho}$ but you can get that by typing `co$rho`

Notice that once we correct the autocorrelation problem the t stat on ch.comp.purch is lower compared to the OLS value.

Newey-West
Newey-West standard errors are heteroskedasticity and autocorrelation-consistent (HAC) so we can use them rather than Cochrane-Orcutt, which are only AR1-consistent. We can adapt the code from section 7.2.3. The full program is now:

```
rm(list=ls(all=TRUE))
inc = read.table("compute1.csv", sep=",", header=TRUE)
attach(inc)

m1 = lm(ch.sales ~ ch.comp.purch)
e = m1$residuals
N=length(e)
e1 = c(0, e[1:97])

m2 = lm(e ~ ch.comp.purch + e1)
bg = N*summary(m2)$r.squared
cat("B-G Test score=", bg, "\n")

print( bgtest(m1))

library(orcutt)
co=cochrane.orcutt(m1)
print(summary(co))

vv = NeweyWest(m1)
sqrt_vv = chol(vv)
sd = diag(sqrt_vv)
new_t = m1$coef / sd
cat("Newey-West Standard Errors and t-stats:", "\n")
print(sd)
print(new_t)
```

8.5 INSTRUMENTAL VARIABLES

To use the IV estimator in R you will need to download and run the <AER> package. We will use the data in the file **rts.xls**. The data are described on page 167 in the textbook (see question 9). As always, convert it to a CSV file. Here is the code to run. The first 3 lines clear the memory, read the data and attach the variables. We then create a dependent variable by taking the log of earnings, then regress log(Earnings) on the variables S, TEST, MALE, ETHBLACK and ETHHISP. You should observe that the coefficient on TEST is 0.0115 and has a t statistic of 7.47. The lm() object is called m1.

```
rm(list=ls(all=TRUE))
library(AER)
rts = read.table("rts.csv", sep=",", header=TRUE)
```

```
attach(rts)

# OLS MODEL
LogEARN = log(EARNINGS)
m1 = lm(LogEARN ~ TEST + S + MALE + ETHBLACK + ETHHISP)
print(summary(m1))
```

Next we create an instrument for TEST which we call TEST.iv, by taking the fitted values from a regression of TEST on SM (schooling of the mother). We conduct the Hausman test by running the original regression with both TEST and TEST.iv, then looking at the t statistic on TEST.iv. This is shown using the summary of the lm object haus. The t statistic on TEST.iv is 1.172 which has a *p* value of 0.2419 which indicates that an IV estimator in this case is not needed.

```
# HAUSMAN TEST
TEST.iv = lm(TEST ~ SM)$fitted.values
haus = lm(LogEARN ~ TEST.iv + TEST + S + MALE + ETHBLACK +
ETHHISP)
print(summary(haus))
```

The next part of the program uses the `ivreg()` routine in R to do the IV regression. The syntax is as follows:

```
ivreg(y ~ endog + exog | exog + instr )
```

where endog denotes the endogenous variable(s), in this case TEST, exog denotes the exogenous variables (S, MALE, ETHBLACK and ETHHISP), and instr denotes the instrument(s), in this case SM. We need to list the exogenous variables on the right of the | so that R knows which one is the endogenous variable to replace using the instruments.

```
#IV REGRESSION
m.iv = ivreg(LogEARN ~ TEST + S + MALE + ETHBLACK + ETHHISP
             | S + MALE + ETHBLACK + ETHHISP  + SM )
print(summary(m.iv, diagnostics=TRUE))
```

The summary of the m.iv object (adding in the option `diagnostics=TRUE`) will show that the Wu-Hausman test score is an F statistic equalling 1.373. This is the same as our result: the square of a t statistic is an F statistic. And you can see that the *p* value is identical to what we computed. The coefficient on TEST in the IV regression is 0.0365

The IV regression looks very different compared to the OLS results. However, the Hausman test tells us that we have no reason to use the IV model since TEST does not appear to be endogenous.

We can try to improve the instrument by including more variables in the explanatory model of TEST. Add SF and SIBLINGS to the instrument list, so the command is now:

```
m.iv = ivreg(LogEARN ~ TEST + S + MALE + ETHBLACK + ETHHISP
          | S + MALE + ETHBLACK + ETHHISP  + SM + SF + SIBLINGS)
```

You will see in the summary of this object that TEST is now significant (p = 0.0363) but the Hausman test still fails to reject the null of exogeneity (p = 0.101).

8.6 Regression Without a Constant

Should you need to, you can modify the regression equation to suppress the constant by adding "`0+`" to the formula:

```
reg.no.con = lm(y ~ 0 + x1 + x2)
```

8.7 Overlaying One Plot on Another

It is sometimes useful to plot more than one line on the same graph, such as when you want to plot a regression fit against the underlying data. To do this you issue two `plot()` commands combined with a `par()` command that tells R not to open a new plot window.

Let's use the data in the file computer.csv, but change the names of the variables to ch_sales and ch_purch.

```
rm(list=ls(all=TRUE))
comp = read.table("computer.csv", sep=",", skip = 1)
ch_sales = comp$V2
ch_purch = comp$V3
```

Now regress ch_sales on ch_purch and call the lm object "m1".

```
m1 = lm(ch_sales ~ ch_purch)
```

Now do an XY-plot of the data, with some extra conditions to make the axes pretty:

```
plot(ch_sales ~ ch_purch,
                ylim=c(-5,6), ylab="% change in sales",
                xlab="% ch computer purch")
```

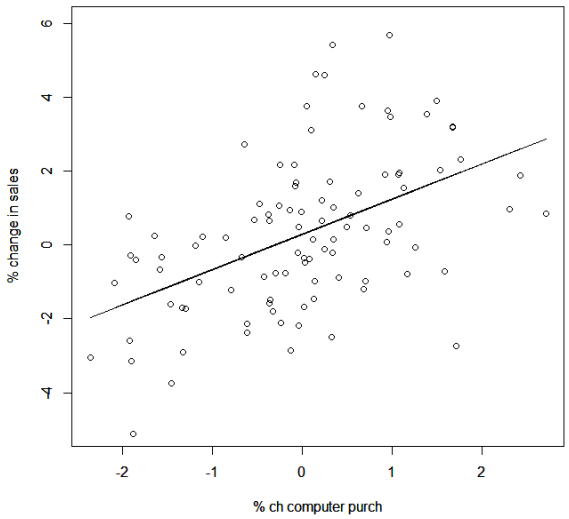
Now tell R you want to plot something else on the same chart:

```
par(new = "TRUE")
```

Now plot the fitted values from the regression on the same chart. We'll tell R to plot this as a type "l" for line, and not to rename the axes:

```
plot(m1$fitted.values ~ ch_purch, type="l",
                      ylim=c(-5,6), ylab="",
                      xlab="")
```

Here's what you get:


% change in sales
6
4
2
0
-2
-4
-2
-1
0
1
2
% ch computer purch

9 BASIC R COMMANDS

apply()	apply a function to every row or column of a data frame
attach(…)	allow variables in data frame to be indexed directly from command level
c(…)	format for list of numbers or words
cat(…)	concatenate the content in the brackets and write to output
cbind(…)	add a new column to a data frame
colnames(x)	rename the columns in data frame x
cor(x,y)	correlation between variables x and y
diag(x)	diagonal elements of matrix x
diff(x)	compute the first differences of x
dim(x)	dimensions of data frame or matrix x
getwd()	show the working directory
head(x)	look at first 5 rows of data frame x; tail(x) displays last 5 rows
ivreg(…|…)	instrumental variables regression
length(x)	Number of observations in variable x, or number of elements in matrix x
library(z)	load package z and make the commands available
lm(…)	linear model or regression
log(x)	take the logarithm of x
ls()	list all the items held in memory
mean(x)	mean of variable x or means of data frame x
ncol(x)	number of columns in data frame or matrix x
nrow(x)	number of rows in data frame or matrix x
options(scipen=999)	forbid printing in scientific notation
plot(…)	draw a plot of the data
print(…)	print whatever is in the brackets to the output screen or file
rbind(…)	add a new row to a data frame
read.table("file.txt")	read in data file from file named in quotes
rm(…)	remove whatever is in the brackets from the working environment
round(x,v)	Round the elements of x off to v decimal places
sd(x)	standard deviation of a variable x
sink(…)	write to a file rather than to the screen
sqrt(x)	square root of the scalar, variable or list x
sqrt(x)	take the square root of x
str(x)	list the elements in the object x
summary(x)	summary statistics of variable or data frame x
write.csv(…)	save data frame as a CSV spreadsheet