Statistical Analysis Using XLispStat, R and Gretl: A Beginning

John E. Floyd
University of Toronto
September 16, 2011

This document presents some details on how to perform OLS-regression analysis and do other statistical work using three freely available computer programs. It is important in learning statistics and econometrics to program things for yourself as a first step—this helps you grasp the details as to why a statistical technique should be used and the details regarding the nature of that technique. The best program for doing this is `XLispStat`, a very sophisticated program developed by Luke Tierney, then of the University of Minnesota, in which it is possible to do programming operations quite easily. Another free program commonly used by really sophisticated statisticians and econometricians is `R`, which is a clone of the commercial statistical program `S`. This program is much more difficult to program things in than XLispStat in that it is very difficult to figure out how to use, but it contains many sophisticated functions programmed by a variety of statisticians and econometricians that one can eventually figure out how to access. Finally, the easiest program to use is `Gretl`, which is in my view the best program for day-to-day work, although you may sometimes have to use `R` to accomplish more sophisticated tasks that cannot be performed in `Gretl`, and sometimes have to write functions in `XLispStat` to figure out how things really work.

My own experience has taught me that I need to use two programs to perform many tasks. This is because it is very easy to make typo's and other procedural mistakes—putting the wrong variable in a regression by accidentally entering the incorrect name, deflating a variable second time having forgotten one's earlier action, and so forth—in any one program. But it is very unlikely that one will make the same mechanical mistakes a second time in another program. So it is always worthwhile to perform a few operations again in a second program to make sure that one gets the same results.

Above all, working through this material should help you improve your understanding of how to do statistical and econometric analysis. The programming in `XLispStat` starts on he next page. Go to page 45 for statistical analysis using `R` and the directions on how to use `Gretl` begin on page 62.

1. XLispStat

Although the `XLispStat` is no longer being developed, a satisfactory version for our purposes is easy to obtain. Simply download the self extracting zip-file `http://www.economics.utoronto.ca/jfloyd/stats/wxls32zp.exe`, and place it in a directory you create for it called `xlispstat` in the `Program Files` directory on your MS-Windows computer. Then click on `wxls32zp.exe` and all the program files will be extracted into that directory. Finally, right-click on the `wxls32.exe` icon in the directory and drag it to your desktop to create a desk-top icon.

Everything you need to know to use `XLispStat` will be developed as we move along. To learn more you can obtain a manuscript I wrote called *Statistics and Econometrics Using XlispStat*, University of Toronto, 2007, by downloading `http://www.economics.utoronto.ca/jfloyd/stats/stemlisp.pdf`. And to really get to the bottom of this beautifully sophisticated piece of software you would need to obtain Luke Tierney's book.[1]

`XLispStat` is basically a `list processing` program. A `list` is a collection of numbers or of words surrounded by quotation marks, or a collection of lists with each list denoted by a word giving its name not surrounded by quotation marks. These lists can be converted to vectors or collected together in matrices. Variables are denoted by words not surrounded by quotation marks.

We load the program by clicking on the `XLispStat` icon on our desktop and thereby obtain a `Listener Window` containing the following

```
XLISP-PLUS version 3.03
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT Release 3.52.8 (Beta).
Copyright (c) 1989-1998, by Luke Tierney.

>
```

[1] Luke Tierney, *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley Series in Probability and Mathematical Statistics, John Wiley & Sons, 1990.

We then type in commands at the > prompt. Or alternatively, we can put the code we want XLispStat to execute in a text file having the suffix .lsp and then load that file by clicking on File and then Load and then accessing the file from the directory in which it resides.

Our first task is to load the data to be analyzed. The normal place to put newly obtained data is in an Excel file which will usually include data descriptions generated by the source from which the data are downloaded. Then we copy the relevant series to a separate .xls file, taking nothing above the line giving the series names and including the left-most column giving the dates or observation names. In the case at hand, this file is named statcomp.xls. Since the data here are quarterly time series, we change the dates column by naming the first-quarter of 1959 in cell A2 as the number 1959.00 and then change the rest of the column by entering the code A2+(1/4) in the next cell below and copying the code in that cell to the rest of the cells in the column. This procedure sets the date in each quarter equal to the fraction of the year that has passed when the quarter begins. In the case of monthly data we add 1/12 to the previous cell and in the case of annual data we use simply increase by one the four-digit number representing the previous year. Finally, we place the character ; at the beginning of the word used to denote the date-column and thereby to the first row of the spreadsheet—it turns out that XLispStat will not read any line starting with that character. We will want R to read that line of variable names, but not XLispStat. The Gretl program, which can read the Excel file directly, will ask you to tell it the beginning date, and the R program requires additional code to attach dates to the data series—the datelist column will become superfluous in these cases and can then be deleted. For reading by XLispStat and R we save the material in our .xls file from Gnumeric as a configurable text file with spaces separating the columns. In the present case, this file is named statcomp.tab.

To load the data into XLispStat we use the code in the file setqdata.lsp. You should keep a copy of this file because it can be easily modified to load in a different data set. The code in the file is as follows:

```
; XLISPSTAT BATCH FILE FOR SETTING UP QUARTERLY LISP DATA SET
;
(def datalists (read-data-columns "statcomp.tab" 7))
(def datesq59 (select datalists 0))
(def usgdpsa (select datalists 1))
(def us3mtbr (select datalists 2))
(def usipdsa (select datalists 3))
(def uscpisa (select datalists 4))
(def usm1sa (select datalists 5))
(def usm2sa (select datalists 6))
;
(def NOTES (QUOTE "All series but the three-month treasury bill rate
(us3mtbr) are seasonally adjusted.  The CPI and the implicit GDP
deflator (usipdsa) are indexes with base 2005 = 100.  All series
were obtained from the St. Louis Federal Reserve Bank data base
FRED and run from 1959Q1 to 2010Q1."))
;
(savevar '(datesq59 usgdpsa us3mtbr usipdsa uscpisa usm1sa usm2sa
NOTES) "stcmpdata")
;
(exit)
```

All functions in `XLispStat` are applied by enclosing in brackets `( )` the function name and then a series of words denoting the arguments the function needs to do its job. The first function used above is the function `def` which defines the list of lists named `datalists`, denoted as its first argument, as one generated by its second argument, the function `read-data-columns`, which takes as its first argument the name of the file containing our data, `"statcomp.tab"`, and as its second argument the number of columns of data in that file. The names of files to to be read in or saved are always surrounded by quotation marks. The next seven lines of code use `def` to give names to the individual lists in `datalists` which are selected one-by-one using the `select` function which takes `datalists` as its first argument and, as its second argument, the number of the list in `datalists` to be assigned the chosen name. After the columns are extracted and named, the function `def` is used again to define an object (actually, a list with one element) called `NOTES` which is a paragraph of words that is surrounded by quotation marks processed by the function `QUOTE` which tells `XLispStat` to simply quote what

follows and not treat it as code to be executed. We create the `NOTES` object to contain readily accessible information about details regarding the data that one might forget. The next function used is the `savevar` function which takes as its first argument the list of objects to be saved—this list is preceded by a `'` character which again tells `XLispStat` to simply quote what follows and not to interpret it as a function. All the variables including the `NOTES` object are saved. The second, and last, argument required by the `savevar` function is a word in quotation marks denoting the name of the file in which the data is to be saved. The program automatically adds the suffix `.lsp` to this name. The final function is the `exit` function which takes no arguments and simply ends the current `XLispStat` session.

When when writing `Lisp` code, it is useful to use a text editor like the one I use, called the `Crimson Editor`, which automatically highlights the bracket combinations that surround the functions we use—this helps us make sure we use the correct inclusion of brackets when we embed functions in other functions.

To subsequently work with the data above, all we need to do is include in our code the line `(load "stcmpdata.lsp")` and the variables in the file will be loaded into the workspace. To recall what these variables are, we simply use the function `variables`, which takes no arguments, by entering the code

```
(variables)
```

and the `XLispStat Listener`, which might be better referred to as the `Interpreter`, will produce a list of the variables in the workspace. In response to these actions, the `Listener` will produce the following output.

```
XLISP-PLUS version 3.03
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT Release 3.52.8 (Beta).
Copyright (c) 1989-1998, by Luke Tierney.

>
; loading E:\DSLMHTML\STATCOMP\stcmpdata.lsp

(variables)
(DATESQ59 NOTES US3MTBR USCPISA USGDPSA USIPDSA USM1SA USM2SA)
>
```

The objects and variables can be printed to the screen by simply typing their names at the prompt. It is useful to print out the `NOTES` object because it reminds us of the characteristics of the data.

```
> NOTES
"All series but the three-month treasury bill rate
(us3mtbr) are seasonally adjusted.  The CPI and the implicit gdp
deflator (usipdsa) are indexes with base 2005 = 100.  All series
were obtained from the St. Louis Federal Reserve Bank data base
FRED and run from 1959Q1 to 2010Q1."
>
```

Of course, if you choose, you need not bother making a notes object and simply use as your reference a printout of descriptions in the spreadsheet file in which you originally set up your data, or from descriptors you write in the data file produced by the `Gretl` program, into which you will likely also want to load these data.

We now need to be able to look at our data without printing out entire lists representing individual variables. Suppose that we want to print the first five elements of a list on the screen. We could enter code using the `select` function, feeding it the series name and then a list of numbers as its second argument. For example, to print out the first five quarters' observations of the variable `US3MTBR`, we would enter the command

```
> (select us3mtbr (list 0 1 2 3 4))
(2.7733333333 3 3.54 4.23 3.8733333333)
```

where the second line represents the answer given by the `Listener`. For frequent use, this process can be made easier by simply creating the following function.

```
(defun first-five (x)
"Args: (x)
Prints the first five elements of list x on screen."
(select x (list 0 1 2 3 4))
) ; end of function
```

This involves using the `defun` function and feeding it four arguments. First we tell it the name we want to give to the function, `first-five`. Then we give it a list of its arguments (one in this case) surrounded by `( )` brackets. Then we write a description of the function enclosed in quotation marks. Finally we insert our line of code. The last line gives the right bracket of the function to match the initial bracket placed before the word `defun`. Writing a comment line `; end of function` after that bracket reminds us to make sure the closing bracket is in place. Recall that the `Listener` does not bother to read any material to the right of the character `;`. To use this function, we simply enter a command like

```
(first-five US3MTBR)
```

where the argument passed to the function is the series that we want to observe the first five elements of.

Using the same type of commands, we can create two additional functions called `last-five` and `chosen-five` to read the last five elements of a list or any five elements starting with some chosen initial element. Consider these two functions in turn.

```
(defun last-five (x)
"Args: (x)
Prints the last five elements of list x on screen."
(select x (list (- (length x) 5)(- (length x) 4)(- (length x) 3)
(- (length x) 2)(- (length x) 1)))
) ; end of function
```

Here we use the fact that the last observation of the list equals the length of the list, which we obtain using the `length` function (which takes as its only argument the list in question), minus one observation. The observation number assigned by `XLispStat` to the last observation is one less than the length of the series because the first element of all lists is called element zero, so that the last element of, say, a 10-element list is be element number 9. We then embed the length function in five separate applications of the subtract `(- )` function which takes as the first of its two arguments the number we want to subtract from and as its second argument the number to subtract from that first number.[2] And these five subtract functions are then embedded in a `list` function, taking the place of the numbers `1 2 3 4 5` that were used in our `first-five` function.

```
(defun chosen-five (x y)
"Args: (x y)
Prints the five elements of list x on screen starting with element y."
(if (> y (- (length x) 5))(error "Less than five elements remaining")
(select x (list y (+ y 1)(+ y 2)(+ y 3)(+ y 4)))
) ; end of if
) ; end of function
```

Our chosen-five function, unlike the other two, takes two arguments and uses the `if` and `error` functions because we want to send ourselves an error message if we mistakenly ask for the five elements of list `x` starting at element `y` when that latter element is within the last five elements of the list. The `error` function takes a single argument consisting of a statement in quotation marks while the `if` function takes three arguments. Its first argument is an inequality statement stating that `y` is greater than the the length of the list minus five elements. If that inequality holds, the `error` function is instructed to print the error message to the screen. If the inequality does not hold, the the `select` function, which is the third argument to the `if` function, is used and the `Listener` prints out the element `y` and the four elements that follow. In this case, the `list` function embedded in the `select` function has embedded in it four applications of the add `(+ )` function which sequentially adds the numbers `1`, `2`, `3` and `4` to the number selected as the first of the

[2]Actually, additional numerical arguments can be included in the subtract function. The function then produces a number obtained by sequentially subtracting each number from what remains after the number to the left of it was subtracted.

chosen five.[3] In interpreting the results, we must keep in mind that counting begins with `0` so asking for the five observations starting with the thirtieth observation will give us the numbers for observations twenty-nine through thirty-four. In this respect, it might be useful to incorporate the zeroth observation into our thinking.

Another task we need to be able to perform in looking at time-series data is to access the value of particular series at specific dates of interest. To do this we can construct a date-to-observation `date2obs` function that will tell us the observation associated with any chosen date in a date list. Reasonable coding for this function is the following, where `x` is the date list and `y` is a particular date.

```
(defun date2obs (x y)
"Args: (x y)
Finds the observation number (first observation zero) associated
with a given date y in the specified date list x."
(if (< y (select x 0))
(error "Date passed to date2obs function is not in the date list")
) ; end of if
(if (> y (select x (- (length x) 1)))
(error "Date passed to date2obs function is not in the date list")
) ; end of if
(def obsnum 0)
(dotimes (i (length x)) (if (< (select x i)(- y .001))
(def obsnum (+ obsnum 1))
) ; end of if
) ; end of dotimes
obsnum
) ; end of function
```

The function begins with two applications of the `if` and `error` functions to tell us if we are selecting a date that is not in the date list. Note that, unlike the previous case, only two arguments are passed to the `if` function. When the third argument is missing, the function does nothing if the first argument passed to the function—that is, the equality or inequality statement—is false. Then comes the main part of the function which initially sets an

[3] Additional numbers can also be included in the plus function, in which case it returns the sum of all its arguments.

integer `obsnum` representing the observation number we are looking for as equal to zero. Then we use the `dotimes` function to go through the date list from observation zero to the final observation—that is, to let `i` alternatively take values from zero to the length of the date list. At each value of `i` from `0` to `length x` we increase `obsnum` by one unit if the date-element of the date list `x` is less than `y` by an amount exceeding a rounding-error of .001 in the specification of the date in the date list. Note again that when the inequality fails to hold, the `if` function gives no command. Once the date-elements become greater than `(- y .001)` the magnitude of `obsnum` is no longer increased. Then the function prints out the magnitude of `obsnum` and leaves this number in the work space for us to use later. Notice that the `dotimes` function takes two arguments: first, the number-of-times specification `(i (length x))` which says that `i` is to take values sequentially from `0` to `(length x)`, and second, the `if` command that specifies on each element of the date list from `0` onward whether the number `obsnum` should be increased by one unit. We can give the resulting value of `obsnum` left in memory a different name using the `def` function as shown below.

To find the level of `USCPISA` in the last quarter of 2004, we enter the first line of code below

```
(def num1 (date2obs DATESQ59 2004.75))
NUM1
> num1
183
> obsnum
183
>
```

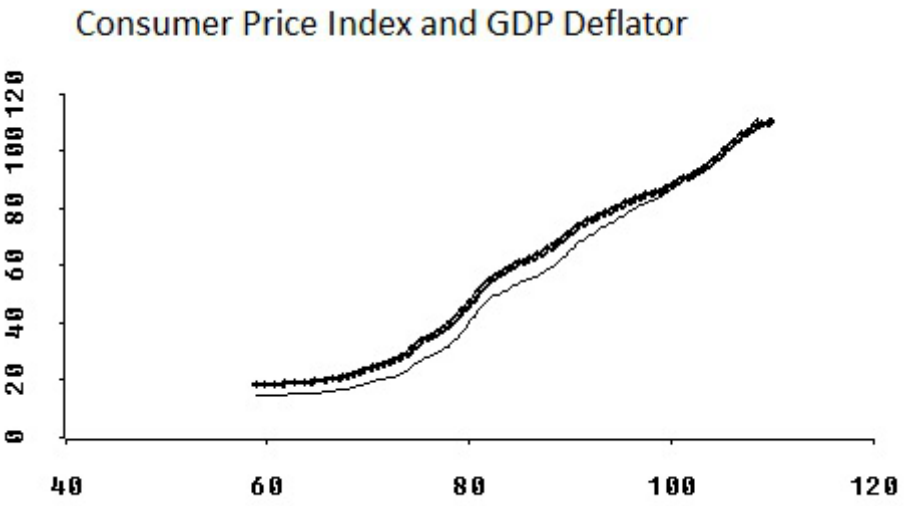
and then type the object names `num1` and `obsnum` to show that the latter object is in the workspace. Now let's have a look at the values of `USCPISA` for the five quarters starting at the last quarter of 2004.

```
(chosen-five USCPISA num1)
(98.0181287448149 98.5131697990987 99.17891466518149 100.681108209146
101.619979174118)
>
```

You can see that the last four of the above observations—those of the year 2005—are in the neighborhood of 100. Indeed, since the base-period of the series is 2005, the average of these four numbers should be 100.

It is a matter of interest whether the two measures of the price level, `USIPDSA` and `USCPISA` are equivalent measurements. One way to investigate this is to look at a plot of the two series. This is done using the `plot-lines` function along with the `def` function as follows:

```
(def plot1 (plot-lines (- DATESQ59 1900) USCPISA
:title "Consumer Price Index and Implicit GDP Deflator"))
(send plot1 :add-lines (- DATESQ59 1900) USIPDSA)
(send plot1 :add-points (- DATESQ59 1900) USIPDSA)
(send plot1 :adjust-to-data)
```



The `plot-lines` function takes two arguments, the variable on the horizontal axis, `(- DATESQ59 1900)` which is the date list, scaled so that `XLispStat` will not print it specified in scientific notation, and the variable `USCPISA` on the vertical axis, and it gives us an opportunity to send the resulting plot-object a message telling it to add a title. The `def` function is to give the object produced by the `plot-lines` function the name `plot1`. This object is a special object that contains a lot of information about the plot. We can access that information and give the plot-object instructions by sending it messages. We send three such messages above—asking it to add-lines and telling it the variables for the horizontal and vertical axes, `(- DATESQ59 1900)` and `USIPDSA` then asking it to add points (which will lie on top of the lines) for the latter variable and finally to `:adjust-to-data`, which instructs

it to adjust the vertical axis to ensure that no series ranges outside the plot. We put points on top of one of the two series plotted as lines to enable us to distinguish between those series when looking at the graph. Keep in mind also that we have to put an adjusted version of the date list on the horizontal axis, using the `subtract` function, to ensure that the scale on the horizontal axis will not be expressed in scientific notation. Better plots for incorporation into documents for publication can be produced using `Gretl`, and even better ones can be produced with `R`. If you want to print out an `XLispStat` plot for subsequent viewing, simply draw a square around it by holding down both mouse buttons and dragging the cursor, then copy it by clicking on `Edit` and then `Copy` and then paste it into a `WordPad` document for subsequent printing. Alternatively, it can be pasted into the Windows `Paint` program and resized and properly titled—which is what I did here. It turns out that the title which appears in the plot in `XLispStat` is on the frame of the window in which the plot appears and therefore cannot be copied in the above fashion.

The two price level series move closely together on the plot with the CPI series increasing a bit faster than the GDP deflator. To more closely examine the relationship between them it is useful to obtain the year-over-year percentage changes of the two series. To do this, we copy each series under a new name, removing the first four observations in the process, and then obtain one-year-lagged versions of both series by again copying the original series to additional new names while removing the last four observations. The two new versions of each series will have the same number of elements with each element in one version having as its counter-part in the other version the four-quarter-earlier value of that same series. It is useful to define the new current-period versions of two series as `CPIL0` and `IPDL0` and the lagged versions as `CPIL4` and `IPDL4`. We then make a new date list called `DATESQ60` by removing the first-four observations of `DATESQ59`.

To perform these tasks, it is useful to create two new functions which we call `remove-first` and `remove-last` that take as their two arguments the number of observations to remove and the series from which to remove them. The first of these functions uses the function `copy-list` to copy the original series list `y` to a new name `temp`. Then it applies to that new list the function `rest` which retains all but the first element of a series, doing so `x` times using the `dotimes` function. Our function leaves the new series `temp` in the workspace, but it is necessary to use the `def` function to give that series a unique name when running the `remove-first` function.

```
(defun remove-first (x y)
"Args: (x y)
Removes the first x observations from series y."
(def temp (copy-list y))
(dotimes (i x)(def temp (rest temp)))
temp
) ; end of function
```

The `remove-last` function also copies the original series to the new name `temp`. It then utilizes the function `remove` which takes as its second argument a list of numbers and as its first argument the number in that list to remove. The list of numbers is obtained by using the `iseq` function which creates a list of integers starting with 0 and ending with the number given as its only argument.

```
(defun remove-last (x y)
"Args: (x y)
Removes the last x observations from series y."
(def temp (copy-list y))
(dotimes (i x)
(def temp (select temp (remove (- (length temp) 1)
(iseq (length temp)))))
) ; end of dotimes
temp
) ; end of function
```

The main line of the function above uses the function `select` to grab all elements of `temp` but the last, which it does by taking as its second argument the sequence of element numbers of `temp` with the last element removed. This sequence of element-numbers
`(remove (- (length temp) 1)(iseq (length temp)))`
is created using the `iseq` function which generates a sequence of integers running from `0` to the length of `temp` minus one (keep in mind that the first element is element-number `0`). Removal of the last element is thus removal of element-number `(- (length temp) 1)`. This process is repeated `x` times using the `dotimes` function, leaving a final series `temp` equivalent to the initial series minus the last `x` elements.

We can now use these functions to create the year-over-year percentage growth rates of the CPI and GDP deflators. First, we lag the CPI and IPD variables and make a new date list

```
> (def CPIL0 (remove-first 4 USCPISA))
CPIL0
> (def CPIL4 (remove-last 4 USCPISA))
CPIL4
> (def IPDL0 (remove-first 4 USIPDSA))
IPDL0
> (def IPDL4 (remove-last 4 USIPDSA))
IPDL4
> (def DATESQ60 (remove-first 4 DATESQ59))
DATESQ60
>
```

Then we calculate the year-over-year difference in the series by subtracting the lagged series from the unlagged ones.

```
> (def YYDCPI (- CPIL0 CPIL4))
> YYDCPI
> (def YYDIPD (- IPDL0 IPDL4))
> YYDIPD
>
```

Next we divide the year-over-year differences by their respected lagged values to get the year-over-year rate of growth—here we use the `divide` function which is represented by the backslash character `(/ )`.

```
> (def YYGCPI (/ YYDCPI CPIL4))
> YYGCPI
> (def YYGIPD (/ YYDIPD IPDL4))
> YYGIPD
>
```

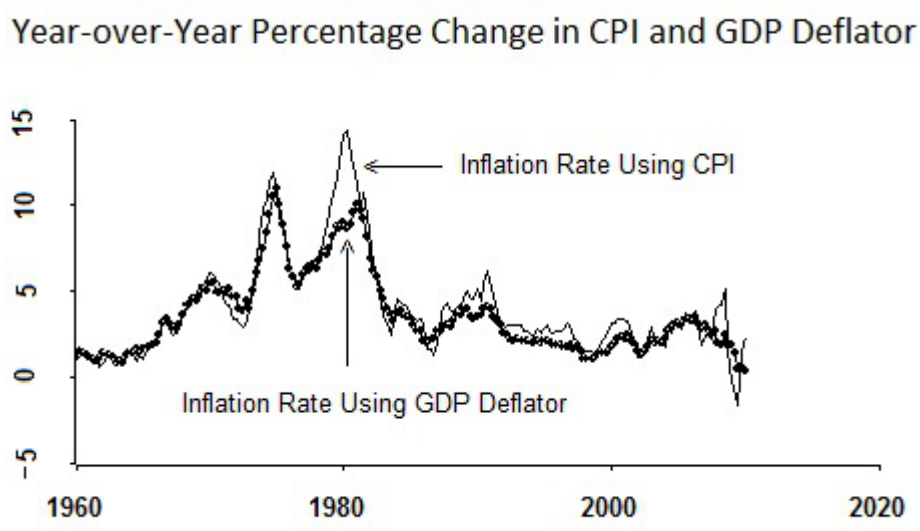
Finally, we use the `multiply` function `(* )` to multiply the above two series by 100 to put the growth rates in percentage terms.

```
> (def YYGCPI (* 100 YYGCPI))
YYGCPI
> (def YYGIPD (* 100 YYGIPD))
YYGIPD
>
```

Notice that the functions `+ - * /` can be used to add, subtract, multiply or divide the elements of the first list given as an argument by the corresponding elements of the second list given, implying that the two lists must have the same length. And these functions can also be used to add, subtract, multiply or divide all elements of the list given as the first argument by a single number given as the second argument. In the cases of addition and multiplication it does not matter whether the single number or the list is the first argument—in the cases of subtraction, the second argument is subtracted from the first and in the case of division the elements or element of the first argument are divided by the element or elements of the second argument. An additional function not shown is the `(^ )` function which takes the first argument, which can be either a number or a list, to the power given by the second argument which must be a number.

Now we can plot our two percentage growth rate series using the similar code to that used in the first plot—we simply change the plot name and the variables involved.

```
> (def plot2 (plot-lines (- DATESQ60 1900) YYGCPI
:title "Year-Over-Year Percentage Growth of CPI and IPD"))
> (send plot2 :add-lines (- DATESQ60 1900) YYGIPD)
> (send plot2 :add-points (- DATESQ60 1900) YYGIPD)
> (send plot2 :adjust-to-data)
```

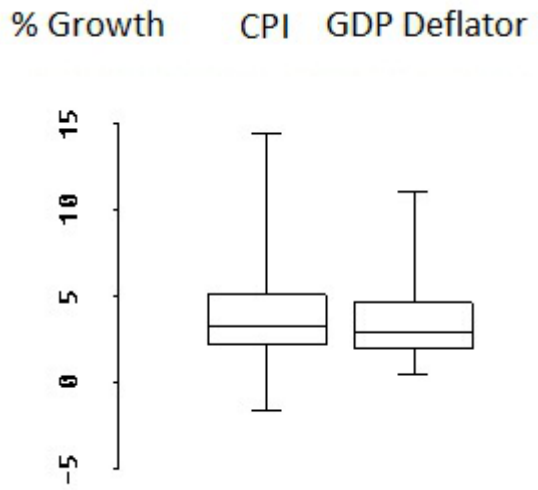


I modified this plot after copying it into the Windows Paint program, cleaning up the presentation of the dates and adding labels for the lines as well as adding a title.

It turns out that the two growth rates vary together within the range of 0 to 15 percent, showing little overall trend, with the CPI series being more variable than the implicit GDP deflator series. This is also evident from a boxplot of the two series which we can produce using the code

```
> (def plot3 (boxplot (list YYGCPI YYGIPD)
:title "CPI (Left) and IPD Growth"))
```

which produces the plot on the next page. Notice how we send the `boxplot` function a list containing the two growth series using the `list` function. And note also that we can put brackets around any word in a group surrounded by quotation marks without thereby defining a function. I added the labels across the top of the plot using the Windows Paint program.



A more thorough analysis requires that we examine the characteristics of the two series in detail using the functions `mean`, `median`, `standard deviation`, `max` and `min`, which each take a series list as their only argument, and the `quantile` function which takes as its two arguments the list in question and the quantile to be calculated (which ranges between 0 and 1). The best way to do this is to make our own function, called `stats`, using the following code.

```
(defun stats (x n)
"Args: (x n)
Calculates and prints to screen summary statistics for the list x
named n."
(terpri)
(princ n)(terpri)
(terpri)
(princ "Mean                = ")(princ (mean x))(terpri)
(princ "Standard-deviation = ")(princ (standard-deviation x))(terpri)
(terpri)
(princ "Minimum             = ")(princ (min x))(terpri)
(princ "First Quartile      = ")(princ (quantile x .25))(terpri)
(princ "Median              = ")(princ (median x))(terpri)
(princ "Third Quartile      = ")(princ (quantile x .75))(terpri)
(princ "Maximum             = ")(princ (max x))(terpri)
(terpri)
) ; end of function
```

Here we also use the `princ` and the `terpri` functions. The `princ` function prints to the screen whatever is given to it as a single argument, and the `terpri` function produces, in effect, a hard-return by moving to the next line.

In order to calculate the correlation between the two series, we need to make a `correlation` function which requires as a prelude a `covariance` function. These use coding that has been previously used and explained.

```
(defun covariance (x y)
"Args: (x y)
Computes the covariance of the elements in two lists."
(/ (sum(* (- x (mean x))(- y (mean y))))
(- (length x) 1))
) ; end of function
```

The `sum` function sums the elements in the list obtained by multiplying the deviations of `x` and `y` from their means. This sum is then divided by the number of elements minus one in whichever of the two lists we choose, both lists necessarily being of the same length. Our `correlation` function has to divide the covariance of the two lists by the product of their standard-deviations.

```
(defun correlation (x y)
"Args: (x y)
Computes the coefficient of correlation between the elements
of two lists."
(/ ( covariance x y )(* (standard-deviation x)
(standard-deviation y)))
) ; end of function
```

Now let us use these new functions.

```
> (stats YYGCPI "CPI Growth")

CPI Growth
Mean               = 4.104037396541741
Standard-deviation = 2.9098164819346595

Minimum            = -1.595183386704363
First Quartile     = 2.253521126760548
Median             = 3.2997568599502767
Third Quartile     = 5.16019279835407
Maximum            = 14.42577030812341

> (stats YYGIPD "IPD Growth")

IPD Growth
Mean               = 3.6539651250012435
Standard-deviation = 2.366958780965972

Minimum            = 0.4868220729139052
First Quartile     = 1.99169753051885
Median             = 2.992597758721814
Third Quartile     = 4.714921148402751
Maximum            = 11.085089773614358

> (princ "Correlation Coefficient of YYGCPI and YYGIPD = ")
(princ (correlation YYGCPI YYGIPD))(terpri)

Correlation Coefficient of YYGCPI and YYGIPD  = 0.9425269296362888
```

We are now ready to put together a function what will compute OLS regressions. The basic regression model can be described in matrix terms as

$$\begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} 1 & X_{11} & X_{21} & X_{31} & \cdots & \cdots & X_{K1} \\ 1 & X_{12} & X_{22} & X_{32} & \cdots & \cdots & X_{K2} \\ 1 & X_{13} & X_{23} & X_{33} & \cdots & \cdots & X_{K3} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & X_{1n} & X_{2n} & X_{3n} & \cdots & \cdots & X_{Kn} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \vdots \\ \beta_K \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \vdots \\ \vdots \\ \vdots \\ \epsilon_n \end{bmatrix}$$

which can be written more simply as

$$\mathbf{Y} \quad = \quad \mathbf{X}\mathcal{B} + \mathcal{E} \tag{1}$$

where $\mathbf{Y}$ is an n by 1 column vector, $\mathbf{X}$ is an n by $K+1$ matrix (i.e., a matrix with n rows and $K+1$ columns), $\mathcal{B}$ is a $K+1$ by 1 column vector and $\mathcal{E}$ is an n by 1 column vector. The first column of the matrix $\mathbf{X}$ is a column of elements all equal to unity, representing the constant term in the regression.

Our problem is now make an estimate (1) of the form

$$\mathbf{Y} \quad = \quad \mathbf{X}\mathbf{b} + \mathbf{e}\,, \tag{2}$$

choosing the $\mathbf{b}$ vector that will minimize the sum of the squared regression residuals

$$\mathbf{e}'\mathbf{e} = \sum_{i=1}^{n} e_i^2.$$

It turns out that the formula for $\mathbf{b}$ is [4]

$$\hat{\mathbf{b}} \quad = \quad (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{Y}\,. \tag{3}$$

To do this in `XLispStat` we need to first construct the $\mathbf{Y}$ vector and the $\mathbf{X}$ matrix.

[4]The theoretical basis for this formula is worked out in Chapter 9 of my document *Statistics for Economists: A Beginning*, pages 235-236.

We proceed to estimate the demand function for U.S. M1. To begin, we have to get our data in the proper form. First, we must obtain real GDP and real M1 series from the nominal series `USGDPSA` and `USM1SA` by dividing both of them by the implicit GDP deflator `USIPDSA` (which from the graphs above seemed more stable than the CPI series) and then multiplying the results by 100. Then we take the natural logarithms of both series.

```
> (def USRGDP (* 100 (/ USGDPSA USIPDSA)))
> (def USRM1 (* 100 (/ USM1SA USIPDSA)))
> (def LUSRGDP (log USRGDP))
> (def LUSRM1 (log USRM1))
```

Next we construct a constant term using the `repeat` function which forms a list consisting of the number of repetitions, given by its second argument, of the number given by its first argument. The list of ones must be as long as the list representing dependent variable.

```
> (def CONST (repeat 1 (length USRM1)))
```

We can now construct the $\mathbf{Y}$ vector and the $\mathbf{X}$ matrix using the `bind-columns` function, which binds together a specified sequence of lists into a matrix. This matrix is a column vector if only one list is fed to the function.

```
> (def Yvec (bind-columns LUSRM1))
(def Xmat (bind-columns CONST US3MTBR LUSRGDP))
```

To calculate the $\mathbf{b}$ vector using equation (3) we enter the code, using the matrix functions `matmult`, `transpose` and `inverse`,

```
> (def XPX (matmult (transpose Xmat) Xmat))  ; X prime times X
> (def XPXINV (inverse XPX))
> (def XPY (matmult (transpose Xmat) Yvec))  ; X prime times Y
> (def coefs (matmult XPXINV XPY))           ; coefficient vector
```

The fitted values,

$$\hat{\mathbf{Y}} \;=\; \mathbf{b}\,\mathbf{X} \tag{4}$$

are then calculated using the code

```
> (def fitted (matmult Xmat coefs))
```

and the regression residuals

$$\hat{\mathbf{e}} \;=\; \mathbf{Y} - \hat{\mathbf{Y}}\,, \tag{5}$$

are calculated using the rather obvious code line

```
> (def resids (- Yvec fitted))
```

The degrees of freedom, which is equal to the number of observations minus the number of parameters estimated, can be extracted from `Xmat` using the `array-dimensions` function, which takes as its sole argument the matrix being examined and produces a list containing two numbers—first, (number `0`) the number of rows and second, (number `1`) the number of columns. Subtraction of the number of columns from the number of rows gives us the degrees of freedom.

```
> (def df (- (select (array-dimensions Xmat) 0)
(select (array-dimensions Xmat) 1)))
```

The sum of squared residuals $\mathbf{e}'\mathbf{e}$ can then be obtained along with the mean squared error s^2.

$$s^2 = \frac{\mathbf{e}'\mathbf{e}}{df}$$

```
> (def SSE (matmult (transpose resids) resids))
> (def MSE (/ SSE df))
```

Next, we need the total sum of squares and the sum of squares explained by the regression, which requires that we first calculate the deviations of $\mathbf{Y}$ from its mean.

```
> (def devYvec (- Yvec (mean Yvec)))
> (def SST (matmult (transpose devYvec) devYvec))
> (def SSR (- SST SSE))
```

Calculation of the R^2 is now a simple matter and, after obtaining the number of observations, the degrees-of-freedom adjusted R^2, called $\bar{R}^2$ can also be calculated.[5]

[5]Using equation (9.7) in *Statistics for Economists: A Beginning* on page 228.

```
> (def RSQ (aref (/ SSR SST) 0 0))
> (def nobs (select (array-dimensions Xmat) 0))
> (def RBSQ (aref (- 1 (/ (* (- nobs 1) SSE) (* df SST))) 0 0 ))
```

where, you must note, we use the function `aref` to select the element associated with a particular row and column number of a matrix. The matrices `SSR` and `SST` both have a single element—that is, one row and one column.

The variance-covariance matrix of the coefficients is

$$E\{(\hat{\mathbf{b}} - \mathcal{B})(\hat{\mathbf{b}} - \mathcal{B})'\} = \sigma^2(\mathbf{X}'\mathbf{X})^{-1}$$

and is estimated by replacing σ^2 on the right side of the equality by s^2, which equals `MSE`.

```
> (def vcvcoefs (* (aref MSE 0 0) XPXINV))
```

It turns out that `MSE` is a 1×1 matrix which cannot be multiplied by the matrix `XPXINV` because it is not of the same dimension. A matrix can be multiplied by an ordinary number, however, so we have to extract that number from the 1×1 matrix using the function `aref`. The vector of variances of the individual coefficients can then be extracted from the variance-covariance matrix of the coefficients using the `diagonal` function, which takes as its sole argument the matrix we want it to return the diagonal of, and the standard-errors of the coefficients are simply the square roots of the elements of this vector. And the vector of t-ratios is the coefficients vector divided by the vector containing their standard-errors.

```
> (def varcoefs (diagonal vcvcoefs))
> (def stdcoefs (sqrt varcoefs))
> (def trats (/ coefs stdcoefs))
```

To obtain the P-Values for the coefficients, we use the cumulative density function for the t-statistic, `t-cdf`, which takes as the first of its two arguments the value of the t-statistic and then as its second argument the number of degrees of freedom. To get a little area at either tail of the distribution, we have to subtract the cumulative density from unity.

```
> (def Pvals (- 1 (t-cdf (abs trats) df)))
```

Here the `abs` function takes the absolute values of the elements of the t-ratios vector—what matters is the absolute magnitude of each t-statistic independent of its sign. Finally, we need to calculate the F-statistic. The numerator is the sum of squares explained by the regression divided by the number of coefficients estimated and the denominator is the mean-squared-error.

```
> (def numF (/ SSR (select (array-dimensions Xmat) 1)))
> (def F-Stat (aref (/ numF MSE) 0 0 ))
```

All the above code is embedded in the run-regession function `runOLS` below. This function leaves many variables in the workspace without printing things out—this is because our next job will be to embed it in another function that will perform that task.

```
(defun runOLS (y x)
"Args: (x y)
Runs an OLS regression of the column vector y on the matrix x
and does not print out the results."
(def Yvec y)
(def Xmat x)
(def XPX (matmult (transpose Xmat) Xmat))
(def XPXINV (inverse XPX))
(def XPY (matmult (transpose Xmat) Yvec))
(def coefs (matmult XPXINV XPY))
(def fitted (matmult Xmat coefs))
(def resids (- Yvec fitted))
(def df (- (select (array-dimensions Xmat) 0)
(select (array-dimensions Xmat) 1)))
(def SSE (matmult (transpose resids) resids))
(def MSE (/ SSE df))
(def devYvec (- Yvec (mean Yvec)))
(def nobs (select (array-dimensions Xmat) 0))
(def SST (matmult (transpose devYvec) devYvec))
(def SSR (- SST SSE))
(def RSQ (aref (/ SSR SST) 0 0))
(def RBSQ (aref (- 1 (/ (* (- nobs 1) SSE) (* df SST))) 0 0 ))
(def vcvcoefs (* (aref MSE 0 0) XPXINV))
(def varcoefs (diagonal vcvcoefs))
```

```
(def stdcoefs (sqrt varcoefs))
(def trats (/ coefs stdcoefs))
(def Pvals (- 1 (t-cdf (abs trats) df)))
(def numF (/ SSR (select (array-dimensions Xmat) 1)))
(def F-Stat (aref (/ numF MSE) 0 0 ))
) ; end of function
```

We now construct the following function to print out the regression results and save the coefficients, the number of observations, the fitted values and the regression residuals under new names that will not be overwritten when the `runOLS` function is run again. Since we may sometimes want to not include a constant term in a regression, we specify three arguments in our new `OLSreg` function, the third being the integer `1` if a constant is to be included and the integer `0` if a constant is not to be included. It follows that the $\mathbf{X}$ matrix passed to the function must not contain a constant, thereby leaving it to `OLSreg` to create one if instructed to do so by the unitary third argument.

```
(defun OLSreg (y x z)
"Args: (y x)
Runs an OLS regression of the column vector y on the matrix x,
incorporating a constant term if z is equal to unity, and
prints the results."
(def conlist (repeat 1 (select (array-dimensions y) 0)))
(if (= z 1)(def x (bind-columns conlist x))
) ;end if
(runOLS y x)
(terpri)
(princ "ORDINARY LEAST SQUARES REGRESSION")(terpri)(terpri)
(princ "Dependent Variable: ")(princ regressand)(terpri)(terpri)
(format t "~20a ~12a ~12a ~12a ~12a" " "
"Coefficient" "Std. Error" "T-stat" "P-Val")
(terpri)(terpri)
(dotimes (i (- nobs df))
(format t "~17a ~12,3f ~11,3f ~10,3f ~11,3f" (select regressors i)
(aref coefs i 0)(select stdcoefs i)(aref trats i 0)
(aref Pvals i 0))
(terpri)
) ; end dotimes i
```

```
(terpri)(terpri)
(princ "Number of Observations: ")(princ NOBS)(terpri)
(princ "Degrees of Freedom:     ")(princ df)(terpri)
(princ "R-Squared:              ")(princ RSQ)(terpri)
(princ "Adjusted R-Squared:     ")(princ RBSQ)(terpri)
(princ "Sum of Squared Errors:  ")(princ (aref SSE 0 0))(terpri)
(princ "F-Statistic:            ")(princ F-Stat)(terpri)
(princ "   P-Value              ")(princ (- 1 (f-cdf F-Stat (- nobs df) df)))
(terpri)(terpri)
(def OLSresids (repeat 0 nobs))
(def OLSfitted (repeat 0 nobs))
(def OLScoefs (repeat 0 (- nobs df)))
(dotimes (i (- nobs df))
(setf (select OLScoefs i)(aref coefs i 0))
); end dotimes
(dotimes (i nobs)
(setf (select OLSresids i)(aref resids i 0))
(setf (select OLSfitted i)(aref fitted i 0))
)
(def RXmat Xmat)
(def OLSnobs nobs)
) ; end of function
```

Before we call this function, we must write two lists to reside in the workspace. The first, `regressand` is a text specification of the name of the dependent variable, and the second `regressors` is a list specifying the names of the independent variables.

```
> (def regressand "Log of Real M1")
> (def regressors (list "Constant" "T-Bill Rate" "Log of Real GDP"))
```

If so instructed by its third argument, the first thing the `OLSreg` function above does is to add a constant term as the new left-most column of the matrix fed to it as its second argument `x`. Then it calls the `runOLS` function, feeding it the y-vector `y` and that matrix. Then it uses the 'hard-return' `terpri` function and the `princ` function to print details about the regression. A new printing function called `format` is then used.

The `format` function takes a complex set of arguments. The initial argument is the letter `t` which specifies that the material is to be printed to the screen

rather than to a file. The next argument is a set of width-specified columns, surrounded by a single set of quotation marks. In the first use of the function, this code is

```
"~20a ~12a ~12a ~12a ~12a"
```

which specifies five columns, the first being 20 characters wide and the other four being 12 characters wide, with the trailing `a` character specifying that the entries in all columns should be text surrounded by quotation marks. This is followed by one additional argument for each column, representing text that is to be inserted. For the first column, nothing is to be inserted, so the specification is `" "`, which means print nothing. For the other four columns, column headings are specified indicating the particular statistics that will be printed below.

```
" " "Coefficient" "Std. Error" "T-stat" "P-Val"
```

Then, following a couple of 'hard-returns', a `dotimes` loop is constructed to print the rows of statistics that follow. The `format` function is used again in each loop.

```
(format t "~17a ~12,3f ~11,3f ~10,3f ~11,3f" (select regressors i)
(aref coefs i 0)(select stdcoefs i)(aref trats i 0)
(aref Pvals i 0))
```

After specifying that the printing is to be to the screen rather than to file, the entry for the first column is specified as text, like above, but in this case with a width of only 17 characters. The other columns are 12, 11, 10, and 11 characters in width, respectively, with characters `,3f` specifying that real numbers must be printed, with three-decimal places. Then the contents of the five columns for that row are specified as members of the `regressors` vector, the vector of coefficient estimates, the list of coefficient standard errors, the t-ratios vector and the P-Values vector. This line is repeated for the constant, if one is being included, and each of the remaining independent variables.

Then a series of lines is printed to write to screen the various statistics of interest. The new piece of code here is the `F-cdf` function which takes as its three arguments the F-statistic, the number of degrees of freedom in the numerator and the number of degrees of freedom in the denominator. The cumulative density is subtracted from unity to capture the appropriate area in the right tail of the distribution.

Finally, the code fragments

```
(def OLSresids (repeat 0 nobs))
(def OLSfitted (repeat 0 nobs))
(def OLScoefs (repeat 0 (- nobs df)))
(dotimes (i (- nobs df))
(setf (select OLScoefs i)(aref coefs i 0))
); end dotimes
(dotimes (i nobs)
(setf (select OLSresids i)(aref resids i 0))
(setf (select OLSfitted i)(aref fitted i 0))
)
(def RXmat Xmat)
(def OLSnobs nobs)
```

copy the `(nobs - df)` elements of the coefficients vector and the `nobs` elements of the `resids` and `fitted` vectors to lists entitled `OLScoefs`, `OLSresids` and `OLSfitted` and save the matrix `Xmat` under the new name `RXmat`. And the last line of code saves the `nobs` list to a new name, `OLSnobs`. These new names ensure that these objects will not be overwritten by calling the `runOLS` function again to conduct Breusch-Pagan heteroskedasticity tests and LM-based tests that will enable us to determine whether the residuals are serially correlated and by making the bootstrapped estimates of the regression coefficients discussed later below.

The XLispStat output below is produced by just five lines of code.

```
> (def regressand "Log of Real M1")
> (def regressors (list "Constant" "T-Bill Rate" "Log of Real GDP"))
> (OLSreg (bind-columns LUSRM1)(bind-columns US3MTBY LUSRGDP) 1)
> (BRPG)
> (LMSC 4)
```

where the Breusch-Pagan test function BRPG and the LM-test for serial correlation in the residuals LMSC remain to be discussed, a task to which we next turn.

```
ORDINARY LEAST SQUARES REGRESSION

Dependent Variable: Log of Real M1

                    Coefficient  Std. Error   T-stat       P-Val

Constant              3.348       0.093     35.983       0.000
T-Bill Rate          -0.019       0.002    -11.281       0.000
Log of Real GDP       0.419       0.010     40.479       0.000


Number of Observations:      205
Degrees of Freedom:          202
R-Squared:                   0.9064457850619718
Adjusted R-Squared:          0.9055195057061498
Sum of Squared Errors:       0.9473691950417262
F-Statistic:                 978.5879166631256
   P-Value                   0.0

Breusch-Pagan ChiSquare Statistic:  = 14.352166650006135
      P-Value                       = 7.646568973018741E-4

LM-Test for Serial Correlation of Residuals:
    Number of Lags      = 4
    Chisquare Statistic = 3249.5072912374058
    P-Value             = 0.0
```

The Breusch-Pagan test for heterskedasticity of the residuals involves regressing the squared residuals on some or all of the independent variables in the regression whose residuals are being tested. The number of observations in this Breusch-Pagan regression times its R^2 is distributed under the null-hypothesis of no heteroskedasticity as Chi-square with degrees of freedom equal to the number of independent variables included other than the constant term. The function developed here for doing this, which takes no arguments, consists of the following code, which includes in the Breusch-Pagan regression all the independent variables in the original regression including a constant term.

```
(defun BRPG ( )
"Args: ( )
Performs a Breusch-Pagan test on the residuals
from the previous regression whose results were
printed out."
(def residsq (^ OLSresids 2))
(runOLS (bind-columns residsq) Xmat)
(def bpchisq (* (length residsq) RSQ))
(def bppv (- 1 (chisq-cdf bpchisq (- (select (array-dimensions Xmat) 1) 1))))
(princ "Breusch-Pagan ChiSquare Statistic:  = ")(princ bpchisq)(terpri)
(princ "      P-Value                       = ")(princ bppv)(terpri)
) ; end of function
```

The first line of code sets up the variable `residsq`, which is the square of the residuals list `OLSresids` created in the last lines of code in the `OLSreg` function. Then the `runOLS` function is called, with the independent variables being the same as in the call to `OLSreg`. The task is then to multiply the resulting R^2 statistic by the number of observations (which equals the length of `residsq`) to produce a Chi-Square statistic with degrees of freedom equal to the number of regressors apart from the constant. The P-Value of that statistic is then calculated using the `chisq-cdf` function which takes as its two arguments the Chi-Square statistic and the degrees of freedom, which equal the number of columns of the matrix `Xmat` minus one. The results are then printed in the usual fashion.

The LM-based test for the presence of serial correlation in the residuals involves regressing those residuals on the independent variables in the original regression, including a constant, with one or more lags of those residuals

added. In the approach adopted here, an F-statistic measuring the contribution of those lagged residuals to the regression is then calculated and multiplied by the number of lagged residuals included to obtain a statistic that has a Chi-square distribution with degrees of freedom equal to the number of included lagged residuals. As a prelude to setting up our `LMSC` function, we need to make two new functions that our subsequently constructed `LMSC` function will need to use. First, we need to be able to create the number of lags of the residuals variable to be specified when running our `LMSC` function. To do this we write the following `makelags` function.

```
(defun makelags (x y)
"Args: (x y)
Constructs x lagged values of the list y starting with an adjusted
level of the list y called lag 0, leaving in the workspace a list
of lags called lagslist containing lags 0 through lag x and a matrix
called lagmat containing lags 1 through x."
(def nlags x)
(def lagnum (iseq 0 nlags))
(def nextlag (remove-first nlags y))
(def lagslist (list nextlag))
(dotimes (i nlags)
(def nextlag (remove-first (- nlags (+ i 1)) y))
(def nextlag (remove-last (+ i 1) nextlag))
(def nextlag (list nextlag))
(def lagslist (append lagslist nextlag))
) ; end of dotimes
(def lagmat (bind-columns (select lagslist 1)))
(dotimes (i (- nlags 1))
(def lagmat (bind-columns lagmat (select lagslist (+ i 2))))
) ; end of dotimes
) ; end of function
```

First we use the `iseq` function to make a list of lag numbers having a length equal to the number of lags. Then we make a list of lags called `lagslist`, the first element of which is the new current level of the series being lagged, obtained by removing a number of elements from the beginning equal to the number of lags, `x`. Then we construct a `dotimes` loop to create the remaining lagged series by removing elements from the beginning and the end of the original series—a lag of (`x - 1`) requires that (`x - 1`) elements be removed

from the beginning of the series and one element be removed from the end, and a lag of (x - 2) requires that (x - 2) elements be removed from the beginning of the original series and two elements be removed from the end, and so forth. After creating each lag of the original series, we add it to lagslist. When we are finished, that list contains (x + 1) series having lags from 0 to x respectively. Finally, we bind all lagged series with lags greater than zero into columns of a matrix called lagmat.

The next function we need is one with which to remove a selected number of rows from the begining of a matrix. This is necessary because in our function LMSC we will be using our original matrix, saved in that original regression under the name of RXmat, along with the matrix of lagged residuals in a regression that uses the residuals as the dependent variable. Since the lagged residuals series and the dependent variable are x elements shorter than RXmat we must remove the first x rows of that matrix. The function to do this is our remove-first-rows function below.

```
(defun remove-first-rows (n x)
"Args: (n x)
Removes the first n rows of the matrix x."
(def xdim (array-dimensions x))
(def oldnumr (select xdim 0))
(def newnumr (- oldnumr n))
(def oldnumc (select xdim 1))
(def newnumc oldnumc)
(def newdimlist (list newnumr newnumc))
(def newx (make-array newdimlist :initial-element 0))
(dotimes (i newnumr)
(dotimes (j newnumc)
(setf (aref newx i j)(aref x (+ i n) j))
); end dotimes j
); end dotimes i
newx
) ; end of function
```

In this function we obtain the number of rows and columns of the original matrix using the array-dimensions function and then, using the def function, set the number of rows of the new matrix equal to the original number minus the number of rows to be removed, specified in the first argument in

the function we are creating. We then set the number of columns in the new matrix equal to the number columns in the original one. Next, we make a new matrix `newx` using the `make array` function, which takes as its first argument a list we constructed called `newdimlist` in the previous line of code, giving the row and column dimensions of the new matrix, and as its second argument a code segment `:initial-element 0` where the character `0` specifies that all elements of the new matrix be set equal to zero (we could have used another number). Then we construct two `dotimes` loops, one embedded in the other, to change the elements of this new matrix to equal the appropriate elements of the old matrix—the `i`th row of the new matrix is the `(i + n)`th row of the original matrix.

We are now ready to set up our `LMSC` function to test for serial correlation in the residuals `OLSresids`. It takes as its sole argument the number of lags.

```
(defun LMSC (x)
"Args: (x)
Performs a LM-based test, using x lags, for serial correlation
in residuals of the previous regression whose results were
printed out."
(def numlags x)
(makelags numlags OLSresids)
(def LMXmat (remove-first-rows numlags RXmat))
(runOLS (bind-columns (select lagslist 0)) LMXmat)
(def SSER (aref SSE 0 0))
(runOLS (bind-columns (select lagslist 0))(bind-columns lagmat LMXmat))
(def SSEU (aref SSE 0 0))
(def numF (/ (- SSER SSEU) numlags))
(def denF (/ SSEU df))
(def LMF-stat (/ numF denF))
(def LMChisq (* numlags LMF-stat))
(def LMpv (- 1 (chisq-cdf LMchisq numlags)))
(terpri)
(princ "LM-Test for Serial Correlation of Residuals:")(terpri)
(princ "    Number of Lags      = ")(princ numlags)(terpri)
(princ "    Chisquare Statistic = ")(princ LMChisq)(terpri)
(princ "    P-Value             = ")(princ LMpv)(terpri)
(terpri)
) ; end of function
```

This function, which follows the specification in G.S. Maddala's textbook,[6] uses our `makelags` function to make the specified number of lags of `OLSresids` and our `remove-first-rows` function to appropriately adjust the `RXmat` matrix to remove a number of rows equal to the number of lags. Then two regressions with the previous regression residuals as the dependent variable are run using the `runOLS` function. The first is a restricted regression that excludes the lagged residuals and the sum of squared residuals are saved as `SEER`, the restricted sum of squares. The second regression is the unrestricted regression that includes the lagged residuals and here the resulting sum of squared residuals is saved as `SEEU`, the unrestricted sum of squares. The F-statistic to test whether the restriction leads to a significant increase in the sum of squared residuals is then constructed as the ratio of (`SSER - SSEU`) divided by the number of restrictions (which equals the number of lagged residuals) over the unrestricted sum of squares `SSEU` divided by the degrees of freedom of the unrestricted regression. This F-statistic is then multiplied by the number of restrictions, which equals the number of lags, to obtain a Chi-Square statistic with degrees of freedom equal to the number of restrictions, and the P-Value is calculated using the `chisq-cdf` function in the usual fashion. Finally these results are printed out using the `princ` and `terpri` functions.

To perform the Breusch-Pagan and LM test we write the following two commands after running the regression whose residuals we are testing.

```
> (BRPG)
> (LMSC 4)
```

where we specify that the serial correlation test is to test for up to four lags of the residuals. Normally, we would run only the `LMSC` test in the case of time-series regressions and only the `BRPG` test when the data are cross-sectional.

It is obvious that there is very substantial serial correlation in the regression residuals. Methods for dealing with this are discussed in detail in the Lesson entitled *Econometrics Basics: Dealing with Serial Correlation in Regression Residuals*. A rather simple and seemingly unsophisticated approach to handling this problem is to make `Bootstrapped` estimates of the regression coefficients. This procedure assumes that the residuals from our regression are a sample from a large population of error terms, and takes

[6] G.S. Maddala, *Introduction to Econometrics*, MacMillan, 1988, page 206.

repeated samples of those residuals, where each sample has the same number of elements as in the original residuals and is thus simply an inter-temporally reorganized collection of original residuals. It adds these resampled residuals to the fitted values of the original regression, rerunning the regression using each of these reconstructed values of the dependent variable. The samples of the residuals are taken with replacement, so the new residuals added to the fitted values are the same as the original ones except that the distribution of those residuals through time is different for each sample, with some of the original residuals possibly appearing more than once and some others therefore appearing not at all. New coefficients are obtained for each sample of the original residuals, and the quantiles of these new coefficients (that is, the magnitudes below which specific fractions of the total number of estimates of the coefficients lie) are calculated. This enables us to get a sense of the likely probability distributions of those individual coefficients—for example, if 20% of the bootstrap estimates of a particular coefficient are negative, the fact that the P-Value for rejection of the null hypothesis of negativity was .01 in the original regression should interpreted with great caution.

Accordingly, we construct a bootstrap function called `OLSBS`, where the last two letters of the function might be viewed with humour by those econometricians who don't like bootstrapping. Despite such objections, the technique is one which every budding econometrician should understand and know how to use. The function presented below takes two arguments. The first is the number of bootstrap runs, which I usually set at 1000. The second is a list of the coefficient names, where each name is surrounded by quotation marks with the distance between the left and right quotation marks being 12 spaces and with the names entered next to the right-most quotation mark. The call to the function in the case of the regression we have been analyzing is

```
(OLSBS 1000 (list "    Constant" "  TBill-Rate" "    Log-RGDP"))
```

and the code in the function is as follows.

```
(defun OLSBS (x y)
"Args: (x)
Bootstraps the coefficient values of the previous printed regression
and writes the percentiles to the screen. The first argument is
the number of bootstrap runs and the second is a list of words
defining the names of the variables, each being at the right side
of a 12 space gap between the quotation marks defining the word."
(def runnums x)
(def numspicked (sample (iseq 0 (- OLSnobs 1)) OLSnobs t))
(def bootedres (repeat 0 OLSnobs))
(dotimes (i OLSnobs)
(setf (select bootedres i)(select OLSresids (select numspicked i)))
) ; end dotimes i
(def newyval (bind-columns (+ OLSfitted bootedres)))
(def coefmat (bind-rows OLScoefs))
;
(dotimes (i runnums)
(runOLS newyval RXmat)
(def coefmat (bind-rows coefmat (transpose coefs)))
(def numspicked (sample (iseq 0 (- nobs 1)) nobs t))
(dotimes (i (- nobs 1))
(setf (select bootedres i)(select OLSresids (select numspicked i)))
) ; end dotimes i
(def newyval (bind-columns (+ OLSfitted bootedres)))
) ; end dotimes i (runnums)
(def coefmat (remove-first-rows 1 coefmat))
;
(def coeflist (column-list coefmat))
(def qlist01 (quantile (select coeflist 0) .01))
(def qlist025 (quantile (select coeflist 0) .025))
(def qlist05  (quantile (select coeflist 0) .05))
(def qlist1   (quantile (select coeflist 0) .1))
(def qlist25  (quantile (select coeflist 0) .25))
(def medlist  (quantile (select coeflist 0) .50))
(def qlist75  (quantile (select coeflist 0) .75))
(def qlist9   (quantile (select coeflist 0) .9))
(def qlist95  (quantile (select coeflist 0) .95))
(def qlist975 (quantile (select coeflist 0) .975))
```

```
(def qlist99  (quantile (select coeflist 0) .975))
(def meanlist (mean (select coeflist 0)))
;
(dotimes (i (- (length coeflist) 1))
(def qlist01 (combine qlist01 (quantile
(select coeflist (+ i 1)) .01)))
(def qlist025 (combine qlist025 (quantile
(select coeflist (+ i 1)) .025)))
(def qlist05  (combine qlist05 (quantile
(select coeflist (+ i 1)) .05)))
(def qlist1   (combine qlist1 (quantile
(select coeflist (+ i 1)) .1)))
(def qlist25  (combine qlist25 (quantile
(select coeflist (+ i 1)) .25)))
(def medlist  (combine medlist (quantile
(select coeflist (+ i 1)) .50)))
(def qlist75  (combine qlist75 (quantile
(select coeflist (+ i 1)) .75)))
(def qlist9   (combine qlist9 (quantile
(select coeflist (+ i 1)) .9)))
(def qlist95  (combine qlist95 (quantile
(select coeflist (+ i 1)) .95)))
(def qlist975 (combine qlist975 (quantile
(select coeflist (+ i 1)) .975)))
(def qlist99  (combine qlist99 (quantile
(select coeflist (+ i 1)) .99)))
(def meanlist (combine meanlist (mean
(select coeflist (+ i 1)))))
) ; end dotimes
;
(def nummat (bind-rows qlist01 qlist025 qlist05 qlist1 qlist25
qlist75 qlist9 qlist95 qlist975 qlist99 medlist meanlist))
;
(def qnames (list ".01" ".025" ".05" ".10" ".25" ".75" ".9" ".95"
".975" ".99" "median" "mean"))
(def widemat (bind-columns qnames nummat))
(def fullmat (bind-rows (combine "Quantiles" y) widemat))
(terpri)
```

```
(princ "BOOTSTRAPPED COEFFICIENTS")(terpri)(terpri)
(def rcnum (array-dimensions fullmat))(dotimes (j (select rcnum 0))
(dotimes (i (select rcnum 1))(format t "~12,4f" (aref fullmat j i)))
(terpri))
) ; end of function
```

The code uses the variables `OLSnobs`, `OLSresids`, `OLSfitted` and `OLScoefs` which are the special names attached to the corresponding variables at the end of the `OLSreg` function—this was done because variables with the original names are written over by the `LMSC` function. Most of the coding in the above function is standard, but there are a number of segments of this code that need to be explained. The lines of code

```
(def numspicked (sample (iseq 0 (- OLSnobs 1)) OLSnobs t))
(def bootedres (repeat 0 OLSnobs))
(dotimes (i OLSnobs)
(setf (select bootedres i)(select OLSresids (select numspicked i)))
) ; end dotimes i
(def newyval (bind-columns (+ OLSfitted bootedres)))
(def coefmat (bind-rows OLScoefs))
```

first create a list called `numspicked` using the `sample` function which produces an ordering of a list of integers starting at `0` of length equal to `OLSnobs`. This ordering of the elements of `OLSresids` is then copied to a new series of residuals called `bootedres`. These residuals are then added to `OLSfitted` to obtain a new version of the dependent variable of the original regression. Also a row vector of the original coefficients is constructed and given the name `coefmat`.

Then a `dotimes` loop is constructed in which a second `dotimes` loop is embedded.

```
(dotimes (i runnums)
(runOLS newyval RXmat)
(def coefmat (bind-rows coefmat (transpose coefs)))
(def numspicked (sample (iseq 0 (- nobs 1)) nobs t))
(dotimes (i (- nobs 1))
(setf (select bootedres i)(select OLSresids (select numspicked i)))
) ; end dotimes i
(def newyval (bind-columns (+ OLSfitted bootedres)))
) ; end dotimes i (runnums)
```

The first of these does `runnums` = 1000 loops, running a regression using the previously constructed new dependent variable vector, adding its coefficient vector to the bottom of `coefmat`, then creating a new `numspicked` list and a corresponding new `bootedres` list which it adds to the `OLSfitted` list to create a new dependent variable vector which it uses in the next regression, repeating this process 1000 times. At this point, all the bootstrapped coefficients are in the matrix `coefmat` from which, after the first row giving the coefficients of the original regression is removed, a list of three lists of bootstrapped values for the three coefficients, called `coeflist` is extracted using the `column-list` function. Then the quantiles for the first of the three lists in `coeflists`, representing the constant term, are extracted one by one and placed in separate lists called `qlist01`, `qlist025`, `qlist05`, etc. A list `meanlist` is also created in this process. At this point, all these lists have only one element. Then a `dotimes` loop is created which adds the relevant quantiles and means for the remaining coefficient lists in `coeflist`. It does this by using the function `combine` which takes as its first argument a particular quantile list and as its second argument the new element to be added to that list. These quantile lists are then bound together as rows in a numerical matrix, to which a column of quantile names is then bound. The list of variable names given in the second argument of the function is then bound to the word "Quantiles" and added as a first row to the matrix of bootstrapped results. Finally, this matrix is written, along with a heading, to the screen using embedded `dotimes i` and `dotimes j` loops in the code segment

```
(princ "BOOTSTRAPPED COEFFICIENTS")(terpri)(terpri)
(def rcnum (array-dimensions fullmat))
(dotimes (j (select rcnum 0))
(dotimes (i (select rcnum 1))(format t "~12,4f" (aref fullmat j i))
) ; end dotimes i
(terpri)
) ; end dotimes j
```

In the present case, the output of the function is as follows.

```
BOOTSTRAPPED COEFFICIENTS

Quantiles       Constant  TBill-Rate    Log-RGDP
.01               3.1248     -0.0232      0.3954
.025              3.1656     -0.0226      0.3998
.05               3.1988     -0.0222      0.4036
.10               3.2262     -0.0216      0.4073
.25               3.2793     -0.0206      0.4130
.75               3.4070     -0.0183      0.4270
.9                3.4619     -0.0172      0.4330
.95               3.4957     -0.0166      0.4363
.975              3.5290     -0.0162      0.4398
.99               3.5290     -0.0154      0.4438
median            3.3431     -0.0194      0.4199
mean              3.3439     -0.0194      0.4199
```

Comparing these quantiles and the mean with the coefficients of the original regression indicates that the mean and median values of each coefficient are almost equal and differ very little from the coefficients of the original regression. And the ranges between the .01 quantiles and the .99 quantiles are not of a major magnitude, giving us no reason at this point to question the original regression results although that might happen upon subsequent further analysis.

All the statistical analysis of the demand for M1 regression in this section is programmed in the`XLispStat` batch file `xlspsect.lsp` and the output from running this batch file is in the file `xlspsect.lou`.

Our Function File

All the functions we created above are collected together in the file `ourfuncs.lsp` which we now load into `XLispStat` every time we do statistical work using that program.

It is useful to add a few additional functions to our `ourfuncs.lsp` file so that we can do basic day-to-day work with `XLispStat`. You can examine the code for these functions by accessing and reading the `ourfuncs.lsp` file with your text editor. Except where noted below, the code in these added functions uses `XLispStat` functions that were used above.

The first function added is one which enables us to construct a date list for series read in using the `read-data-columns` function when the text file being read by that function does not have an appropriate date list as its first column. Our new function, called `setdates`, takes three arguments—first, a series for which the date list is being constructed, then the date of the first observation in that series, and finally the frequency of the series for which the date list is being created. Additional functions added enable us to remove `n` last rows, `n` first columns, `n` last columns (where `n` is a positive integer), or a selected row or column from a matrix and to write any selected row or column of a matrix as a list. These functions are

`(remove-last-rows n x)` —removes the last `n` rows of matrix `x`
`(remove-first-columns n x)` —removes the first `n` columns of matrix `x`
`(remove-last-columns n x)` —removes the last `n` columns of matrix `x`
`(remove-selected-row n x)` —removes row `n` of matrix `x`
`(remove-selected-column n x)` —removes column `n` of matrix `x`
`(copy-matrix-row (n x)` —copies row `n` of matrix `x` to a list
`(copy-matrix-column (n x)` —copies column `n` of matrix `x` to a list

Keep in mind that numbering starts at 0 and that the result produced by these functions must be assigned a name using the function `def`.

`XLispStat` has a function `print-matrix` which takes as its only argument the matrix to be printed and prints that matrix to the screen. Often the numbers will be in scientific notation and the result looks rather messy. Accordingly, it is useful for us to construct a `write-matrix` function to print a more orderly presentation of the matrix to the screen. The code for our function, which uses functions already used frequently above, is as follows.

```
(defun write-matrix (x)
"Args:  (x)
Prints a matrix on screen with format 12,3f."
(def rcnum (array-dimensions x))(dotimes (j (select rcnum 0))
(dotimes (i (select rcnum 1))(format t " 12,3f" (aref x j i))
) ;end dotimes
(terpri))
) ; end of function
```

The matrix prints all the numbers to three decimal places in columns which are twelve characters wide.

Also it is useful to be able to write a matrix to a file that can be loaded into another program to plot the variables it contains in a better way than can be done with `XLispStat`. This function, which we call `write-matrix-to-file` involves only a slight addition to and modification of the one above.

```
(defun write-matrix-to-file (x y)
"Args:  (x y)
Writes the matrix x to the file y."
(setf f (open y :direction :output))
(def rcnum (array-dimensions x))(dotimes (j (select rcnum 0))
(dotimes (i (select rcnum 1))(format f " 12,3f" (aref x j i)))
(terpri f))
(close f)
) ; end of function
```

Notice the addition of the line
`(setf f (open y :direction :output))`
which sets up the second argument we give to the function as a file `f` into which output is to be printed. Then notice that the `format` function, two lines below, takes as its first argument the file denoted by `f` instead of the screen denoted by `t`.

Of course, as you might expect, `XLispStat` has a built-in OLS regression function of its own. The function is called `regression-model`, which takes as its first argument the matrix of independent variables (excluding the constant term) or a list of those same variables and as its second argument a list representing the dependent variable. Like the plot functions discussed previously, the `regression-model` function produces an object which we can send instructions to and ask questions. If we do not want to include a constant we add a key-word `:intercept nil` as the third argument to the function. If we do not want to print the results, we add the key-word `:print nil` as an argument.

To do our demand for U.S. M1 regression using the `regression-model` function we can enter either of the following two lines of code

```
(def M1REG (regression-model (list US3MTBR LUSRGDP) LUSRM1))
```
or
```
(def M1REG (regression-model (bind-columns US3MTBR LUSRGDP) LUSRM1))
```

which will produce the following output.

```
Least Squares Estimates:

Constant                   3.34776      (9.303677E-2)
Variable 0                -1.941507E-2  (1.720993E-3)
Variable 1                0.419451      (1.036210E-2)

R Squared:                0.906446
Sigma hat:                 6.848318E-2
Number of cases:               205
Degrees of freedom:            202

M1REG}
```

We can send the regression object `M1REG` a request for help as follows:

```
> (send M1REG :help)
loading in help file information - this will take a minute ...done
REGRESSION-MODEL-PROTO
Normal Linear Regression Model
Help is available on the following:

:ADD-METHOD :ADD-SLOT :BASIS :CASE-LABELS :COEF-ESTIMATES
:COEF-STANDARD-ERRORS :COMPUTE :COOKS-DISTANCES :DELETE-DOCUMENTATION
:DELETE-METHOD :DELETE-SLOT :DF :DISPLAY :DOC-TOPICS :DOCUMENTATION
:EXTERNALLY-STUDENTIZED-RESIDUALS :FIT-VALUES :GET-METHOD :HAS-METHOD
:HAS-SLOT :HELP :INCLUDED :INTERCEPT :INTERNAL-DOC :ISNEW :LEVERAGES
:METHOD-SELECTORS :NEW :NUM-CASES :NUM-COEFS :NUM-INCLUDED :OWN-METHODS
:OWN-SLOTS :PARENTS :PLOT-BAYES-RESIDUALS :PLOT-RESIDUALS :PRECEDENCE-LIST
:PREDICTOR-NAMES :PRINT :PROTO R-SQUARED :RAW-RESIDUALS :REPARENT
:RESIDUAL-SUM-OF-SQUARES :RESIDUALS :RESPONSE-NAME :RETYPE :SAVE :SHOW
:SIGMA-HAT :SLOT-NAMES :SLOT-VALUE :STUDENTIZED-RESIDUALS :SUM-OF-SQUARES
:SWEEP-MATRIX :TOTAL-SUM-OF-SQUARES :WEIGHTS :X :X-MATRIX :XTXINV :Y
NIL
```

We can ask our regression object `M1REG` for information on many of the topics listed above. For example:

```
> (send M1REG :COEF-ESTIMATES)
(3.347763684601947 -0.019415067242538744 0.41945052715781467)
> (send M1REG :COEF-STANDARD-ERRORS)
(0.09303677467363122 0.0017209927145427299 0.010362096755322637)
>
```

Or, more appropriately, we can obtain and assign names to a whole range of elements generated by the regression.

```
> (def OLSresids (send M1REG :RESIDUALS))
> (def OLSfitted (send M1REG :FIT-VALUES))
> (def OLScoefs (send M1REG :COEF-ESTIMATES))
> (def OLSnobs (send M1REG :NUM-CASES))
> (def OLSdf (send M1REG :DF))
> (def RXMat (send M1REG :X-MATRIX))
> (def RSSQ (send M1REG :RESIDUAL-SUM-OF-SQUARES))
> (def TSSQ (send M1REG :TOTAL-SUM-OF-SQUARES))
> (def RSQ (send M1REG :R-SQUARED))
> (def DEPVAR (send M1REG :Y))
```

When doing more sophisticated tasks with `XLispStat` than we did in this section it will be convenient to use the `regression-model` instead of the `runOLS` function we developed earlier because we can give the regression objects names and the results produced by the regressions will remain inside those objects until we request them and therefore will not be overwritten in subsequent code. Obviously, however, we can produce a much better printout of the results than that done by the `regression-model` function.

The above U.S M1 results are produced using the script `xlspsect.lsp`, with the results saved in the output file `xlspsect.lou` which is created by naming `xlspsect.lou` as the `dribble` file before the `xlspsect.lsp` batch file is loaded.

Exercise

Write an `XLispStat` batch file to do the same analysis of the demand for U.S. M2 as was done for U.S. M1 in the analysis above.

2. The Statistical Program R

Now we outline the process of doing basic statistical analysis using the freely available program R. To obtain this program over the internet, just go to www.r-project.org and download and set it up. Two manuals, *An Introduction to R* and *R Reference Manual*, along with other options, are made available in PDF form when you click on (help) after loading the program. Also, you should download the paper *Econometrics in R*, written by Grant V. Farnsworth, which will provide valuable help, from cran.r-project.org/doc/contrib/Farnsworth-EconometricsInR.pdf.

The first thing you should do after clicking on the R icon and loading the program is to click on File and then on Change dir... and follow the prompts to to focus the program's attention on the directory in Windows that you will be working out of. Then the same data file we used in the case of XLispStat can be loaded, with the group of series being called usqdata, using the command

```
> usqdata <- read.table("statcomp.tab",header=TRUE)
```

The group of series loaded can then be checked by issuing the command

```
> names(usqdata)
```

which will prompt the response

```
[1] "X.YEAR"  "USGDP"   "US3MTBR" "USIPD"   "USCPI"   "USM1"    "USM2"
```

Actually, the best procedure for accomplishing this is to write a text file using any text editor (the one provided by MS-Windows will be sufficient) containing the two code-lines

```
usqdata <- read.table("statcomp.tab",header=TRUE)
names(usqdata)
```

and then, after pointing R to the directory you are working in, click on the left-most icon above the R Console to load that text file into the text editor provided by the R program. You can then write all subsequent code using that editor and run that code in R simply by highlighting text in that file from the bottom to the top and then clicking on the third icon from the left above the R Console. The results will appear in the Console and necessary

corrections to your script can be made in the `R Editor` and the script re-run until you get things right. At any point, your script can be saved by clicking on the second icon from the left above the `R Console` when the focus is on the `R Editor`. And you can print your script and your results by clicking on the printer icon when the focus is on the material you want printed. To save your results in a text file, you must copy the material in the console and paste that material into a blank file in your text editor and then save that file under an appropriate name—I find it convenient to use the same name as my script file except for the suffix which I call `.Rou`.

To bring all of the individual series into the workspace so that they can be accessed by simply typing their names, we enter the code line

```
attach(usqdata)
```

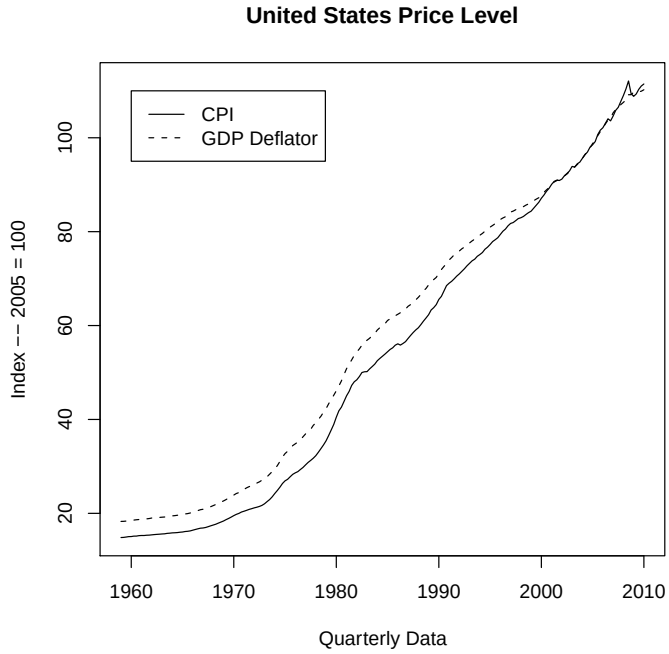
When working with time-series data, we then have to set up the variables we will want to later use in time-series form using the code

```
USGDP <- ts(usqdata\$USGDP,start=c(1959,1),end=c(2010,1),frequency=4)
US3MTBR <- ts(usqdata\$US3MTBR,start=c(1959,1),end=c(2010,1),frequency=4)
USIPD <- ts(usqdata\$USIPD,start=c(1959,1),end=c(2010,1),frequency=4)
USCPI <- ts(usqdata\$USCPI,start=c(1959,1),end=c(2010,1),frequency=4)
USM1 <- ts(usqdata\$USM1,start=c(1959,1),end=c(2010,1),frequency=4)
USM2 <- ts(usqdata\$USM2,start=c(1959,1),end=c(2010,1),frequency=4)
```

Now that our data have been loaded and set up, we can begin our analysis. The first item of interest is whether the U.S. inflation rate can be best approximated by movements in the consumer price index or the implicit GDP deflator. We can plot the two series by entering the following code

```
plot(USCPI,type="l",
xlab="Quarterly Data",
ylab="Index -- 2005 = 100",
main="United States Price Level")
lines(USIPD,lty=2)
legend(1960,110,c("CPI","GDP Deflator"),lty=c(1,2))
```

and when the code is run obtain the exact figure below.



It makes better sense to look at the year-over-year inflation rates calculated from these two alternative price level estimates. To obtain these, we add the code

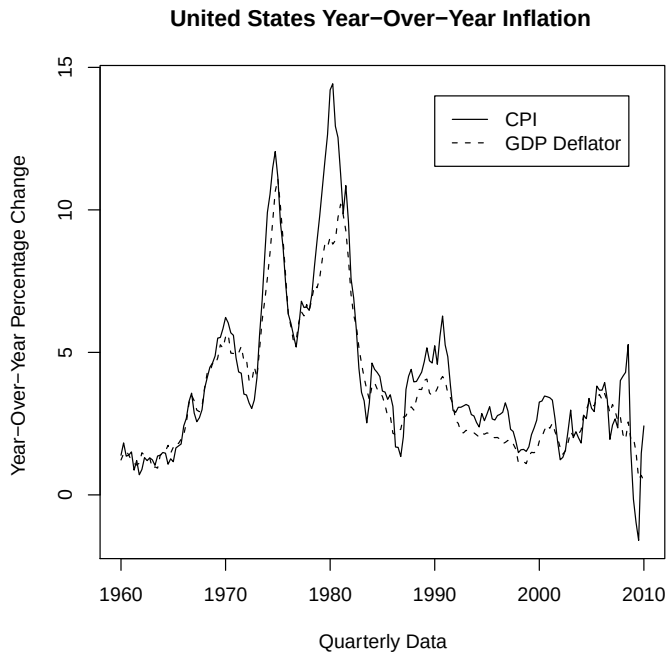
```
USCPIL4 <- lag(USCPI, -4)
USIPDL4 <- lag(USIPD, -4)
# Set the time-series properties of these new series
USCPIL4 <- ts(USCPIL4,start=c(1960,1),end=c(2010,1),frequency=4)
USIPDL4 <- ts(USIPDL4,start=c(1960,1),end=c(2010,1),frequency=4)
#
YYGCPI <- 100*(USCPI - USCPIL4)/USCPIL4
YYGIPD <- 100*(USIPD - USIPDL4)/USIPDL4
```

and then the appropriate code for plotting these year-over-year inflation rates.

```
plot(YYGCPI,type="l",
xlab="Quarterly Data",
ylab="Year-Over-Year Percentage Change",
```

```
main="United States Year-Over-Year Inflation")
lines(YYGIPD,lty=2)
legend(1990,14,c("CPI","GDP Deflator"),lty=c(1,2))
```

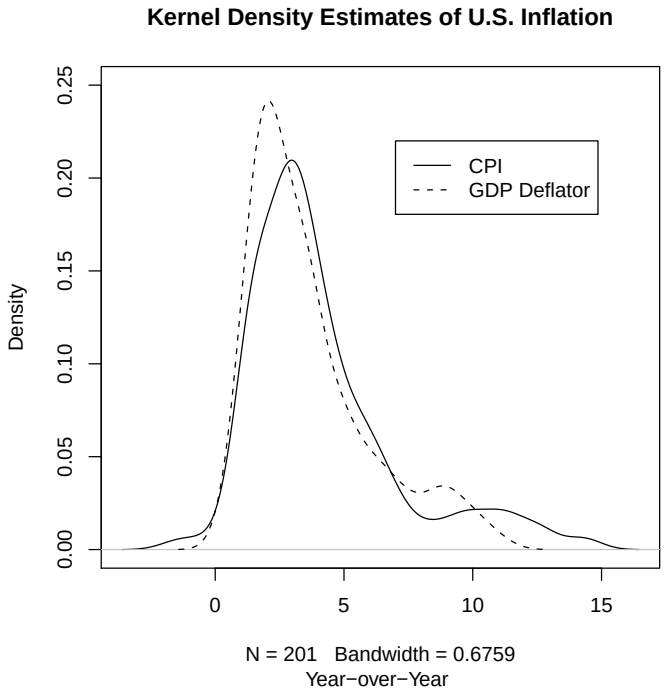
which produces the figure below.



It is also useful in visualizing the relationship between the two series to construct and plot their kernel densities, which are fitted smooth functions approximating the probability distributions of the two inflation rates. This is done using the code

```
DENYYGCPI <- density(YYGCPI)
DENYYGIPD <- density(YYGIPD)

plot(DENYYGCPI,
ylim=c(0,.25),
main="Kernel Density Estimates of U.S. Inflation",
sub="Year-over-Year")
lines(DENYYGIPD,lty=2)
legend(7,.22,c("CPI","GDP Deflator"),lty=c(1,2))
```

which produces the figure below.



It is clear that the GDP-deflator based inflation rate is more concentrated around a lower modal value and less variable than the CPI based inflation rate. Specific details about the nature of these two inflation rate series can be obtained using the code lines that are presented, along with the program's response, below.

```
# Functions to give details of YYGCPI and YYGIPD
> summary(YYGCPI)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 -1.595   2.254   3.300   4.104   5.160  14.430
> summary(YYGIPD)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.4868  1.9920  2.9930  3.6540  4.7150 11.0900
```

The information presented above plus additional features of the two series can be obtained by applying the individual functions as follows. The results are also presented along with my comments included to the right of # characters (delineating material which the program will ignore).

```
> mean(YYGCPI)
[1] 4.104037
> median(YYGCPI)
[1] 3.299757
> var(YYGCPI)
[1] 8.467032
> sd(YYGCPI)
[1] 2.909816
> mean(YYGIPD)
[1] 3.653965
> median(YYGIPD)
[1] 2.992598
> var(YYGIPD)
[1] 5.602494
> sd(YYGIPD)
[1] 2.366959
> quantile(YYGCPI, .25)
     25%
2.253521
> quantile(YYGCPI, .75)
     75%
5.160193
> quantile(YYGCPI, .75) - quantile(YYGCPI, .25) # inter-quartile range
     75%
2.906672
> max(YYGCPI)
[1] 14.42577
> min(YYGCPI)
[1] -1.595183
> max(YYGCPI) - min(YYGCPI)
[1] 16.02095
> range(YYGCPI)
[1] -1.595183 14.425770
> #
```

It is not clear why the first number, 75%, is produced by my inter-quartile range calculation, although the number resulting from the calculation, 2.906672, is is correct. While I was able to find a `range` function in `R`, it was impossible

to find one giving the inter-quartile range.

Since the GDP deflator more or less directly estimates the price of output, it would seem best to use it in calculating real M1 and real GDP for use in a regression that attempts to estimate the demand function for money. Also, these real variables are converted to natural logarithms before running the regression. The code required is as follows.

```
> USRGDP <- 100*USGDP/USIPD   # Calculate real GDP
> USRM1 <- 100*USM1/USIPD     # Calculate real M1
> LUSRGDP <- log(USRGDP)      # Natural logarithm of real GDP
> LUSRM1 <- log(USRM1)        # Natural logarithm of real M1

# Demand for Money Regression -- 1959:Q1 through 2010:Q1
dmreg <- lm(LUSRM1 ~ US3MTBR + LUSRGDP)
#
# Present basic regression results
#
summary(dmreg)
```

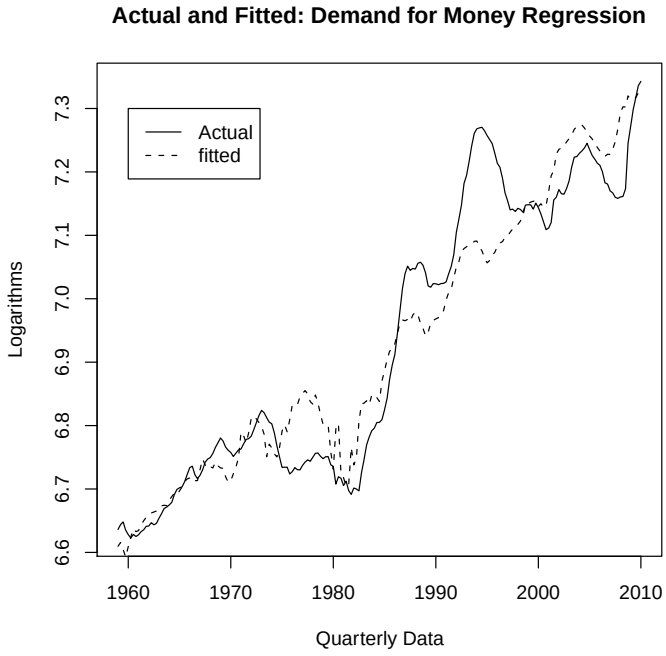
This code produces the regression results below.

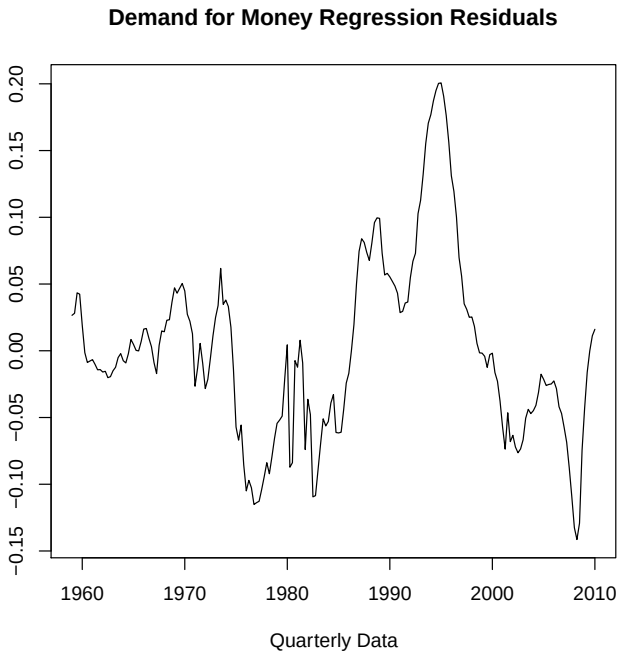
```
Call:
lm(formula = LUSRM1 ~ US3MTBR + LUSRGDP)

Residuals:
     Min       1Q   Median       3Q      Max
-0.14142 -0.04701 -0.00664  0.03474  0.20066

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.347764   0.093037   35.98   <2e-16 ***
US3MTBR     -0.019415   0.001721  -11.28   <2e-16 ***
LUSRGDP      0.419451   0.010362   40.48   <2e-16 ***
---
Signif. codes:  0 *** 0.001 ** 0.01 * 0.05 . 0.1   1

Residual standard error: 0.06848 on 202 degrees of freedom
Multiple R-squared: 0.9064,     Adjusted R-squared: 0.9055
F-statistic: 978.6 on 2 and 202 DF,  p-value: < 2.2e-16
```

The next step is to retrieve various important statistics produced by the regression, among them the regression residuals which will ultimately be plotted.

```
> dmoutput <- summary(dmreg)
> SSR <- deviance(dmreg)
> DMCOEF <- dmreg$coefficients  # regression coeficients
> DF <- dmreg$df
> DMFIT <- dmreg$fitted.values
> DMRES <- dmreg$residuals
> s <- dmoutput$sigma  # standard error of regression residuals
> dmRSQ <- dmoutput$r.squared
> dmCovMat <- s^2*dmoutput$cov  # variance-covariance matrix of coefficients
```

The plots of the actual and fitted values and the residuals from this regression are presented below and on the next page.





Next we need to know how to conduct a Breusch-Pagan test for heteroskedasticity of the residuals. Recall from our `XLispStat` programming that this involves regressing the squared residuals on the some or all of the independent variables in the original regression and multiplying the R^2 by the number of observations to obtain a Chi-square statistic having degrees of freedom equal to the number of regressors other than the constant term. This test is performed by entering the code line

```
bptest(dmreg)
```

and produces the result

```
        studentized Breusch-Pagan test

data:  dmreg
BP = 14.3522, df = 2, p-value = 0.0007647
```

The next issue is that of testing for serial correlation in the regression residuals. The three most common of these tests are all provided by R. Consider first the Breusch-Godfrey LM test which obtains the relevant Chi-square statistic by regressing the residuals from the original regresson on the lagged values of these residuals of some desired order as well as on the independent variables in the original regression, and then multiplying the R^2 from that regression by the number of observations in the original regression. The code for a specification of 4 lags of the residuals, and the result, are as follows.

```
> bgtest(dmreg,order=4,type="Chisq")

        Breusch-Godfrey test for serial correlation of order 4

data:  dmreg
LM test = 193.2608, df = 4, p-value < 2.2e-16
```

where the Chi-square value with 4 degrees of freedom is 193.2608.

Next consider an alternative version of the Breusch-Godfrey LM test which obtains the F-statistic for the null-hypothesis that the lagged residuals explain none of the movements in the current residual. This involves running a second regression, called the restricted regression, of the residuals on the variables other than the lagged residuals and calculating the restricted sum of squared residuals, SSER, of this restricted regression. The F-statistic for testing the restriction is equal to

$$\mathtt{F} = \frac{(\mathtt{SSER} - \mathtt{SSEU})/4}{\mathtt{SSEU}/198}$$

where SSEU is the sum of squared residuals of the regression that included the four lagged residuals, the number of restrictions imposed is 4, and the degrees of freedom of the unrestricted regression is 198 —the number of observations (which equals 205) minus the number of coefficients (which equals 7) being estimated in the unrestricted regression. The number of observations in the unrestricted regression is the same as in the original demand-for-money regression because, following Davidson and MacKinnon,[7] all unavailable values

[7]Russell Davidson and James MacKinnon, *Estimation and Inference in Econometrics*, Oxford University Press, 1993, pages 358 and 359.

of the lagged residuals are set equal to zero. The code for this version of the LM-test, and the resulting response by R, are as follows.

```
> bgtest(dmreg,order=4,type="F")

        Breusch-Godfrey test for serial correlation of order 4

data:  dmreg
LM test = 814.915, df1 = 4, df2 = 198, p-value < 2.2e-16
```

The F-statistic is 814.915 with 4 degrees of freedom in the numerator and 198 degrees of freedom in the denominator.

This test differs from the one proposed by G.S. Maddala in two ways.[8] First, Maddala uses an approximation by which unavailable residuals retain their NA values and the restricted and unrestricted regressions are therefore based on 201 rather than 205 observations. Second, the procedure followed by Maddala imposes the condition of sufficient data under the null-hypothesis to generate a normally distributed standard error of the regression, with the result that the denominator of the F-statistic under the null-hypothesis equals unity and the numerator becomes a Chi-square statistic divided by the number of restrictions. Thus, multiplication of the F-statistic by the number of restrictions yields a Chi-square statistic with 4 degrees of freedom. The value of this statistic would be $4 \times 814.915 = 3259.66$ under the above conditions where the unavailable residuals are assigned zero values. Both the F-statistic and the resulting Chi-square statistic differ by small amounts from the values that would arise where the missing residuals are not replaced by zeros and the regression that generates the F-statistic starts later than the original demand-for-money regression by the number of periods, equal to the number of lags, for which lagged residuals are missing. This accounts for the difference between the numbers obtained here and those that resulted under our LMSC function programmed in XLispStat.

The practice of multiplying the F-statistic by the number of restrictions and treating the resulting statistic as Chi-square with degrees of freedom equal to that number of restrictions makes it much more likely that the null-hypothesis of no autocorrelation in the residuals will be rejected in any given case, adding power to the test by increasing the likelihood that the

[8] G.S. Maddala, *Introduction to Econometrics*, Macmillan, 1988, page 206.

alternative hypothesis of serial correlated residuals will be accepted when it is true. Since OLS regression results depend in important ways on the assumption of independently and identically distributed residuals, this is the more conservative approach.

An additional test for the presence of serial correlation in the regression residuals is the Ljung-Box Q test. This statistic is equal to

$$Q = T\,(T+2)\,\sum_{k=1}^{s}\frac{r_k^2}{T-k}$$

where T is the number of observations, r_k is the estimate of the k^{th} order serial correlation coefficient ρ_k and s is the order of serial correlation being tested. Recall that first-order serial correlation of the residuals is the correlation between the current residual and that residual once lagged, second order serial correlation is the correlation between the current residual and that residual twice lagged, and so forth. The statistic Q is distributed according the the Chi-square distribution with s degrees of freedom. If it exceeds the critical value, at least one of the r_k is significantly different from zero at the specified significance level. To run this test in `R` in the current situation, we insert the code

```
Box.test(DMRES, lag = 4, type = "Ljung")
```

and receive the response

```
        Box-Ljung test

data:  DMRES
X-squared = 664.5834, df = 4, p-value < 2.2e-16
```

where the `X` in the last line is a text-font representation of the Greek symbol χ.

In the case at hand, the residuals of our demand for money regression are most certainly highly serially correlated, a conclusion that is clearly substantiated by all four of the above alternative tests—in every case the P-Value is less than 0.00000000000000022.

A final issue is the bootstrapping of the regression coefficients. The code for doing this in R for the problem at hand is presented below. What is presented is a set of calculations, not a bootstrap function—accordingly, the calculations have to be adjusted to suit each different regression for which the bootstrapped coefficient estimates are to be obtained.

```
dmreg <- lm(LUSRM1 ~ US3MTBR + LUSRGDP)
summary(dmreg)
ORIGFITTED <- dmreg$fitted.values
ORIGRESID <- dmreg$residuals
#
COEFSCON <- c(1:1000)-c(1:1000)
COEFSTBR <- c(1:1000)-c(1:1000)
COEFSGDP <- c(1:1000)-c(1:1000)
for (i in 1:1000)
{
NEWRESID <- sample(ORIGRESID)
NEWLRM1 <- ORIGFITTED + NEWRESID
newdmreg <- lm(NEWLRM1 ~ US3MTBR + LUSRGDP)
NEWCOEFS <- newdmreg$coefficients
COEFSCON[i] <- NEWCOEFS[1]
COEFSTBR[i] <- NEWCOEFS[2]
COEFSGDP[i] <- NEWCOEFS[3]
}
```

The first line of code runs the regression and saves the results in the object `dmreg`. The second line tells R to print out the basic regression results. The third and fourth lines extract the fitted values and the residuals from this regression and save them under the new names `ORIGFITTED` and `ORIGRESID`. The three lines of code that follow construct three vectors, each containing 1000 zeros into which the bootstrapped estimates of the three coefficients will ultimately be placed—`COEFSCON` is for the constant term, `COEFSTBR` is for the T-Bill rate variable and `COEFSGDP` is for the real GDP variable. Next we begin a loop that will be passed through 1000 times with the index `i` taking values from 1 to 1000. In each loop we use the `sample` function to sample the original residuals. Then we add that new set of residuals to the fitted values to obtain `NEWLRM1` a new measure of the logarithm of real M1. Then we run a new regression with this as the dependent variable where the independent variables are the original ones. Then we extract the coefficients from this

new regression and replace the $\texttt{i}^{th}$ observations of each the three coefficient vectors with the appropriate coefficient estimate. After 1000 passes through the loop, we end up with three lists of bootstrapped coefficient estimates, one for each independent variable in the original regression.

There are many different ways to analyse these bootstrapped coefficient vectors. One can take various quantiles, as we did in the `XLispStat` function we wrote in that section. Here, we simply use the `summarize` function to extract the basics, obtaining the following results.

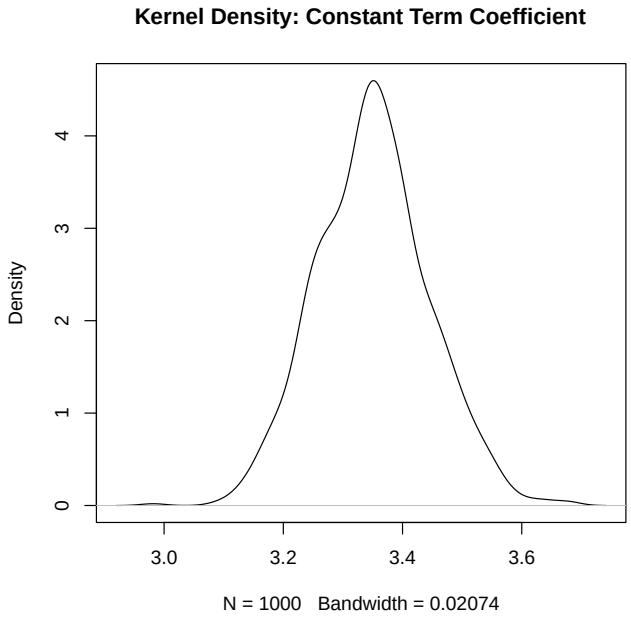
```
> summary(COEFSCON)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  3.058   3.283   3.349   3.348   3.411   3.630
> summary(COEFSTBR)
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.02383 -0.02059 -0.01948 -0.01943 -0.01823 -0.01386
> summary(COEFSGDP)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.3875  0.4125  0.4192  0.4195  0.4266  0.4514
```

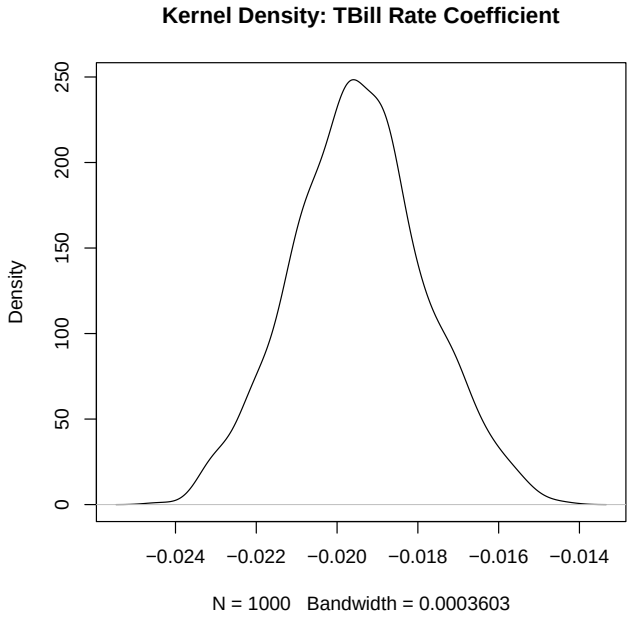
The medians and means compare quite favorably with our original coefficient estimates, which are 3.347764 for the Constant Term, -0.019415 for the T-Bill Rate coefficient and .419451 for the coefficient of log real GDP, and the maxima and minima suggest that the basic conclusions we can draw from our regression concerning the range of magnitudes of the coefficients are not reversed in any important way by the bootstrap evidence. Another interesting view of the bootstrapped coefficient estimates is their kernel density plots, the code for which is presented below, along with the plots that follow.

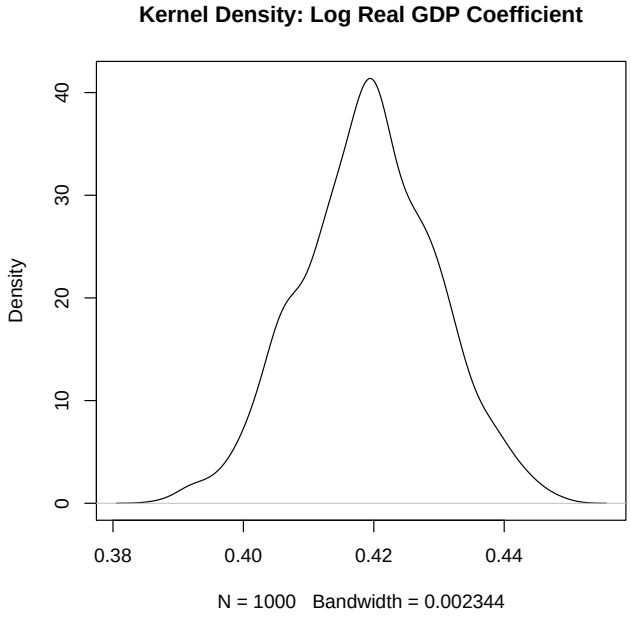
```
> DENCOEFSCON <- density(COEFSCON)
> DENCOEFSTBR <- density(COEFSTBR)
> DENCOEFSGDP <- density(COEFSGDP)
> #
> plot(DENCOEFSCON,
+ main="Kernel Density: Constant Term Coefficient")
> #
> plot(DENCOEFSTBR,
+ main="Kernel Density: TBill Rate Coefficient")
> #
```

```
> plot(DENCOEFSGDP,
+ main="Kernel Density: Log Real GDP Coefficient")
> #
```

In looking at the graphs below, it is important to understand that the probability density associated with any particular point on the horizontal axis is equal to the distance of the density curve from that axis multiplied by the bandwidth shown at the bottom.




Kernel Density: TBill Rate Coefficient
Density
0
50
100
150
200
250
−0.024
−0.022
−0.020
−0.018
−0.016
−0.014
N = 1000 Bandwidth = 0.0003603


Kernel Density: Log Real GDP Coefficient
Density
0
10
20
30
40
0.38
0.40
0.42
0.44
N = 1000 Bandwidth = 0.002344

Exercise

Write an `R` batch file to do the same analysis of the demand for U.S. M2 as was done for U.S. M1 in the analysis above. You will find the `R` code for the material in this section in the script file `Rsect.R` and the resulting output in the file `Rsect.Rou`.

3. Gretl (Gnu Regression, Econometrics and Time-series)

Now we turn to `Gretl`, the easiest program of the three to use. First, of course, you have to download the latest version of `Gretl` from the Web and set it up. At the time of this writing, you should go to

`http://gretl.sourceforge.net/win32/`

and obain the self-extracting zip file `gretl-1.9.5.exe`. Simply run this file and follow the prompts, making sure that you end up with a `Gretl icon` on your desktop.

The next task is to load your data into `Gretl`. As in the case of `XLispStat` and `R`, take the spreadsheet file into which you obtained your data off the web or from elsewhere, and copy a version that has as its first row the variable names along the top of the data columns, and a left-most column providing some sort of identification of the observations—for time-series data this will indicate the dates of the observations. Make sure that the observation column has a variable name in the upper-left cell of the spreadsheet, without the `;` character that you would put in front of it before saving it as a text file to be read by `XlispStat`. After saving this spreadsheet as an `.xls` file, load `Gretl` by clicking on its icon. Click on `File`, then `Open data`, and then `Import` and then choose `Excel`. In the window that appears, go to the directory you are working in and click on the Excel spreadsheet file—in the case at hand it is called `statcomp.xls`. Follow the prompts and you will see another window with a list of variables in it and, most likely, be given a little window telling you that the imported data has been interpreted as undated (cross-sectional) and asking if you would like to give it a time-series or panel interpretation. Answer yes, choose `Time series`, and then click on `Forward`. Choose quarterly in the present case, click on `Forward` again and set the date as `1959:1` in the window that appears. Then click again on `Forward` and choose `Yes` if the beginning and ending dates that appear are the correct ones. Now delete the variable that has the name of your column of observations by clicking on it and then pressing the right mouse button and selecting `Delete`. You can examine the remaining variables by double-clicking on them with the left mouse button.

At this point it would be desirable for you to add descriptions to the data series in the data file. Do this by highlighting each variable in turn using the left mouse button, then clicking on `Variable`, choosing `Edit Attributes`

and typing the appropriate descriptive material in the window that will be presented to you. You can also plot any variable by highlighting it and choosing `Variable` and then `Time-series plot`. Alternatively, you can plot two or more variables together by clicking on `View` and then `Graph specified vars`, selecting the type of plot, and then specifiying in the resulting window the variables you want to plot. By this point, it should be obvious how you can perform many statistical operations by pointing and clicking. Before going further, however, you should save your data file by clicking on `File`, then `Save data` and giving your data file a name with the suffix `.gdt`. You should probably choose the name `Gretl` suggests, which will be the same name as your Excel file with the suffix `.gdt` instead of `.xls`.

It turns out that the fastest way to work in `Gretl` is probably by feeding it prepared script files, so this is the way we will begin. To do this, exit `Gretl`, ignoring any prompt to save things (assuming that you have already saved your data file), and then load `Gretl` again. When you click on the `Gretl` icon, a screen will appear on which you should select `File` and then `Open data` and you will be prompted to load the data set that you had previously saved. Click on that name and the data window will again appear. Next, click on `Tools` and then `Command log` and and a window will appear containing the following information.

```
# Log started 2010-08-19 15:57
# Record of session commands.  Please note that this will
# likely require editing if it is to be run as a script.
open E:\DSLMHTML\STATCOMP\statcomp.gdt
```

which you should highlight, then `Copy` by selecting the third-icon from the right along the command line. Now open up your text editor and save this material in a new file, here named `gretsect.inc`. From now on, you can add commands to this file and execute it by clicking on `File`, then on `Script files`, and then on the name of the file. A script window will appear containing the material in the file. From this point forward, you can add material to the file in the script window and then click on the sixth icon from the left along the top of the window to execute the commands it contains. You have no choice but run the whole file unless you do as I do and insert your favorite single-word profanity in the file where you want the processing to stop—the program, being clean minded, will object to the profanity and processing will cease.

At this point we can write our script for subsequent processing. The script, presented below in its entirety, contains code lines which will do everything we did in R except bootstrap the regression coefficients. The meaning of the code should be obvious, although I have provided prompts by making comments to the right of the code lines. Gretl, like R ignores all material on a line to the right of the character #. At any time in Gretl you can click on help along the top of the data file to access detailed information about the program.

Also, when you want to find the code for a particular procedure it is often useful to click on that procedure in the data window and then check he command log. You can then copy he code from the command log into your script file. And, of course, the best way to figure out what coding to use is to steal the code from previously written script files.

```
# GRETL SCRIPT FOR STATISTICAL COMPUTING ANALYSIS
#
open E:\DSLMHTML\STATCOMP\statcomp.gdt
#
lags 4; USIPD USCPI  # Generate lags of 1 through 4 periods.
# Lags are denoted by the characters _1, _2, _3, and _4 added
# to the variable names.
#
genr YYGCPI = 100*(USCPI - USCPI_4)/USCPI_4  # Generate year-over-year
genr YYGIPD = 100*(USIPD-USIPD_4)/USIPD_4    #  inflation rates.
#
summary YYGCPI    # Provide information about the
summary YYGIPD    # two series.
QCPI05 = quantile(YYGCPI,0.05) # Calculate various quantiles
QCPI25 = quantile(YYGCPI,0.25) #  and print them to screen.
QCPI50 = quantile(YYGCPI,0.50)
QCPI75 = quantile(YYGCPI,0.75)
QCPI95 = quantile(YYGCPI,0.95)
#
QIPD05 = quantile(YYGIPD,0.05)
QIPD25 = quantile(YYGIPD,0.25)
QIPD50 = quantile(YYGIPD,0.50)
QIPD75 = quantile(YYGIPD,0.75)
QIPD95 = quantile(YYGIPD,0.95)
#
```

```
# RUN DEMAND FOR MONEY REGRESSION
#  after putting M1 and GDP in real terms
#   and taking the logarithms
#
genr USRM1 = 100*(USM1/USIPD)   # Calculate real values by deflating
genr USRGDP = 100*(USGDP/USIPD) #  the series by the GDP deflator.
genr LUSRM1 = log(USRM1)        # Generate logarithms of the
genr LUSRGDP = log(USRGDP)      #  two series.
#
ols LUSRM1 const US3MTBR LUSRGDP # Run the regression
RESIDS = $uhat  # Extract regression residuals and
FITTED = $yhat  #  fitted values for subsequent plotting
#
modtest --breusch-pagan   # test for heteroskedaticity
modtest  4 --autocorr     # tests for autocorrelation in residuals
#
```

The last two commands use the function `modtest`, giving it the instruction to first run a Breusch-Pagan test and then do tests for autocorrelation in the residuals using 4 lags.

Let us now have a look at the results, which `Gretl` presents in a very clear way.

```
gretl version 1.9.1
Current session: 2010-08-20 11:22
GRETL SCRIPT FOR STATISTICAL COMPUTING ANALYSIS
#
? open E:\DSLMHTML\STATCOMP\statcomp.gdt

Read datafile E:\DSLMHTML\STATCOMP\statcomp.gdt
periodicity: 4, maxobs: 205
observations range: 1959:1-2010:1

Listing 7 variables:
  0) const      1) USGDP      2) US3MTBR    3) USIPD      4) USCPI
  5) USM1       6) USM2
```

```
#
? lags 4; USIPD USCPI  # Generate lags of 1 through 4 periods.
Listing 15 variables:
  0) const      1) USGDP      2) US3MTBR    3) USIPD      4) USCPI
  5) USM1       6) USM2       7) USIPD_1    8) USIPD_2    9) USIPD_3
 10) USIPD_4   11) USCPI_1   12) USCPI_2   13) USCPI_3   14) USCPI_4

# Lags are denoted by the characters _1, _2, _3, and _4 added
# to the variable names.
#
? genr YYGCPI = 100*(USCPI - USCPI_4)/USCPI_4  # Generate year-over-year
Generated series YYGCPI (ID 15)
? genr YYGIPD = 100*(USIPD-USIPD_4)/USIPD_4    #  inflation rates.
Generated series YYGIPD (ID 16)
#
? summary YYGCPI    # Provide information.

Summary statistics, using the observations 1959:1 - 2010:1
for the variable 'YYGCPI' (201 valid observations)

  Mean                      4.1040
  Median                    3.2998
  Minimum                  -1.5952
  Maximum                   14.426
  Standard deviation        2.9098
  C.V.                     0.70901
  Skewness                  1.4524
  Ex. kurtosis              1.9968
```

```
#
? summary YYGIPD    # Provide information.

Summary statistics, using the observations 1959:1 - 2010:1
for the variable 'YYGIPD' (201 valid observations)

  Mean                      3.6540
  Median                    2.9926
  Minimum                  0.48682
  Maximum                   11.085
  Standard deviation        2.3670
  C.V.                     0.64778
  Skewness                  1.2175
  Ex. kurtosis             0.77655


? QCPI05 = quantile(YYGCPI,0.05) # Calculate various quantiles
Generated scalar QCPI05 = 1.20246
? QCPI25 = quantile(YYGCPI,0.25) #  and print them to screen.
Generated scalar QCPI25 = 2.23928
? QCPI50 = quantile(YYGCPI,0.50)
Generated scalar QCPI50 = 3.29976
? QCPI75 = quantile(YYGCPI,0.75)
Generated scalar QCPI75 = 5.17358
? QCPI95 = quantile(YYGCPI,0.95)
Generated scalar QCPI95 = 11.1064
#
? QIPD05 = quantile(YYGIPD,0.05)
Generated scalar QIPD05 = 1.11431
? QIPD25 = quantile(YYGIPD,0.25)
Generated scalar QIPD25 = 1.97793
? QIPD50 = quantile(YYGIPD,0.50)
Generated scalar QIPD50 = 2.9926
? QIPD75 = quantile(YYGIPD,0.75)
Generated scalar QIPD75 = 4.73358
? QIPD95 = quantile(YYGIPD,0.95)
Generated scalar QIPD95 = 8.93689
#
```

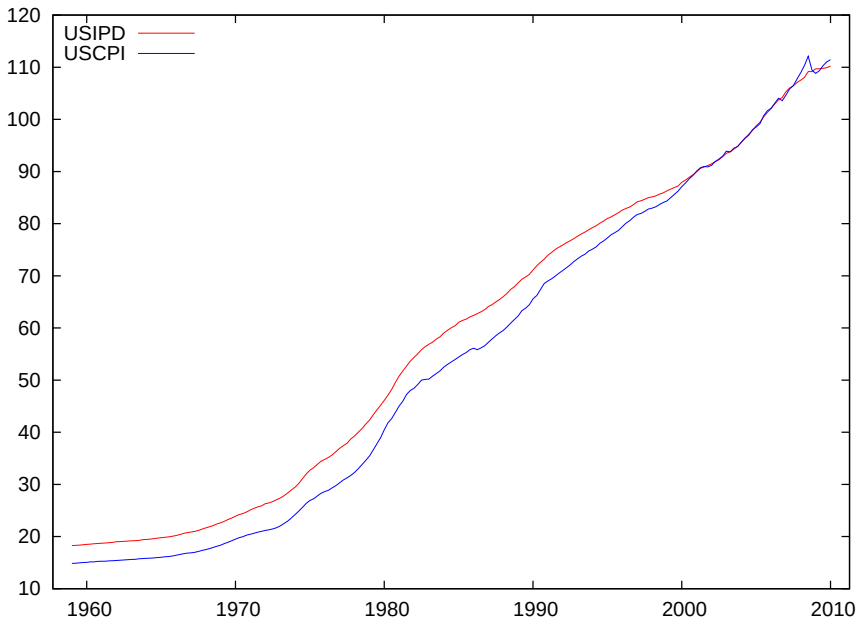
```
# RUN DEMAND FOR MONEY REGRESSION
#  after putting M1 and GDP in real terms
#   and taking the logarithms
#
? genr USRM1 = 100*(USM1/USIPD)   # Calculate real values by deflating
Generated series USRM1 (ID 17)
? genr USRGDP = 100*(USGDP/USIPD) #  the series by the GDP deflator.
Generated series USRGDP (ID 18)
? genr LUSRM1 = log(USRM1)        # Generate logarithms of the
Generated series LUSRM1 (ID 19)
? genr LUSRGDP = log(USRGDP)      #  two series.
Generated series LUSRGDP (ID 20)
#
? ols LUSRM1 const US3MTBR LUSRGDP # Run the regression

Model 1: OLS, using observations 1959:1-2010:1 (T = 205)
Dependent variable: LUSRM1

             coefficient   std. error   t-ratio    p-value
  ---------------------------------------------------------
  const       3.34776      0.0930368     35.98    8.49e-090 ***
  US3MTBR    -0.0194151    0.00172099   -11.28    3.31e-023 ***
  LUSRGDP     0.419451     0.0103621     40.48    7.14e-099 ***

Mean dependent var   6.931613   S.D. dependent var   0.222799
Sum squared resid    0.947369   S.E. of regression   0.068483
R-squared            0.906446   Adjusted R-squared   0.905520
F(2, 202)            978.5879   P-value(F)           1.2e-104
Log-likelihood       260.2679   Akaike criterion    -514.5359
Schwarz criterion   -504.5668   Hannan-Quinn        -510.5036
rho                  0.965846   Durbin-Watson        0.067818

Log-likelihood for USRM1 = -1160.71

? RESIDS = $uhat
Generated series RESIDS (ID 21)
? FITTED = $yhat
Generated series FITTED (ID 22)
```

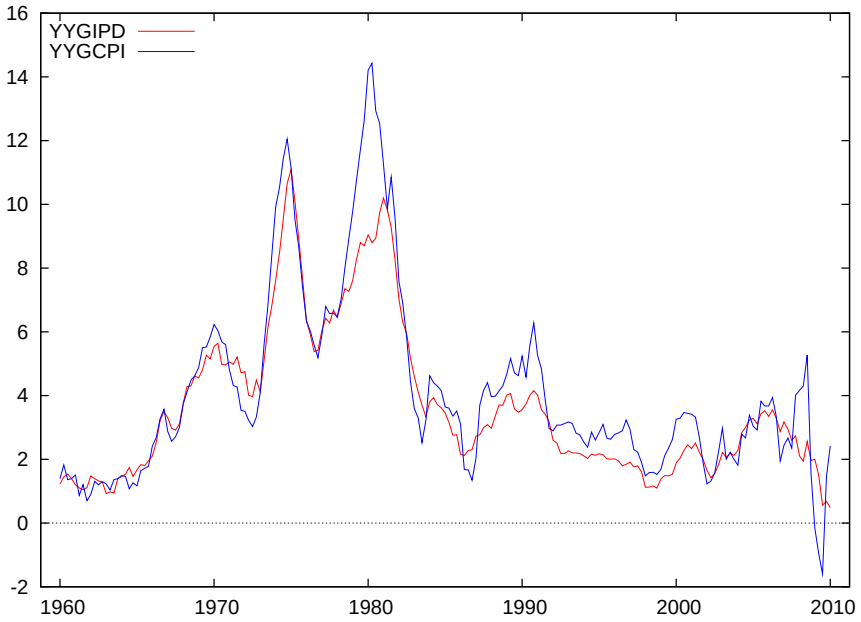
```
? modtest --breusch-pagan   # test for heteroskedaticity

Breusch-Pagan test for heteroskedasticity
OLS, using observations 1959:1-2010:1 (T = 205)
Dependent variable: scaled uhat^2

             coefficient   std. error   t-ratio   p-value
  ---------------------------------------------------------
  const      -7.19539      2.19599      -3.277    0.0012  ***
  US3MTBR    -0.00569001   0.0406214    -0.1401   0.8887
  LUSRGDP     0.935988     0.244581      3.827    0.0002  ***

  Explained sum of squares = 39.7335

Test statistic: LM = 19.866746,
with p-value = P(Chi-square(2) > 19.866746) = 0.000049

? modtest  4 --autocorr     # tests for autocorrelation in residuals

Breusch-Godfrey test for autocorrelation up to order 4
OLS, using observations 1959:1-2010:1 (T = 205)
Dependent variable: uhat

             coefficient    std. error    t-ratio     p-value
  ------------------------------------------------------------
  const       0.00429298    0.0224893      0.1909    0.8488
  US3MTBR    -0.000431786   0.000416595   -1.036     0.3012
  LUSRGDP    -0.000220610   0.00250467    -0.08808   0.9299
  uhat_1      1.25031       0.0679086     18.41      9.02e-045 ***
  uhat_2     -0.440575      0.107538      -4.097     6.10e-05  ***
  uhat_3      0.429152      0.107642       3.987     9.41e-05  ***
  uhat_4     -0.289821      0.0680998     -4.256     3.21e-05  ***

  Unadjusted R-squared = 0.942736

Test statistic: LMF = 814.914971,
with p-value = P(F(4,198) > 814.915) = 1.01e-121
```

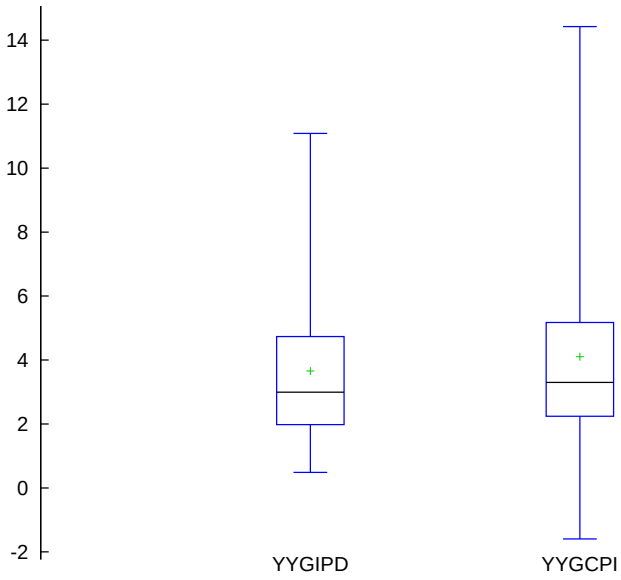
```
Alternative statistic: TR^2 = 193.260847,
with p-value = P(Chi-square(4) > 193.261) = 1.06e-040

Ljung-Box Q' = 664.583,
with p-value = P(Chi-square(4) > 664.583) = 1.62e-142
```

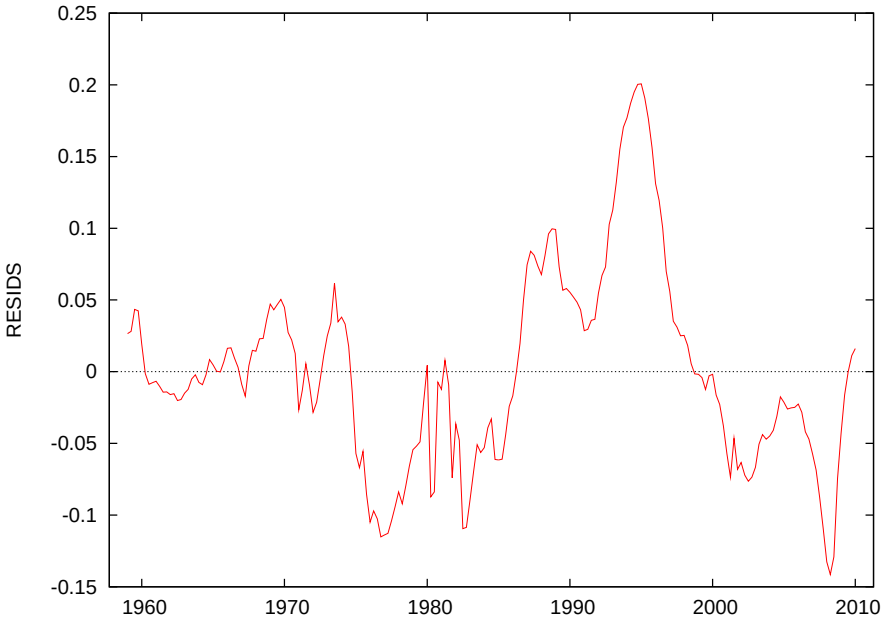
All of the summary statistics and regression results are consistent with the results produced by `XLispStat` and `R` except for the Breusch-Pagan test for heteroskedasticity which uses the non-studentized approach, and the tests for serial correlation in the regression residuals which use only a straight F-test like was used in `R` as well as the Chi-square test with a statistic obtained by multiplying the R^2 by the number of observations. The test procedure noted by Maddala, involving the multiplication of the F-statistic by the number of lagged residuals to obtain a Chi-square statistic, which we used in `XLispStat`, was not performed by either `Gretl` or `R`. Note that the Breusch-Godfrey serial correlation tests above follow the Davidson-MacKinnon procedure of replacing missing lagged values of the residuals by zero—the regression presented immediately above uses 205 observations.

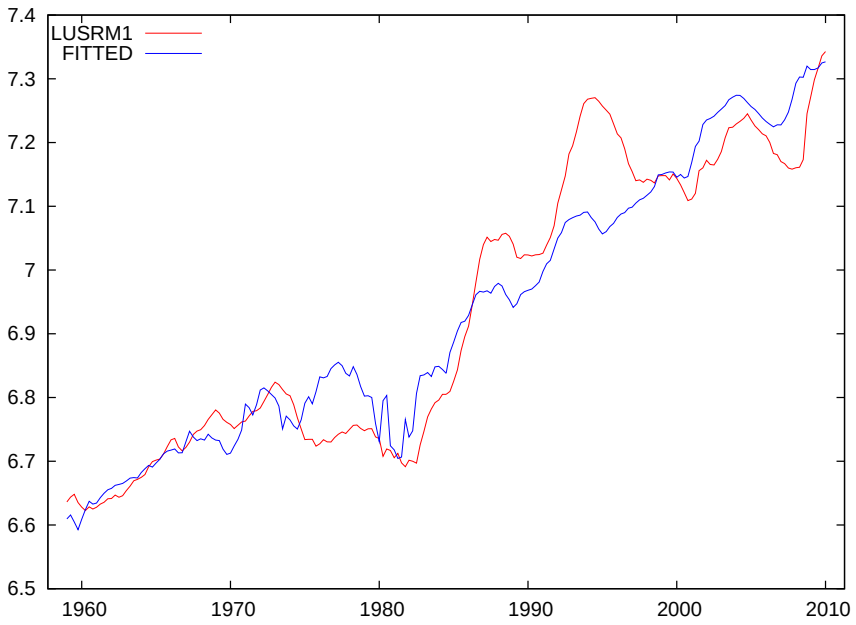
You will notice that there are no commands in our script file that deal directly with the plotting of particular variables of interest. This is because the easiest way to create plots is to do so from the data-file page after the script file has been executed. All series produced by the execution of the script will appear among the variables in the modified data file. To plot these, we just click on `View`, then on `Graph specific vars` and then choose the type of graph and the series to be included by following the prompts. When a graph appears, we can add a title, change the labels, and so forth by holding town the right-mouse button and choosing `Edit`. And we can save each of the plots by clicking on one of the `Save` commands. The five plots of special interest are presented below without editing the raw graphs. Kernel densities of the two inflation rate measures are not plotted because both could not be plotted on the same chart.


USIPD
USCPI
120
110
100
90
80
70
60
50
40
30
20
10
1960
1970
1980
1990
2000
2010


YYGIPD
YYGCPI
16
14
12
10
8
6
4
2
0
-2
1960
1970
1980
1990
2000
2010

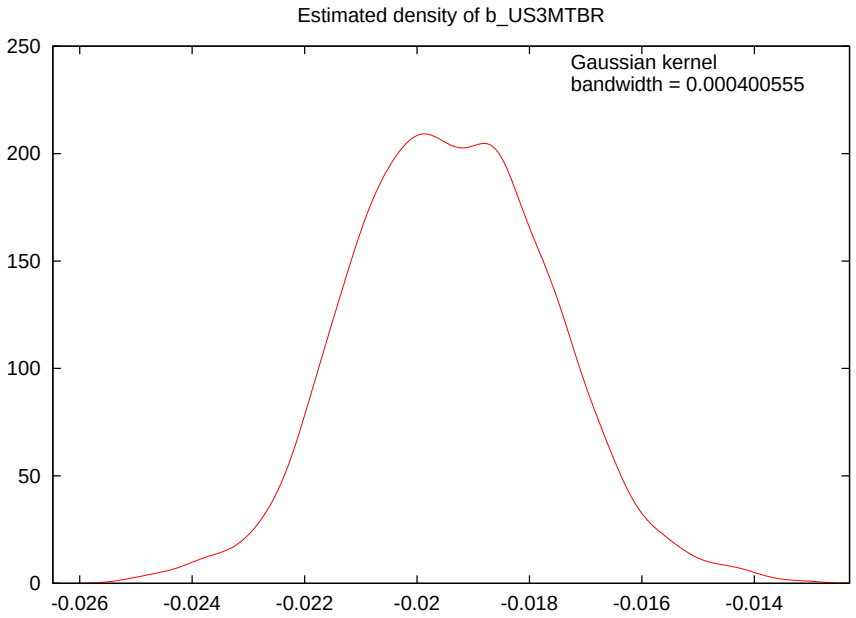

14
12
10
8
6
4
2
0
-2
YYGIPD
YYGCPI

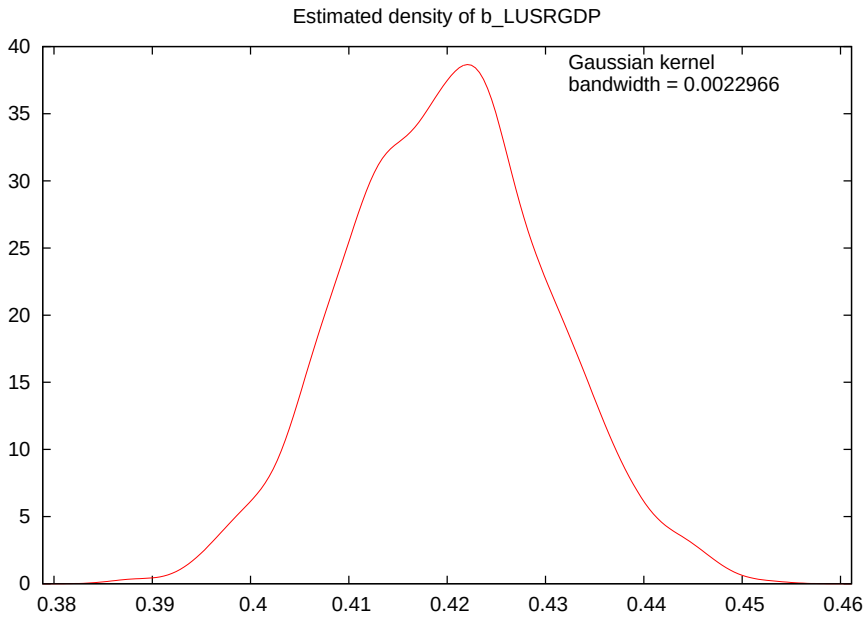

RESIDS
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
1960
1970
1980
1990
2000
2010



Finally, we need to produce bootstrapped values for the regression coefficients. To do this, we must first run the script, and then run the main regression again by clicking in the data page on `Model`, then `Ordinary Least Squares`, and selecting the appropriate variables in the window that results. After clicking on `OK` to run the regression a window will appear containing the regression results alone. Click on `Analysis` along the top of that window and then on `Bootstrap` and select the options `Confidence interval`, `Resample residuals` and `Save bootstrap data to file`. Then, after selecting the coefficient you want bootstrapped estimates of, click on `OK`. A window will appear with print delineating a 95% confidence interval for the coefficient, on top of which will be pasted another window asking you for the file name in which to save the bootstrap results. After choosing a directory and file name for the bootstrap results for that coefficient, leaving off the suffix because `Gretl` will add `.gdt`, the top window will disappear and you can make note of or cut and paste the confidence interval material somewhere. Now go back to the regression model window and proceed to obtain bootstrap results for the coefficient of the next variable, following the same procedure as in the case of the first coefficient and saving its bootstrap results in a different file to which `Gretl` will again attach the suffix `.gdt`. Repeat this process for all remaining coefficients you want obtain bootstrap estimates of. The final step

is to restart `Gretl` and access one of the bootstrap coefficient files. In the data window that appears, select `File` and then `Append data` and sequentially append the other bootstrap coefficient files to the first one. Then save all the bootstrapped coefficients in the resulting file in standard format under a more general name—here, I used `bootres.gdt`. The individual coefficient files can then be deleted, and you can load `bootres.gdt` and save summary statistics and plot and save kernel density estimates of the bootstrapped coefficients. Plots of the kernel density estimates of the bootstrapped treasury bill and log real GDP coefficients are presented below.





We end this section with some guidelines to an alternative way of using the `Gretl` statisical program by working with session files. Exit `Gretl` and then load the program again and access the minimal `Gretl` data file you worked with in the script approach to analysis above. After highlighting the two price level series `USCPI` and `USIPD` by clicking on them while holding down the `Ctrl` key, go to `Add` and then to `Lags of selected variables` and on the prompt screen choose the number of lags to equal 4 and then click `OK`. Plus signs will appear to the left of the numbers assigned to the variables in the data window. Clicking on those signs will cause the lagged series for those variables to be exposed.

Now click on `File`, then `Session files` and then `Save session`. At the prompt, type in the directory path and add a file name with no suffix. I suggest the name `statcomp` to which the program will automatically add the suffix `.gretl`. Now click again on `File` and then `Session files` and load the session file you just saved. A new data window will appear along with an `icon view` window which will display a group of icons that you can click on. You can edit your data by clicking on the `Data set` icon. You can add notes by clicking on the `Notes` icon. Clicking on the `Summary` icon will provide you with summary statistics for all variables in your data set. And clicking

on the Correlations icon will result in the calculation and presentation of cross-correlations between all the series in the data set.

To create the two year-over-year inflation rate series, click on Add in the data window and then on Define new variable. A window will appear in which you can write the formula for new data series you want to create. Simply type in the following formula, which you should have seen before,

```
YYGIPD = 100*(USIPD - USIPD_4)/USIPD_4
```

and click OK. Then create in the same fashion the year-over-year growth rate of the consumer price index.

```
YYGCPI = 100*(USCPI - USCPI_4)/USCPI_4
```

Now back up your work by clicking on File, then Session file and then Save session.

You can now plot the price level and inflation rate series by clicking on View, then Graph specified variables and then Time series plot. A window will appear in which you can select the variables to appear in the plot. After following the prompts a plot will appear. Edit it if you wish by clicking on the right mouse button and choosing Edit. Then click on the right mouse button again and chose Save to session as an icon and a graph icon will appear in the icon window. By clicking on that icon you can view the graph. A box plot of the two inflation rate series can be created and saved as an icon by following the same procedure.

In preparation for running the demand for money regression, use the Define new variable procedure and enter the following formulae.

```
USRGDP = 100*USGDP/USIPD
USRM1 = 100*USM1/USIPD
```

Now highlight these two new series, which will have appeared in the data window and click on Add and then Logs of selected variables and new series representing the logarithms of the two series will appear with the names l_USRM1 and l_USRGDP.

You are now ready to run the demand for money regression. In the main data window, click on Model and then on Ordinary Least Squares and a window will appear in which you can select the dependent variable and the

independent variables. After clicking on `OK` a model window will appear containing the regression results. Click on `File` and then `Save to session as an icon` and the window will be copied as an icon to the `icon view` window. To graph the residuals and the actual and fitted values, simply click on `Graphs` along the top of the model window and follow the prompts. Save these graphs as icons in the usual fashion. You should also click on `Graphs` and then `Residual correlogram` to produce a plot of the autocorrelations and partial autocorrelations of the regression residuals and a window containing the numerical measures of these autocorrelations. Again, you should save both the table and the graph as session icons.

Further details about your regression can be obtained by clicking on `Tests` along the top of the regression model window. Select `Heteroskedasticity` and then `Breusch-Pagan`. A new window will appear with the details, which you can save as an icon, and a summary of the test results will be added to the bottom of the material in the regression model window. Now choose `Test` and `autocorrelation` and, after indicating the maximum number of lags, you will be presented with another window containing the results of the residuals regression and the three alternative serial correlation tests noted before. Only the first of these will be summarized in the regression model window but the detailed results can and should be saved as an icon. Another interesting test is the test for normality of the residuals which, again produces a plot and a window of detailed results, both of which can be saved as icons.

Now click on `Analysis` along the top of the regression model window. By clicking on `Confidence intervals for the coefficients` you will be presented with a little window containing material that can be copied and pasted in the `Notes` icon. Similarily, you can extract and save the `Coefficient covariance matrix`. And, finally, you can click on `Bootstrap`. Here you will end up following the same procedure as before. The data will be collected separately for each coefficient in turn. The confidence interval is presented in a window from which the details can be copied to the `Notes` icon. Kernel density graphs of each coefficient's bootstrap estimates can be obtained and copied as icons, after editing to provide clear identification of the coefficient involved. And the bootstrapped data for each coefficient can be saved to separate data files which can later be merged into a single file.

As you work along, the session should be frequently saved so that a correct record can be kept of everything. Also, you can copy as many as six of the graphs (by drag and drop) to the `Graph Page` icon and later saved or printed as a PDF file.

You can find the `Gretl` code for the material in this section in the file `gretsect.inp` and the corresponding output in the file `gretsect.out`. The basic data file on which the analysis is based is `statcomp.gdt`. And a session file that reflects the discussion immediately above is `statcomp.gretl`.

Exercises

1. Write a `Gretl` batch file to do the same analysis of the demand for U.S. M2 as was done for U.S. M1 in the analysis above.

2. Read the section in the *Gretl User's Guide* dealing with matrix manipulation and then extend your batch file in Question 1 above to create the relevant Y-vector and X-matrix for your basic demand for money regression. Use this vector and matrix to calculate by matrix manipulation the regression coefficients, the standard error of the regression, the R^2, the coefficient standard errors and t-ratios. Compare your results with those produced by the basic demand for money regression you ran in Question 1.

3. Load the data file `statcomp.gdt` into `Gretl` and then set up the variables and run the regression in question 1 above and save the result as a session file. Then graph the actual and fitted and residuals and the autocorrelations of the residuals and test the residuals for normality and graph the results, saving all graphs and tests as icons. Test the residuals for autocorrelation. Do a bootstrap test of the coefficient of the T-Bill Rate and graph the probability density function. Then put all the graphs on a single graph page. Calculate confidence intervals for the coefficients based first on the regression results and then on bootstraps of the coefficients, saving all these confidence intervals in the session notes. Make sure that everything above is represented directly or indirectly in icon form.