Ects

Software for Econometrics
Version 3

Russell Davidson

April 2004
Revised, September, 2005

To econometrics students everywhere

Table of Contents

Table of Contents **iv**

Introduction **1**

1 New Features **3**

1 Introduction 3
2 Graphics 4
3 Automatic Differentiation 15
4 Numerical Integration 19

2 Nonlinear Estimation **23**

1 Introduction 23
2 Nonlinear Least Squares 23
3 Nonlinear Estimation with Instrumental Variables 26
4 Maximum Likelihood 31
5 The Generalised Method of Moments 40
6 Procedures 46

3 Simulation **55**

1 Recursive Simulation 55
2 ARMA Processes 57
3 ARMAX and VAR Processes 61
4 ARCH and GARCH Processes 68
5 Resampling and the Bootstrap 72

References **84**

Introduction

This is the English translation of part of the second volume of documentation for ***Ects***. With any luck, the whole volume will be available in translation in the near future. The second volume appeared in French in March 1999, and was intended to serve as a supplement to the first volume, which had appeared in 1993 along with version 2 of the software. This translation, too, is a supplement, not a complete guide in its own right. The purpose of this second volume is to describe the new features available in version 3, and, more specifically, in version 3.3. It should be used in conjunction with the earlier volume, which appeared in English translation in March 2004.

For the user's point of view, the most interesting of the new features are to do with graphics, nonlinear estimation, and automatic differentiation of functions. In addition, the range of functions known to ***Ects*** has been considerably extended, and a numerical integration facility has been introduced.

The first two versions of ***Ects*** showed themselves useful not just as a teaching tool, but also for a good many practical applications. This version has behind it another decade of experience, which has led to its being more flexible, and, I hope, more useful still, especially for setting up simulation experiments, which are required more and more by modern statistical and econometrics procedures and practice.

In 2003, the old DOS version of ***Ects*** was abandoned. It had become harder and harder to compile, and did not always work as well as one would have wished in a Windows environment. Thanks to Pierre-Henri Bono, a version of the software that not only runs under Windows, but runs fast and well, has been made available, and it is recommended for people foolish enough not to have broken free from that operating system.

The present version of ***Ects***, and the subsequent version that is currently in preparation, were developed using the Linux operating system. I recommend Linux unreservedly to all those for whom it is important to have a computing environment that is at once user friendly, efficient, and suitable for serious scientific work. Linux is available at no cost, and can be installed on most types of hardware at present on the market. Upgrades are free, and the range of available software is enormous. For scientific work, but not only for that, this software is virtually always at least as good as, and usually superior to, the competition from the Redmond monopolist.

Earlier versions of of ***Ects*** made considerable use of the programs in the well-known *Numerical Recipes* of Press, Flannery, Teukolsky, and Vetterling (1986). Most unfortunately, the later edition of this work, Press *et al*

(1992), imposes undesirable restrictions on the use of the programs contained in that edition. Whether this is due to the authors' own wish or to pressure from their publisher, I don't know, but it certainly limits the usefulness of their book. In the present version of ***Ects***, none of their programs is used. Fortunately, one can these days find all sorts of useful programs and algorithms on the Web. In particular, I have taken considerable advantage of the functions in the Cephes Mathematical Function Library, a library of functions written in C. The author of the library is Stephen L. Moshier (`moshier@world.std.com`), who has also written a related textbook, Moshier (1989). Although he reserves the author's rights to the library, he allows free use of his programs; for this many thanks. The algorithm used for the Singular Value Decomposition (SVD), used in least squares estimation, was written by Brian Collett (`bcollett@hamilton.edu`). The code, in C, is based on an algorithm written in Algol and published in Golub and Reinsch (1970).

The first version of ***Ects*** was written in C, the second partially rewritten in C++, and version 3 completely rewritten in C++, except for a few purely numerical routines, for which the C version is totally adequate. The stated aim of C++ was to be "a better C" – see Stroustrup (1991). I can confirm that programming in C++, while it requires a period of relearning and intellectual reorientation, is a much more pleasant activity than programming in C. In addition, C++ programs can be much more readable than C programs. One problem I had was that, back in 1994 when the development of version 3 began, the development of the C++ standard library, now an integral part of the language, had hardly begun. In early versions, I made substantial use of a "draft" library, published in Plauger (1995). This was indeed just a draft; hardly had the book been published when further sweeping changes were decreed by the C++ committee. Now the problem has vanished, and the standard library is widely available.

I also wish to express my gratitude to all the people who, under the aegis of the Free Software Foundation, developed the GNU C++ compiler. It is hard sometimes to realise just how much GNU software has transformed the world of scientific computing. Linux itself depends completely on the immense collection of GNU software, all freely available. ***Ects*** would not exist without the GNU compiler.

This slim volume is not an econometrics textbook. For that reason, I make frequent reference to a real textbook, namely Davidson and MacKinnon (1993), to which I refer simply as DM. Only once or twice in this English translation have I taken the opportunity to mention the new book, Davidson and MacKinnon (2004). It is also occasionally necessary to refer to the first volume of this manual, written for version 2: the reference used is Man2.

Chapter 1
New Features

1. Introduction

In this first chapter, we will see how to use some of the new features available in version 3 of ***Ects***. Although nonlinear estimation is not treated until Chapter 2, we will talk here about a new tool that vastly improves the setting up of such estimations, namely, automatic differentiation.

Among the errors in writing programs for ***Ects***, the most frequent were, without doubt, errors in the specification of the derivatives of the functions. In order to run the nonlinear commands `nls`, `ml`, and `gmm`, it is necessary to specify the derivatives of a regression function, or of a loglikelihood, or of the objective function, with respect to the parameters we are trying to estimate. Now, we can leave this task to ***Ects***.

Most econometrics software packages available on the market are praised because of their capabilities for graphics. Up until now, ***Ects*** lacked that functionality. In spite of requests, frequent and insistent, I hesitated for a long time on how best to create a graphical interface for ***Ects***. One of the difficulties was simply that there were too many possibilities. Finally, it came to me that, almost every day, I was using a freeware package named `gnuplot` for my own graphical needs, whether it was for displaying on a screen or for printing. The authors of this software had indeed solved all the problems I was facing. The development of this software was a joint effort, with the help of numerous programmers: those who own the rights to the current version are Thomas Williams and Colin Kelley.

Often in the calculation of a set of functions or derivatives, we have to reevaluate the same expression many times. This is particularly true with the derivatives we have to give to `nls` and its cousins. Redoing the same thing over and over again is always tiresome and not very practical, but in the case of a nonlinear estimation, a slow procedure by its very nature, there is great profit to be had by eliminating needless repetitive operations. In order to ease this elimination, ***Ects*** now allows us to define **procedures**. These procedures constitute a block of operations that are used to calculate, once and for all, all that we need. The calculation will only be started if the arguments of the procedure change. Later, we will see how to use the `procedure` command, which lets us accelerate some estimations.

For graphics, we need the `gnuplot` program. This software is available freely and can be compiled for at least as many architectures as ***Ects***. The executable file for `gnuplot` must be located in the access path of the user. Further, we need a directory named `tmp` in the root directory. Unix systems, including Linux, are installed with such directories, configured so as to give the user permission to create files. Under other operating systems, this directory must be created, if it doesn't exist, such that everyone can write (in other words, create files) in this directory.

It is impossible to prevent a program like ***Ects*** from freezing or crashing if the data fed to it are weird enough to give rise to exceptions at the level of the floating point processor. However, this should happen only relatively rarely. On the other hand, I should be informed about anything that gives rise to a segmentation fault so that I may correct the bug which created it. Even if such mistakes are not very dangerous under Unix, other, less powerful, operating systems must be restarted in certain circumstances following such bugs.

2. Graphics

Before creating graphics, it would be prudent to verify the configuration of `gnuplot`. Type the following command:

```
gnuplot
```

If the system responds that it can't find it, we know that the `gnuplot` executable file is not in the access path. But if it is found, `gnuplot` will display a welcome message, followed by a line that will look like the following:

```
Terminal type set to 'x11'
```

This means that the graphical mode of `gnuplot` is `x11`. Under Unix, graphics are handled by the X Window system, of which the current version (in 1998) is 11, explaining the `x11`. Under Windows, the graphical mode will be different; it will most often be SVGA. `gnuplot` is normally able to detect automatically the graphical mode that is the best suited, but in case of an error, we can enter the value needed in the environment variable `GNUTERM`. To find out all the available graphical modes recognized by `gnuplot`, type

```
set term
```

after starting `gnuplot`. It will display a long list, and you will see the names of a few printers. We will later see how to print your graphics.

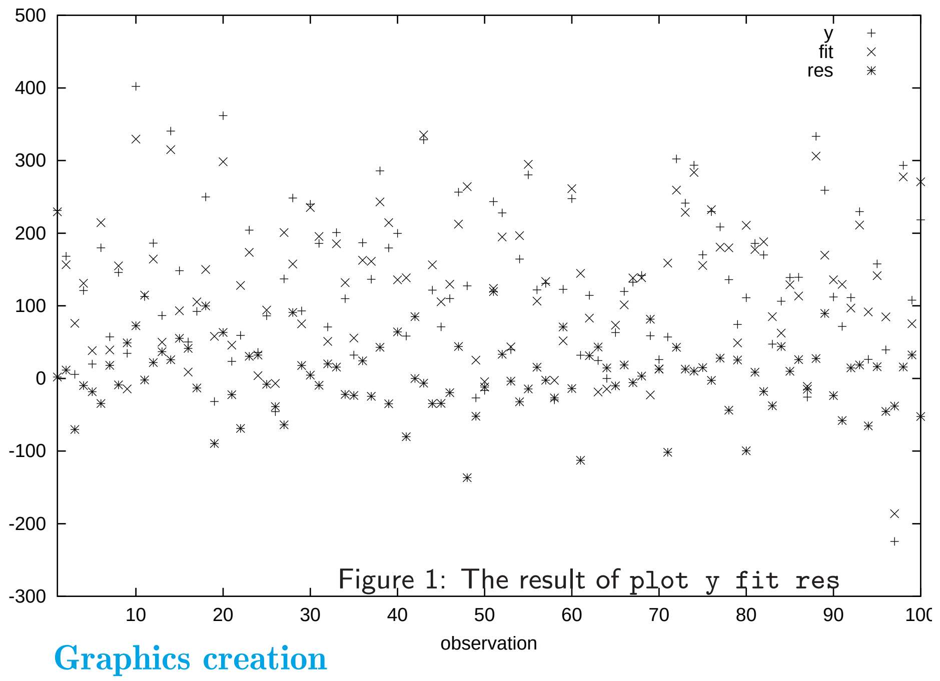

Figure 1: The result of `plot y fit res`

Graphics creation

To create graphics, we use the `plot` command. The syntax of this command is very flexible, and lets us create and display one or many graphics on screen. Now consider the command file `testplot.ect`, which contains

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
%set linestyle = 1
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
quit
```

We will use our favorite data file `ols.dat`. The `plot` command will display 6 graphics, one by one. When we want to switch to the next graphic, we use the Return or Enter keys.[1].

The first graphic would be given by

```
plot y fit res
```

which gives three graphs, `y`, `fit` and `res`. After each `ols` command, ***Ects*** puts the fitted values of the regression in the variable `fit`, and the residuals of the regression in the variable `res`. For each graph, we have on the X-axis the subscript of the observation. In fact, the word observation is located just below the horizontal axis, and we can see that the indices vary from 1 to 100,

[1] Under Windows, we may have to push these keys twice.

corresponding to the sample size. On the Y-axis, we have the value of the variable as a function of the observation. Different lines are given different colors and, at top right, are displayed the names of the variables, with the corresponding color. Without color, but with different symbols instead, the graph displayed is similar to the one we see in Figure 1.

Note that, above the `plot` command, there is a `set` command, preceded by the `%` symbol. The result of `%` is the same as that of the `rem` command. This means that whatever follows this symbol will not be read by ***Ects***. We can obtain the same result in a third way, namely, by putting the `#` symbol as first letter of a command line.

* * * *

By "first letter", we mean the first letter other than a blank space or a horizontal tabulation.

* * * *

The `%` and `#` symbols are used in this way by many packages, and I was asked to implement the same feature in ***Ects***.

What happens if we erase the `%`? As long as the `linestyle` variable is not defined, or its value is zero, `gnuplot` puts a big dot for each observation. But if the value of `linestyle` is different from zero, straight lines are graphed between observations. If a variable is defined smoothly from one observation to the next, the lines will be prettier than the dots. However, if a variable changes in a very irregular manner, the dots might better show this fact.

Exercises:

Generate four or five variables with the standard normal distribution, using the `random` function:

```
gen y1 = random()
gen y2 = random()
⋮
```

and display them with the `linestyle` variable equal to 0 and 1. In this way, you will see the typical behavior of **white noise**.

Repeat the same exercise using other variables that you generate in a deterministic manner. Try for instance

```
sample 1 180
gen y = sin(time(0)*PI/180)
plot y
```

Note that the variable `PI` is available automatically in ***Ects*** version 3.3. Its value is, of course, $\pi = 3.1415926535897932$.
Change the sample size using

```
sample 90 180
```

and redo

```
plot y
```

You will then see that the graph is done only for the current sample.

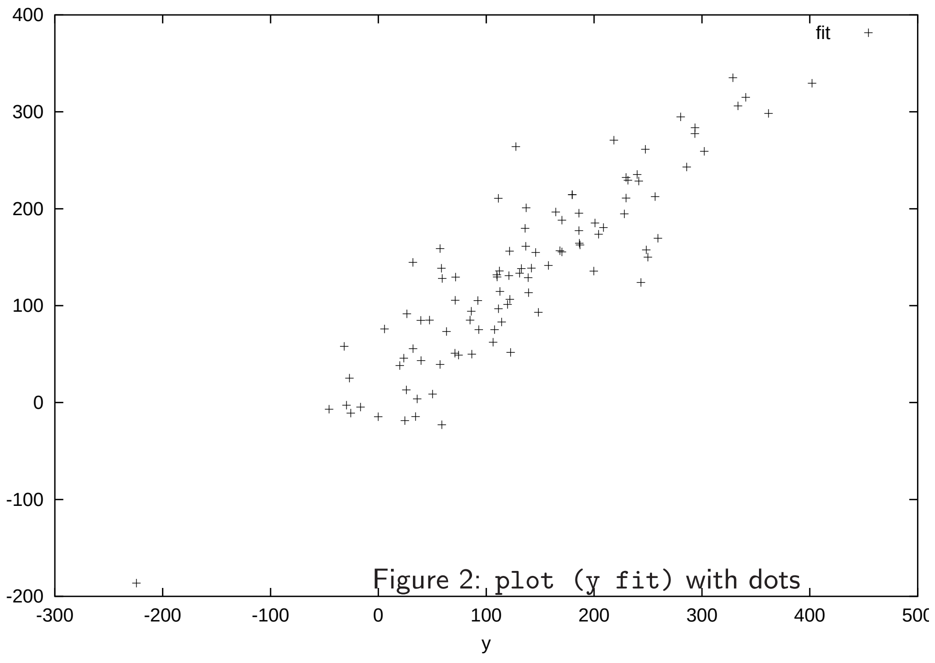

Figure 2: `plot (y fit)` with dots

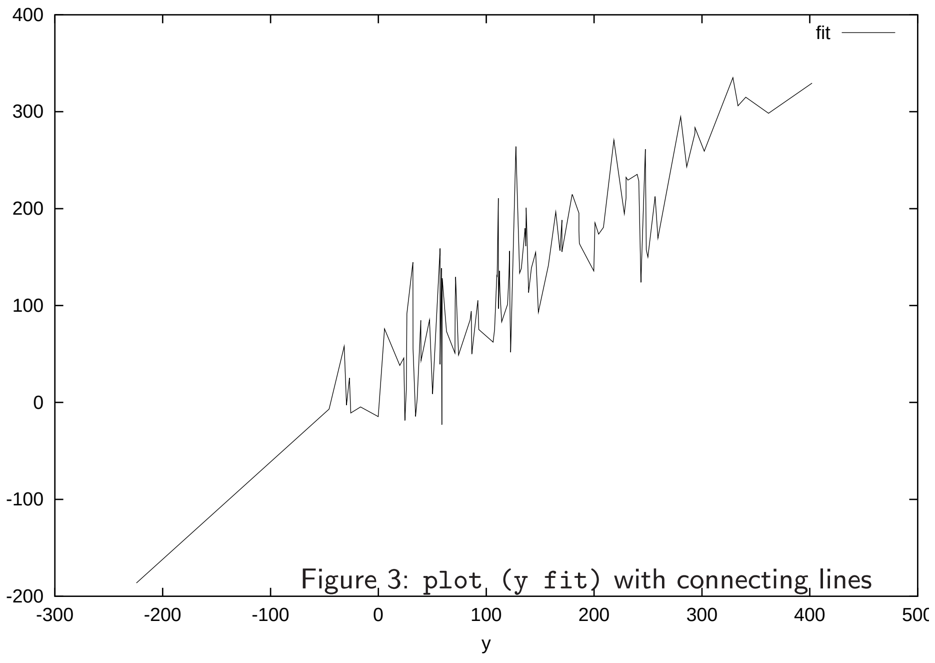

Figure 3: `plot (y fit)` with connecting lines

The second graph produced by the `plot` command of `testplot.ect` would be given all by itself by the command

```
plot (y fit)
```

The parentheses mean that the graph will no longer have observation on the X-axis, but instead, the first variable found inside the parentheses, here `y`. On the Y-axis, we have `fit`, as a function of `y`. The scale of variation of both variables is automatically calculated by `gnuplot`. We see the result of the command in Figure 2 with dots, and with lines in Figure 3.

The third graph is given by itself by the command

```
plot (y res)
```

and is constructed in just the same way as the previous one.

The fourth graph, generated by

```
plot (y fit res)
```

demonstrates the ability to have many graphs on the same plot, as when we had observation on the X-axis. The results of this command are shown in Figure 4.

There are still two more graphs produced by the command in `testplot.ect`. They would be generated singly by the commands

```
plot y fit
plot y res
```

For both of them, observation is on the X-axis, and on the Y-axis, we have `y` and either `fit` or `res`.
Why create many graphs using only one `plot` command, instead of using many `plot` commands and one graph per command? The second choice is entirely possible, but, in this case, ***Ects*** will call `gnuplot` as many times as it finds `plot` commands. This need not be bothersome, but we may notice that the graphs are displayed slower.

To separate the different graphs in a `plot` command, we use a comma: `,` . However, if we close a parenthesis, everything that follows will be part of another graph and, consequently, we can, optionally, get rid of the comma. Thus, in the command:

```
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
```

we have put a comma after `(y fit)`, but not after `(y res)` or `(y res fit)`. If we delete the comma between `y fit` and `y res` at the end of the command, we will obtain only one graph instead of two, and, in that graph, the variable `y` will be plotted twice. If we insert the comma inside the set of variables in the parenthesis, the consequences can be unpredictable and, most likely, different from what you would expect! Even if we don't need the comma after *closing* a parenthesis, it is compulsory before opening a parenthesis. If we forget, doing for instance

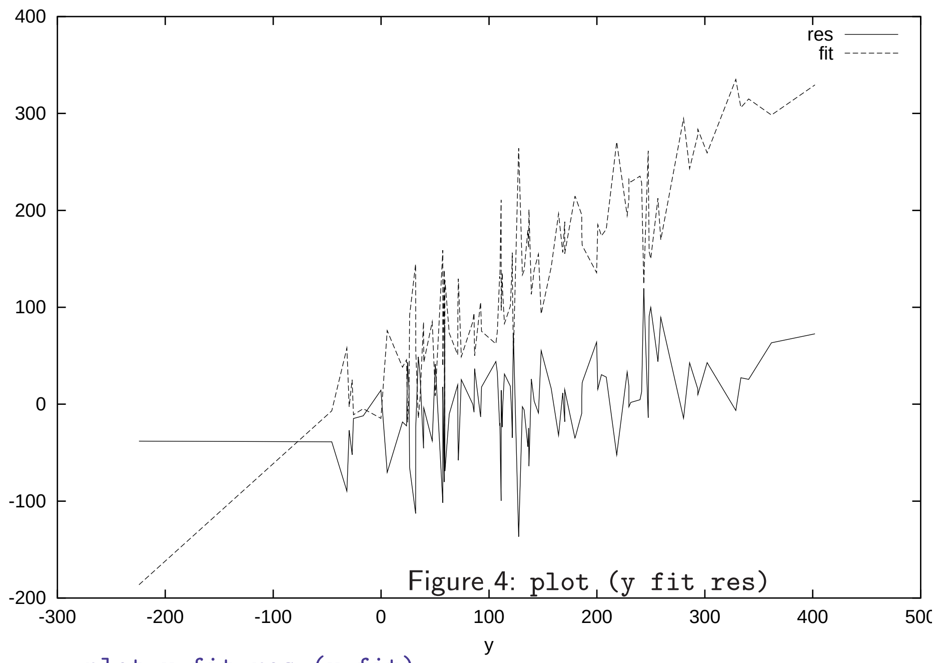

Figure 4: `plot (y fit res)`

```
plot y fit res (y fit)
rem ERROR!!
```

Ects will be troubled by the presence of `(y` which it tries to interpret as the name of a variable.

Up until now, all the arguments submitted to `plot` were vectors, each representing only one variable. We can also have matrices with more than one column as arguments. For instance, if we do:

```
gen yfr = colcat(y,fit,res)
plot yfr
plot (yfr)
```

it is the same as doing

```
plot y fit res
plot (y fit res)
```

except for the variable names displayed by `gnuplot`. Instead of the explicit names, `y`, `fit`, `res`, we will have the name of the matrix, followed by the column subscript: `yfr1`, `yfr2`, `yfr3`. We can do:

```
plot yfr, (yfr)
```

to avoid an interruption between the display of both plots. The rule is simply that any matrix argument is decomposed into columns before being used by `gnuplot`.

EXERCISES:

Once again, use the data in the `ols.dat` file and program a loop which will help run the estimation

```
ols y c x1 x2 x3
```

for all samples defined by

```
sample 1 n
```

with `n` $= 10, \ldots, 100$. As the loop runs, save the parameter estimates and the estimates of $\hat{\sigma}^2$ in five variables `a`, `b1`, `b2`, `b3`, and `s2`. Then create a graph in which you plot these estimates as a function of the sample size, $n = 10, \ldots, 100$.

Printing Graphs

The graphical features of ***Ects*** were conceived mainly for displaying on a computer screen. If we wish to print graphs, `gnuplot` takes care of this task. For various operating systems, it is more or less easy to give `gnuplot` the necessary information.

When executing the `plot` command, ***Ects*** creates files and stores them in the `tmp` directory. The first of these files is a command file for `gnuplot`'s use. This file bears the name of `gnuplot.gnu`. Then, depending on the number of graphs asked for, it creates data files, which bear the names `gnuplot.`n, for $n = 0, 1, \ldots, m-1$, where m is the number of graphs that are needed. If we work under a multitasking operating system, we can launch ***Ects***, and, during the display of the graph on the screen, we can go into the `tmp` directory to find and save the pertinent files by copying them somewhere else. It is important to do this *during* the execution because the files will be erased after quitting ***Ects***. Alternatively, if we do

```
set savegnu = 1
```

before launching the `plot` command, the files will not be erased and, we will be able to find them in the `tmp` directory after quitting ***Ects***.

* * * *

> But watch out! We will be able to find only the files associated to the *last* graph. Any files that were there before and bearing the same name as the last one used will be overwritten when the latter is created.

* * * *

With an operating system that is not multitasking,[2] unless we use the `savegnu` variable, we have to create the necessary files ourselves. Even with modern operating systems, it is useful to know how to proceed. Let us look here at the command file `gnuplot.gnu` created by the

```
plot y fit res, (y fit), (y res) (y res fit) y fit, y res
```

command we looked at earlier. For the first graph, we have

[2] What I had in mind in writing this is the now largely vanished DOS.

```
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.0" using 1:2 title 'y',\
"/tmp/gnuplot.0" using 1:3 title 'fit',\
"/tmp/gnuplot.0" using 1:4 title 'res'
pause -1
```

The `xrange` defines the limits of the X-axis, here, the artificial variable observation. In fact, the sample is defined from 1 to 100. The `xlabel` is the label of the horizontal axis. The next command requires particular attention. The syntax is the following:

`plot "`⟨*file name*⟩`" using` n`:`m `title '`⟨*var*⟩`'`

where, instead of ⟨*file name*⟩, we insert the file name that contains the data to be displayed. This file, which in our case bears the name of `/tmp/gnuplot.0`, contains many columns of data. Each line of the file corresponds to a point in the graph. The subscripts n and m are the numbers of columns to be used for the horizontal coordinates n and the vertical coordinates m which make up our graph. Finally, *var* is the name or the label associated with the plot.

As we can notice, the necessary information for plotting the `y` variable is located in columns 1 and 2 of the file. The first column gives the variable observation, which is the horizontal coordinate of all the plots. The second column corresponds to the variable `y`. Similarly, the plot of the `fit` variable is constructed based on columns 1 and 3 of the file, and the one for `res` on columns 1 and 4.

If we cannot or do not wish to benefit from the automatic construction of a data file by ***Ects***, we will have to construct the file `/tmp/gnuplot.0` directly. This is not at all difficult. The variable observation can be directly generated by

```
gen observation = time(0)
```

and the data file by

```
write gnuplot.0 observation y fit res
```

Note that, if we create the data file directly, it is no longer necessary to put it in the `tmp` directory.

The line

```
pause -1
```

is a signal to `gnuplot`, which makes it stop after displaying the graph, or until we type Return or Enter. For obvious reasons, this line is unnecessary if we wish to print a graph.

In the file `gnuplot.gnu` created by ***Ects***, the \ symbol is not used, because all the `plot` command, from the word `plot` up until the word `'res'`, takes up only one line. It would be impossible to print such a line in the user's guide, and it is often desirable in practice to avoid lines that are too long.

gnuplot enables us to spread the command on to many lines if we insert a \ at the end of each line, except the last one of the command, and that is what we have done here so that the program is clearer to read.

* * * *

The same method can be used with ***Ects*** itself: see section 2.5

* * * *

The last two graphs, those that can generated by

```
plot y fit
plot y res
```

are created by giving gnuplot the following commands:

```
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.4" using 1:2 title 'y',\
"/tmp/gnuplot.4" using 1:3 title 'fit'
pause -1
set xrange [1 :  100]
set xlabel "observation"
plot "/tmp/gnuplot.5" using 1:2 title 'y',\
"/tmp/gnuplot.5" using 1:3 title 'res'
pause -1
```

Here, the only element to note is that the successive graphs use different data files. Even if the variables plotted in these graphs are the same as those in the first graph, ***Ects*** creates new files.

Let us now look at the other gnuplot commands that generate the other graphs, those where we have something other than the variable observation on the X-axis.

```
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.1" using 1:2 title 'fit'
pause -1
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.2" using 1:2 title 'res'
pause -1
set autoscale
set xlabel "y"
plot "/tmp/gnuplot.3" using 1:2 title 'res',\
"/tmp/gnuplot.3" using 1:3 title 'fit'
pause -1
```

The

```
set autoscale
```

command asks gnuplot to determine by itself the extreme values of the variables. Thus, it replaces the command

```
set xrange ...
```

that we used for the graphs above. The `xlabel` is no longer observation, but rather `y`, because `y` is the variable of the abscissa.

In order to generate the files `gnuplot.1,2,3` directly, we have to do more manipulation, which is not necessary when the variable on the X-axis is observation. We know that every line of the data file corresponds to a point on the graph. `gnuplot` constructs its graphs in the order of the lines of the file. Consequently, if the first column of the data file is not *ordered*, in decreasing or increasing order, the points of the graph will be constructed in an irregular manner. If every point is represented by a point, this has no importance, but if the successive points are connected by straight lines, those straight lines will be an excellent representation of Chaos.

To generate the file `gnuplot.1` directly, we then proceed as follows:

```
gen yf = colcat(y,fit)
gen yf = sort(yf)
write gnuplot.1 yf
```

The command `sort` is used to sort the rows of the `yf` matrix by ascending order of the elements of the first column, in other words, the elements of the X-axis variable, `y`.

Let us now summarize the procedure which enables us to print graphs. The first point is that it is always preferable to print only one graph at a time. Otherwise, we may encounter pagination difficulties that will not be easy to overcome. Here, we limit ourselves to printing the graphs created by the two commands

```
plot y fit res
plot (y fit res)
```

These graphs appear in Figure 1 and 4. First, to create the data file, we do

```
gen observation = time(0)
gen data = colcat(observation,y,fit,res)
write gnuplot.0 data
gen data = colcat(y,fit,res)
gen data = sort(data)
write gnuplot.1 data
```

Then, we use any text editor to create the file `gnuplot.gnu`, of which the content will be similar to

```
set xrange [1 :  100]
set xlabel "observation"
set term postscript eps
set out "fig1.ps"
plot "gnuplot.0" using 1:2 title 'y',\
"gnuplot.0" using 1:3 title 'fit',\
"gnuplot.0" using 1:4 title 'res'
```

Note the two commands

```
set term postscript eps
set out "fig1.ps"
```

The first one is used to select the format of the file which will subsequently be sent to the printer. The choice I made here is the one best suited for printing a figure in this user's guide, namely, encapsulated Postcript. Another choice could be

```
set term hpljii 300
```

which means the format required by the printer HP LaserJet II, with a resolution of 300 dots per inch, or again

```
set term latex
```

which would yield a file in LaTeX format. These are only two examples: we could find hundreds of different possibilities in the documentation supplied with `gnuplot`.

The command

```
set out "fig1.ps"
```

asks `gnuplot` to create a file `fig1.ps` and to use it as an output file. As I had asked for a PostScript file, I give a name of such a file, with the extension `.ps`.

Then, we launch `gnuplot` the name of the command file as argument:

```
gnuplot gnuplot.gnu
```

After the execution of this command, we should find the output file in the current directory. For me, this file would bear the name of `fig1.ps`. The last step is, of course, to send this file to the printer. The appropriate way to do this depends on the operating system used. In the present case, it is also necessary to have a PostScript printer, or else to find some other way of printing PostScript files.

The procedure for printing the graph created by

```
plot (y fit res)
```

is very similar. The sorted data already exist in the file `gnuplot.1`. Only what is in `gnuplot.gnu` has to be changed. We have

```
set autoscale
set xlabel "y"
set term postscript eps
set out "fig2.ps"
plot "gnuplot.1" using 1:2 title 'res',\
"gnuplot.1" using 1:3 title 'fit'
```

After `gnuplot` executes these commands, the file `fig2.ps` can be sent to the PostScript printer.

Exercises:

Create a file containing the necessary data in order to create a plot of the function $\sin x$ for $x \in [0, 2\pi]$. If you have access to a printer, write a program for `gnuplot` that will be able to print the graph. Otherwise, modify the program so it will display the graph on the screen.

3. Automatic Differentiation

The data manipulated by ***Ects*** are tables of numbers, which represent scalars, vectors, or matrices. When we calculate a derivative, it is the derivative of a *function*. However, ***Ects*** does not enable the user to represent functions directly. But the **macros** created by the `def` command can be used to represent algebraic expressions that can subsequently be used by ***Ects***. It is on the basis of this feature that the automatic differentiation mechanism of version 3 of ***Ects*** is built.

Let us take a simple example. We know that the partial derivative of the function x^2 is $2x$. We might hope that a construction like

```
diff(x^2,x)
```

using a pseudo-function `diff`, would represent the derivative of x^2 with respect to x. Indeed, if we do

```
set x = 4
set y = diff(x^2,x)
show y
```

the answer ***Ects*** yields is

```
y = 8.000000
```

We find that `y` is twice the value of `x`.

It is crucial to understand that the derivatives calculated by ***Ects*** are obtained by symbolic manipulations. If, for instance, we ran

```
set x = 4
set x2 = x^2
set y = diff(x2,x)
show y
```

the answer would be

```
y = 0.000000
```

because `x2` does not depend on `x`. We need the *expression* `x^2`, given in terms of `x`, so that the derivative can be something other than zero.

We could use any function of x instead of x^2. Some examples: the program

```
set y = diff(x^4-3*x^3+2*x^2+3*x-4,x)
```

```
show y
set y = diff(sin(x),x)
show y
set y = diff((sin(x))^2+(cos(x))^2,x)
show y
```

will give the successive answers:

```
y = 131.000000
y = -0.653644
y = 0.000000
```

Let us calculate:

$$\frac{d}{dx}(x^4 - 3x^3 + 2x^2 + 3x - 4) = 4x^3 - 9x^2 + 4x + 3.$$

This expression, evaluated at $x = 4$, is equal to $4.64 - 9.16 + 4.4 + 3 = 131$, which is the first answer. If we do

```
set y = cos(x)
show y
```

we can check the second answer: we do indeed obtain -0.653644. The last answer is the consequence of the trigonometric identity

$$\sin^2 x + \cos^2 x = 1.$$

The derivative of the constant 1 is equal to 0, in accordance with the answer given by ***Ects***.

In all these calculations, ***Ects*** manipulates the symbols of the expression to be differentiated in order to find a symbolic representation of the derivative, and only after doing this does it evaluate this symbolic representation according to the rules of the current command. In all examples considered until now, this command was `set`. We could have `gen` as well. For instance, if we run

```
sample 1 100
gen x = 0.1*time(0)
gen y = diff(chisq(x,2),x)
plot (x y)
```

a plot of the density of the chi-squared (χ^2) distribution with 2 degrees of freedom will be displayed on the interval $]0, 10]$. The procedure is the following. First, ***Ects*** finds that the derivative of the $\chi^2(2)$ distribution function can be written in terms of the **incomplete gamma function**.

* * * *

See section 7.1 of ***Ects*** Version 2 user's guide, for more information concerning this function. We will talk about it later in section 4.1, when we look at new functions recognized by ***Ects***.

* * * *

A symbolic representation of this function is created. Then this representation is passed on to the `gen` command, which generates a `y` vector, the components of which are the values of the $\chi^2(2)$ density (the derivative of the distribution function) evaluated at the corresponding elements of `x`.

We can also use the `diff` function in a `mat` command. Consider the following program:

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
set a = coef(1)
set b1 = coef(2)
set b2 = coef(3)
set b3 = coef(4)
def residual = y - a*c - b1*x1 - b2*x2 - b3*x3
def criterion = residual'*residual
mat db1 = diff(criterion,b1)
show db1
quit
```

We again use the data from the `ols.dat` file. After running the regression, we save the estimated parameters and we define a macro named `residual`, which is the algebraic expression of the residuals in the regression. Note that a *macro* is an *expression* for ***Ects***, rather than a numerical matrix. A second macro, `criterion`, is used to define the criterion function, namely the sum of squared residuals, of which the minimization yields the least squares estimates. The expression

```
diff(criterion,b1)
```

produces a representation of the derivative of the criterion function with respect to one of the parameters, namely `b1`. If we wrote out this representation explicitly, we would have something like

```
-x1'*(y-a*c-b1*x1-b2*x2-b3*x3)-(y-a*c-b1*x1-b2*x2-b3*x3)'*x1
```

If this expression is evaluated by a `mat` command, the result is a 1×1 matrix, that is, a scalar. The value of the scalar should be zero, thanks to the first-order conditions of the minimization. If we ran the above program, we would see that `db1` is indeed equal to 0.

A necessary remark: In the definition of the `residual` macro, we explicitly made use of the constant vector `c`. Under `gen`, we could have simply written

```
y - a - b1*x1 - b2*x2 - b3*x3
```

without the vector `c`. But under `mat`, there would have been a problem due to the mix of vectors with the scalar `a`. In the automatic differentiation process, the derivative of `a` with respect to `a` is 1, a scalar, but the derivative of `a*c` is `c`, a vector, in accordance to the rules of matrix manipulation.

Exercises:

Run the above program in order to check that `db1` $= 0$. Evaluate the `residual` macro by a `gen` command and check that the result is identical to the vector `res` created by the `ols` command. Also verify that this result is not changed if the `residual` macro is evaluated by `mat`. Then, change the definition of the `residual` macro by imposing the constraint that `b3` $=$ `b1`. Evaluate the `criterion` macro (using `mat`): the value will be greater than the `ssr` variable, because this variable contains the minimized value of the sum of squared residuals. Finally, evaluate `db1` once again, and check that its value is now different from zero.

There exist two other functions which make use of automatic differentiation. These functions, `grad` and `hess`, are available only with `mat` commands. They are used to calculate, respectively, the **gradient** and the **hessian** of a matrix expression, which must be a scalar, even if it is itself made up of nonscalar elements. In extending the above example, consider the following code:

```
mat gr = grad(criterion,a,b1,b2,b3)
sample 1 4
show gr
mat H = hess(criterion,a,b1,b2,b3)
mat H = 2*H inv
show H XtXinv
```

The `criterion` macro is always the sum of squared residuals, in matrix form. But the value is scalar, because `criterion` defines a scalar product. The command

```
mat gr = grad(criterion,a,b1,b2,b3)
```

asks for the calculation of the 4×1 vector of partial derivatives of the sum of squared residuals function with respect to four variables, `a`, `b1`, `b2`, and `b3`. The following command:

```
mat H = hess(criterion,a,b1,b2,b3)
```

asks for the calculation of the 4×4 matrix of second derivatives of the function with respect to the same variables.

The value of the gradient of the sum of squared residuals in the least squares estimator case, should be zero, because of the first-order conditions for the minimization. If we execute the above commands, we can check that that is the case. As for the hessian, using the usual notation, we have to construct the matrix of derivatives of the gradient

$$-2\boldsymbol{X}^\top(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{\beta}),$$

and this matrix is just $2\boldsymbol{X}^\top\boldsymbol{X}$. Twice the inverse of this matrix is $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$, and the matrices `H` and `XtXinv` are thus identical.

Exercises:

If we differentiate an expression with respect to a variable on which it does not depend, the result must be zero. ***Ects*** allows us to differentiate expressions with respect to variables that do not even exist. Calculate the derivatives of the `residual` and `criterion` macros, with respect to a variable `b4`. In all cases, the result will be a simple scalar, 0. Then, create the variable `b4`, using a `set` command for instance. Check that the results of the differentiation are unchanged. Change the value of `b3`, so that the gradient of `criterion` is no longer zero. After that, calculate the gradient and the hessian of `criterion` with respect to the variables `a`, `b1`, and `b4`. Here, we will obtain a 3×1 vector and a 3×3 matrix, but with zero elements corresponding to the derivatives with respect to `b4`.

We can calculate second, third, *etc* derivatives by using `diff` more than once. For instance,

```
mat d2f = diff(diff(f,x),x)
```

calculates the second derivative of an `f` macro with respect to `x`. Calculate the elements of the hessian of `criterion` in the same manner, and check that the results are the same as those given by `hess`.

The most important uses of the functions `diff`, `grad`, and `hess` are in the context of nonlinear estimation. We will talk about this in greater length in the next chapter.

* * * *

There exists a last command connected with automatic differentiation, namely, `differentiate`. The syntax it uses is

`differentiate` <*expression*> <*variable*>

where the <*expression*> to differentiate can include macros, and the differentiation is done with respect to the <*variable*>. This command gives nothing in the output file, but it displays some things on screen. If I am not more specific, it is because these things are not simple to understand, being written in an internal representation of ***Ects***. This command is much more useful for me than for you!

* * * *

4. Numerical Integration

There exist today very powerful algorithms for the symbolic integration of functions. These functions are part of software such as Mathematica, Maple, and Maxima, which do symbolic operations on algebraic expressions. At this point in time, ***Ects*** does not include such operations. Yet it is not very difficult to calculate the values of some integrals *numerically*, and version 3.3 of ***Ects*** is endowed with the function `int`, which does such calculations.

Like differentiation, integration operates on *functions*, which must be represented for ***Ects*** by *expressions*, either explicitly, or in macro format. Contrary to `diff`, the `int` function can be used only in a `set` command. This can be explained by the fact that the result of a numerical integration is a scalar. The file `integral.ect` contains many examples of the use of the `int` function.

```
#set showint = 1
set I = int(1,0,3,x)
show I
set I = int(1,3,0,x)
show I
set I = int(x,0,3,x)
show I
set I = int(-x,3,0,x)
show I
set I = int (x^2,0,3,x)
show I
set I = int (x^2,-3,0,x)
show I
set I = int(sin(x),0,PI,x)
show I
set I = int(cos(x),0,PI,x)
show I
set I = int(asin(x),0,1,x)
set J = PI/2-1
show I J
set I = int(diff(chisq(z,8),z),0,1,z)
set J = chisq(1,8) - chisq(0,8)
show I J

set maxintiter=10
set INTTOL=1E-8
set E = 2*int(z*(1-chisq(z,5)),0,200,z) \
  - (int(1-chisq(z,5),0,100,z))^2
show E
quit
```

We see here that the function takes on four arguments, which can be interpreted as follows:

`int(`$\langle expression \rangle, \langle a \rangle, \langle b \rangle, \langle symbol \rangle$`)`

where $\langle expression \rangle$ is the expression to be integrated, $\langle a \rangle$ is the lower bound of the integral, $\langle b \rangle$ is the upper bound, and $\langle symbol \rangle$ is the name of the variable with respect to which we integrate.

Thus, we can interpret the command

```
set I = int(1,0,3,x)
```

as a demand for the evaluation of the integral

$$\int_0^3 1\, dx,$$

whose value is 3. Indeed, after the

```
show I
```

command, ***Ects*** gives the answer

```
I = 3.000000
```

In the same way, the command

```
set I = int (x^2,0,3,x)
```

computes the integral

$$\int_0^3 x^2 \, dx$$

and gives the result

```
I = 9.000000
```

We easily check that this is indeed the right answer.

We know that integration and differentiation are inverse operations. The following commands:

```
set I = int(diff(chisq(z,8),z),0,1,z)
set J = chisq(1,8) - chisq(0,8)
```

demonstrate this fact. Let $F_8(\cdot)$ be the χ^2 distribution function with 8 degrees of freedom. The value of this function evaluated at z can be written in ***Ects*** as `chisq(z,8)`. Then,

$$\int_0^1 F_8'(z)\, dz = \Big[F_8(z)\Big]_{z=0}^{z=1} = F_8(1) - F_8(0),$$

where the last expression can be written as `chisq(1,8) - chisq(0,8)`. Actually, $F_8(0) =$ `chisq(0,8)` $= 0$.

The commands

```
set maxintiter=10
set INTTOL=1E-8
set E = 2*int(z*(1-chisq(z,5)),0,100,z) \
  - (int(1-chisq(z,5),0,100,z))^2
```

illustrate how we can modify the way the function `int` works. We want the variable `E` to contain the value of the variance of a χ^2 with 5 degrees of freedom.

* * * *

The variance of a χ^2 with n degrees of freedom is $2n$. If the calculation is correct, we should obtain a value of 10.

* * * *

If we use F_5 to denote the distribution function of this distribution, we have

$$\begin{aligned}\mathrm{Var}(\chi_5^2) &= E\Big(\big(\chi_5^2\big)^2\Big) - \Big(E\big(\chi_5^2\big)\Big)^2 \\ &= \int_0^\infty z^2\, F_5'(z)\, dz - \left(\int_0^\infty z\, F_5'(z)\, dz\right)^2 \\ &= 2\int_0^\infty z\big(1-F_5(z)\big)\, dz - \left(\int_0^\infty \big(1-F_5(z)\big)\, dz\right)^2.\end{aligned}$$

The last line is the result of an integration by parts: for the second integral note that

$$\frac{d}{dz}\Big(z\big(1-F_5(z)\big)\Big) = 1 - F_5(z) - zF_5'(z)$$

and that

$$\Big[z\big(1-F_5(z)\big)\Big]_{z=0}^{z=\infty} = 0,$$

because $F_5(\infty) = 1$. The first integral can be justified using the same reasoning. The expression given by the `int` command is thus correct, except that we had to replace infinity by a finite value, here 100.

Numerical integration is done using an iterative algorithm. This algorithm is controlled by the values of the variables `maxintiter`, which determines the maximum number of iterations, and `INTTOL`, which determines the convergence criterion used by the algorithm. The iterations stop when the value calculated differs by less than `INTTOL` from one iteration to the next, or when the `maxintiter` iterations have been done, even if convergence has not yet been achieved. In the case of the variance of the χ^2, we see, by running `integral.ect`, that the choice of `maxintiter` and of `INTTOL`, and also of infinity, enables us to obtain the right answer of 10, at least with the precision of the displayed result.

The first line of the the file `integral.ect` has no effect, because it starts by the `#` character. If we take out this character, we can see that the values calculated by successive iterations are displayed on screen. This happens whenever the value of the `showint` variable is nonzero.

Exercises:

Play with the values of `maxintiter` and `INTTOL` until the answer is no longer correct. You can ease this task by looking at intermediate values displayed if the `showint` variable is different from zero.

Chapter 2
Nonlinear Estimation

1. Introduction

Version 3 of ***Ects*** offers a substantially greater choice of nonlinear estimation procedures than its predecessor, which provided only three of them, namely `nls`, `ml`, and `gmm`. These three commands are still present in version 3, but they are joined by six new commands, which serve to extend the possibilities for nonlinear estimation, and facilitate some procedures of which the implementation used to be rather tricky.

2. Nonlinear Least Squares

The command `nls` has changed very little relative to what was in version 2. The main difference from the user's point of view is that it is now possible to make use of automatic differentiation, in order to avoid the all-too-frequent errors that can be made in formulating the derivatives of a regression function in the manner required by ***Ects***.

Here we rapidly recall the syntax of the `nls` command, and some details concerning how it works. The econometric model for which `nls` is used for estimation purposes is the **nonlinear regression model**, which can be expressed as

$$\boldsymbol{y} = \boldsymbol{x}(\boldsymbol{\beta}) + \boldsymbol{u},$$

where $\boldsymbol{y}$ is the dependent variable, $\boldsymbol{u}$ is a vector of disturbances, and $\boldsymbol{x}(\boldsymbol{\beta})$ is the vector of regression functions: see Chapters 2 and 3 of DM for more on this model, as also Chapter 4 of the manual for version 2, Man2. Let us look again at the model used as an example in Man2. The regression function for observation t is

$$\alpha + \beta x_{t1} + \frac{1}{\beta} x_{t2}.$$

The command used in Man2 for estimating this model is:

```
nls y = alpha + beta*x1 + (1/beta)*x2
deriv alpha = 1
deriv beta = x1 - x2/(beta*beta)
end
```

If we prefer to make use of the automatic differentiation facility, another way of doing things is as follows:

```
def fctreg = alpha + beta*x1 + (1/beta)*x2
nls y = fctreg
deriv alpha = diff(fctreg,alpha)
deriv beta = diff(fctreg,beta)
end
```

We begin by defining a macro `fctreg` that represents the regression function. Then, in the `deriv` lines, we can call the `diff` function so as to have the derivative calculated automatically by ***Ects***. We would get the same results if, instead of defining the macro, we put the explicit expression `alpha + beta*x1 + (1/beta)*x2` in its place. But it's clear that a macro is much more convenient in cases in which the regression function is at all complicated.

Before version 3 is was mandatory for the parameters (here `alpha` and `beta`) to be defined in advance, by use of `set` commands for instance. My reasoning for this was that it was in any case necessary to provided starting values for the parameters before setting `nls` in motion. Now we can be a bit more flexible. If a parameter is not defined when `nls` is executed, a default value of zero is assigned to it. This applies not only to `nls`, but to all the nonlinear extimation commands described in this chapter.

We can now follow the progress of the minimisation or maximisation of the criterion function used in nonlinear estimation. If the value of the variable `showprogress` is nonzero, the value of the criterion function is displayed on the screen at the end of each iteration of the optimisation loop.

* * * *

More exactly, the value is sent to the log file. In most cases, this is indeed just the screen (see Chapter 5 of Man2). But if the output to what is called **standard error** of ***Ects*** is redirected, then the log file need not be the screen. If this expression means nothing to you, don't worry. This whole remark is directed at people who want to know everything about computers.

* * * *

The maximal number of iterations is controlled by the variable `maxiter`. Normally, if the optimisation has not converged after `maxiter` iterations, ***Ects*** stops, and asks the question

```
n iterations without convergence.  Continue (y/n)?
```

where n is the number of iterations performed up to this point. If you reply `n` (no), the execution of the command stops at once, whether or not the convergence criterion is met.

In some circumstances, it is not desirable that the user has to give an answer to questions of this sort. For instance, if we are in the middle of a simulation experiment with many replications, we want the program to carry on without interruption until all the replications have been completed. In such cases, we can make use of the variable `noquestions`. If a nonlinear procedure comes to the end of `maxiter` interations and finds that the value of the `noquestions` variable is nonzero, then the execution of the nonlinear command ceases , and ***Ects*** moves on to the next command with neither question nor comment.

In the command file `nls.ect`, which has been slightly modified relative to the file supplied with version 2, the NLS estimation is followed by these commands:

```
gen e = y - alpha - beta*x1 - (1/beta)*x2
gen ralpha = 1
gen rbeta = x1 - x2/(beta*beta)
ols e ralpha rbeta
```

What is being set up here is the GNR (Gauss-Newton regression) that corresponds to the model that has just been estimated: see for example Chapter 6 of DM. It is always highly desirable to run the GNR after any estimation by NLS, so that one may check whether the first-order conditions that define the estimated parameters are indeed satisfied. The regressand of this **artificial regression** is the vector of NLS residuals, and the regressors are the derivatives of the regression function with respect to the model parameters. This being so, we could have made use of the fact that the residuals are made available in the variable `res`. If we run

```
ols res ralpha rbeta
```

we can do without the variable `e` in the above program fragment. In order to make running the GNR after estimation by `nls` even easier, the current version of ***Ects*** recycles the variable `CG`. After an `nls` command, this variable contains the matrix, which we would write as $\boldsymbol{X}(\hat{\boldsymbol{\beta}})$ in algebraic notation, of derivatives of the regression functions with respect to the model parameters, evaluated at $\hat{\boldsymbol{\beta}}$. This matrix is also referred to as the **Jacobian matrix** of the regression functions. We can therefore replace the code given above by one single command:

```
ols res CG
```

to be run immediately after the execution of the nonlinear estimation. If the first-order conditions are satisfied, then the estimated parameters of the GNR, as also the t statistics, are zero up to rounding error.

For a variety of reasons, it may be interesting to know the mean of the dependent variable in a regression model. Although this mean is simple to compute, no explicit calculation is necessary if one makes use of the variable `ybar` (A frequent notation for the mean of a variable y is $\bar{y}$.) This new variable is also made available by the commands `ols` and `iv`. Indeed, if we use these commands for multivariate regressions (that is, with more than one dependent variable), the variable `ybar` becomes a row vector, with one element for each dependent variable.

One last remark about the variables created or updated by `nls`. For reasons having more to do with the history of econometrics than with common sense, it is common practice to include in the regression output an R^2 not only for linear regressions, but also for nonlinear ones. ***Ects*** bows to historical imperatives, and the value given in the output file, also available in the variable `R2`, is equal to `sse/(sse+ssr)`, where `sse` is the explained sum of squares, and `ssr` is the sum of squared residuals. A consequence of this definition is that $0 \leq R^2 \leq 1$. You are formally reminded that the equation `sse+ssr = sst`, where `sst` is the total sum of squares, does *not* hold for nonlinear regressions.

3. Nonlinear Estimation with Instrumental Variables

Estimation with instrumental variables (IV) is necessary whenever the explanatory variables in a model are determined simultaneously with the dependent variable. This rule applies equally well to nonlinear and linear models. With previous versions of ***Ects***, it was necessary in the nonlinear case to use the command `gmm` in order to do IV estimation. (See Section 4.3 of Man2 for the details.) While this procedure is perfectly feasible, it requires a few matrix manipulations that are less than obvious. Consequently version 3 provides a command intended specifically for this sort of estimation.

The command file `nliv.ect` contains an ***Ects*** program that illustrates the use of the `nliv` command. The necessary data are in the data file `ivnls.dat`. In another command file, `ivnls.ect`, there is another program that works with version 2. Comparison of the two command files shows that the new command is easier to use and, in addition, gives better results, on account of an algorithm better adapted to nonlinear IV estimation. Here are the first few commands of `nliv.ect`:

```
sample 1 50
read ivnls.dat y x1 x2 w
iv y c x1 x2 (c x1 w)
set b0 = 1
set b1 = 1
set b2 = 1
nliv y = b0*c + b1*x1 + b2*x2
instr c x1 w
deriv b0 = c
deriv b1 = x1
deriv b2 = x2
end
```

After reading in the data from `ivnls.dat`, we first perform a linear IV estimation, using the old command, `iv`. Then we redo the estimation (unnecessarily) using `nliv`. Observe that the syntax is very similar to that used by the `nls` command. We begin with the `nliv` line, where, following the name of the command, we write down in ***Ects*** notation the equation the parameters of which we want to estimate, linear or nonlinear. So far, this is exactly like what we do with `nls`. On the next line we have something specific to `nliv`, a line headed by the keyword `instr`, followed by the list of variables to be used as instruments. The difference here relative to the linear `iv` command is that this list gets a line to itself instead of being enclosed in parentheses on the command line after the list of explanatory variables. After that, the `deriv` lines are exactly like those used for `nls`. Each line gives the partial derivative of the regression function with respect to one of the parameters. In this particular instance, since the regression function is linear, we have no need of automatic differentiation, the derivatives being just the explanatory variables.

EXERCISES:

Run the above ***Ects*** commands, and show that the two commands `iv` and `nliv` give identical results.

Now for a genuinely nonlinear estimation. If we impose the nonlinear constraint $\beta_2 = 1/\beta_1$, the model becomes

$$\boldsymbol{y} = \beta_0 + \beta_1 \boldsymbol{x}_1 + \boldsymbol{x}_2/\beta_1 + \boldsymbol{u}. \tag{1}$$

The ***Ects*** version of this constrained model is written as

```
set showprogress = 1
gen W = colcat(c, x1, w)
def fctreg = b0*c + b1*x1 + x2/b1
nliv y = fctreg
instr W
deriv b0 = c
deriv b1 = diff(fctreg,b1)
end
ols res CG
```

There are several points to make about this code. First, just as with linear `iv`, the instruments can be grouped together in a matrix with more than one column, and we can mix single-column variables and such matrices. For instance, we could have written

```
gen W = colcat(x1,w)
```

and then for the list of instruments

```
instr c W
```

without any change in the results.

The above code also shows that a macro can be useful in order to represent the regression function. This also simplifies the use of automatic differentiation, as we see here for the parameter `b1`.

As with all nonlinear estimation, it is good practice to check the first-order conditions after the estimation has been done. It turns out that this can be done in exactly the same way as after an `nls` command. The command

```
ols res CG
```

is once again the easiest way to run the **IVGNR** that corresponds to the nonlinear estimation just completed. If we denote the nonlinear regression, as usual, as

$$\boldsymbol{y} = \boldsymbol{x}(\boldsymbol{\beta}) + \boldsymbol{u}\,;$$

and the matrix of instruments as $\boldsymbol{W}$, then the first-order conditions can be written as

$$\boldsymbol{X}^\top(\hat{\boldsymbol{\beta}})\boldsymbol{P_W}\big(\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\big) = \boldsymbol{0}, \tag{2}$$

where the matrix $\boldsymbol{X}(\boldsymbol{\beta})$ is again the Jacobian matrix of the regression functions, that is, the partial derivatives of the components of the vector $\boldsymbol{x}(\boldsymbol{\beta})$ with respect to the parameters, and $\boldsymbol{P_W}$ is the orthogonal projection on to the space spanned by the instrumental variables; see DM, Chapter 7. The variable `res`, as we might expect, holds the residuals of the regression, that is, the vector $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$. The matrix `CG` is now the matrix we can write algebraically as $\boldsymbol{P_W}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$. If the first-order conditions (2) are satisfied, then the regression of `res`, or $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$, on `CG`, or $\boldsymbol{P_W}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$, gives parameter estimates and t statistics equal to zero, since the residuals are orthogonal to the columns of $\boldsymbol{P_W}\boldsymbol{X}(\hat{\boldsymbol{\beta}})$.

You may have noticed that, before running the `nliv` command, we set the variable `showprogress` equal to 1. As a result, while the `nliv` command is running, we can follow the progress of the iterations on the screen:

```
crit = 22.1712; newcrit = 10.3901
crit = 10.3901; newcrit = 0.546993
crit = 0.546993; newcrit = 0.161842
crit = 0.161842; newcrit = 0.156805
crit = 0.156805; newcrit = 0.156803
crit = 0.156803; newcrit = 0.156803
```

Each of the above lines represents one iteration of the nonlinear procedure that tries to minimise a criterion function. For nonlinear IV, the appropriate criterion function is

$$Q(\boldsymbol{\beta}) = \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^\top\boldsymbol{P_W}\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big). \tag{3}$$

It is not hard to show that the first-order conditions for a minimum of (3) are just the estimating equations (2). The value of this function $Q(\boldsymbol{\beta})$ at the start of each iteration is what is given by the variable `crit`; at the end of the iteration the value is given by `newcrit`. As we see, after the first iteration, `crit` is equal to the `newcrit` of the previous iteration. At each step, the value of $Q(\boldsymbol{\beta})$ decreases, until the minimum is attained. At the end, therefore, the ***Ects*** variable `crit` contains the minimised value.

Let us now inspect the results of the `nliv` command, as found in the output file:

```
Nonlinear Instrumental Variables Estimation:

Number of iterations = 6

Parameter  Parameter estimate  Standard error  T statistic

b0             -0.014659            0.069393      -0.211248
b1              0.328674            0.014647      22.439042

Number of observations = 50   Number of estimated parameters = 2
Number of instruments = 3
Mean of dependent variable = 2.030593
Sum of squared residuals = 4.316909
Explained sum of squares = 293.919327
Estimate of residual variance
  (with d.f. correction) = 0.089936
Mean of squared residuals = 0.086338
Standard error of regression = 0.299893
Overidentification statistic = 1.816150

Estimated covariance matrix:

0.004815  0.000805
0.000805  0.000215
```

The table of results contains the usual things: the parameter estimates, available as usual in the vector `coef`, along with the standard errors and the t statistics, available in the vectors `stderr` and `student` respectively. The covariance matrix, printed as always at the end of the table, is the matrix $\hat{\sigma}^2(\hat{\boldsymbol{X}}^\top \boldsymbol{P}_{\boldsymbol{W}} \hat{\boldsymbol{X}})^{-1}$, where $\hat{\boldsymbol{X}} \equiv \boldsymbol{X}(\hat{\boldsymbol{\beta}})$, and

$$\hat{\sigma}^2 = \frac{1}{n-k}\|\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2. \tag{4}$$

This matrix can be found under the name `vcov`, and the value of $\hat{\sigma}^2$ is found in the variable `errvar`. The choice of a denominator of $n-k$ or of n has little importance with IV estimation, If it matters to us, we can note that `Mean of squared residuals` gives the value of (4) with $n-k$ replaced by n.

After a linear IV estimation, the matrix $(\boldsymbol{X}^\top \boldsymbol{P}_{\boldsymbol{W}} \boldsymbol{X})^{-1}$ is saved in the variable `XtPwXinv`. For a nonlinear estimation, the matrix saved in that variable is the matrix expressed algebraically as $(\hat{\boldsymbol{X}}^\top \boldsymbol{P}_{\boldsymbol{W}} \hat{\boldsymbol{X}})^{-1}$.

The other variables generated by `nliv` can be briefly described. The variables `res` and `fit` respectively contain the residuals $\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})$ and the fitted values $\boldsymbol{x}(\hat{\boldsymbol{\beta}})$ of the regression. The scalar variables `ssr`, `sse`, `sst`, and `ybar` contain the following expressions:

$$\texttt{ssr} = \|\boldsymbol{y} - \boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2, \quad \texttt{sse} = \|\boldsymbol{x}(\hat{\boldsymbol{\beta}})\|^2, \quad \texttt{sst} = \|\boldsymbol{y}\|^2, \quad \texttt{ybar} = n^{-1}\sum_{t=1}^{n} y_t.$$

The sample size, n, is found in `nobs`, the number of parameters, k, in `nreg`, the number of instruments in `ninst`, and the number of iterations in `niter`.

The last line of the table before the covariance matrix gives the `Overidentification statistic`. This statistic is equal to $Q(\hat{\boldsymbol{\beta}})/\hat{\sigma}^2$, in other words, the minimised value of the criterion function (3), divided by the estimate of the variance of the disturbances that uses a denominator of n instead of $n - k$. Under the null hypothesis according to which $\mathrm{E}(u_t \,|\, \boldsymbol{W}_t) = 0$, the statistic is asymptotically distributed as χ^2 with $l - k$ degrees of freedom, where l is the number of instruments, so that $l - k$ is equal to the degree of overidentification. The statistic, available in the variable `oir`, is provided not only by `nliv`, but also by `iv`.

We may remark here that, in version 3 of ***Ects***, the `iv` command generates a new matrix variable, with the name `PwX`. This is an $n \times k$ matrix, equal to the algebraic expression $\boldsymbol{P_W X}$.

EXERCISES:

Redo the estimation of model (1) using the data of `ivnls`, but with `b1=1` as starting value. You will see that the algorithm converges to a different point from the one found before.

This phenomenon arises from the fact that the criterion function $Q(\boldsymbol{\beta})$ has two local minima, of which only one is the global minimum. Before studying these two minima in more detail, show that, if the variables $\boldsymbol{y}$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$ and $\boldsymbol{w}$ are centred, for example by the action of the orthogonal projection $\boldsymbol{M_\iota}$ that annihilates the constant, then the estimation of the model

$$\boldsymbol{M_\iota y} = \beta_1 \boldsymbol{M_\iota x}_1 + \boldsymbol{M_\iota x}_2/\beta_1 + \boldsymbol{u}, \tag{5}$$

with the two instruments $\boldsymbol{M_\iota x}_1$ and $\boldsymbol{M_\iota w}$, gives the same estimate of β_1, and the same residuals, as the estimation of model (1).

This reduction is convenient, since the criterion function for the model (5) depends on only one parameter, namely β_1. Graph the criterion function as a function of β_1 on the interval $[0.2, 2.5]$. This will show the two minima of the function quite clearly.

4. Maximum Likelihood

The functionality provided by ***Ects*** for maximum likelihood (ML) estimation has been very considerably extended in version 3. We can take a look at a **logit model** in order to appreciate the different procedures that are available. For such a model, the dependent variable is a **binary variable**, by which we mean a variable that takes on one of only two possible values, conventionally set as 0 and 1. For observation t, the probability that the dependent variable y_t is equal to 1 is given by

$$\Pr(y_t = 1) = F(\boldsymbol{X}_t\boldsymbol{\beta}),$$

where $F(\cdot)$ is a function the values of which are confined to the $[0, 1]$ interval, $\boldsymbol{X}_t$ is a $1 \times k$ vector of explanatory variables, and $\boldsymbol{\beta}$ is a $k \times 1$ vector of parameters to be estimated. The theory of **binary choice models** can be found in Chapter 15 of DM, and an example of logit estimation can be found in Section 5.3 of Man2. We take up this example again in order to illustrate the implementation of the different methods of ML estimation that can be used.

The loglikelihood function of the model can be written as a sum of contributions from the individual observations:

$$\ell(\boldsymbol{y}, \boldsymbol{\beta}) = \sum_{t=1}^{n} \ell_t(y_t, \boldsymbol{\beta}),$$

where the contribution $\ell_t(\cdot)$ from observation t is defined as follows:

$$\ell_t(y_t, \boldsymbol{\beta}) = y_t \log\bigl(F(\boldsymbol{X}_t\boldsymbol{\beta})\bigr) + (1 - y_t)\log\bigl(1 - F(\boldsymbol{X}_t\boldsymbol{\beta})\bigr). \tag{6}$$

We can see that there is in fact only one term per observation: If $y_t = 1$, the second term vanishes; if $y_t = 0$, the first term vanishes. In either case, the contribution is the log of the probability of having observed y_t.

The logit model is the special case of a binary choice model for which the function $F(\cdot)$ is the **logistic function**:

$$F(x) = \frac{e^x}{1 + e^x}. \tag{7}$$

We can check that, with this definition, F is an increasing function of its argument. In fact, F has all the properties of a **cumulative distribution function**, that is:

$$\lim_{x\to-\infty} F(x) = 0, \quad \lim_{x\to\infty} F(x) = 1, \quad F'(x) \geq 0.$$

The probability distribution characterised by the logistic function is, not surprisingly, called the **logistic distribution**.

In the command file `newlogit.ect` are found the commands necessary for the estimation of a logit model. For the data, we reuse the variables in the data file `ols.dat`. We generate the variables that we actually use by the following commands:

```
sample 1 100
read ols.dat y x1 x2 x3
gen x1 = x1/100
gen x2 = x2/100
gen x3 = x3/100
gen Y = y - 100
gen y = 0.5*(sign(Y) + 1)
```

The binary variable `y` is generated by the last two lines. The **limited dependent variable** `y` is equal to 1 if the continuous variable `Y` is greater than 100, and to 0 otherwise. Thus the probability that $y_t = 1$ is equal to the probability that $Y_t > 100$.

* * * *

Observe that the explanatory variables $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ are all divided by 100. This is a precautionary measure taken so as to avoid potential numerical difficulties. The exponential function e^x grows extremely rapidly with x, so much so that there is a danger of needing a value greater than the maximum value that can be represented in the floating-point arithmetic of the computer. Even if that does not happen – it depends on how many bits the particular computer architecture uses for its floating-point representation – there is a risk of substantial rounding errors.

* * * *

In addition to the explanatory variables $\boldsymbol{x}_i$, $i = 1, 2, 3$, we need the constant, denoted by `iota`. Recall also that macros are very useful for representing nonlinear functions. The code continues as

```
gen iota = 1
def X = a*iota + x1*b1 + x2*b2 + x3*b3
def e = exp(X)
def F = e/(1+e)
def lhd = y*log(F)+(1-y)*log(1-F)
def dldF = (y-F)/(F*(1-F))
def dFde = 1/(1+e)^2
```

The macro `lhd` gives us the loglikelihood function, `dldF` is the derivative of ℓ with respect to F, and `dFde` is the derivative of F with respect to `e` $= \exp(\boldsymbol{X}_t\boldsymbol{\beta})$.

* * * *

In the execution of the command `ml` and the other related commands in the same family that we will see shortly, all expressions are evaluated as in a `gen` command. It is therefore not strictly necessary to use the vector `iota` as we have used it here. But it is good practice to make the constant into an explicit vector, for in the commands of the `gmm` family expressions are evaluated as in a `mat` command, in which a scalar is just that, and is never interpreted as a vector.

* * * *

The next set of commands, in which we use the `ml` command, would be fine with version 2 of ***Ects***:

```
set a = 0
set b1 = 0
set b2 = 0
set b3 = 0
ml lhd
deriv a = dldF*dFde*e
deriv b1 = dldF*dFde*e*x1
deriv b2 = dldF*dFde*e*x2
deriv b3 = dldF*dFde*e*x3
end
ols iota CG
```

Version 3 accepts the same commands, of course. However. it is not *necessary* to initialise the parameters `a`, `b1`, *etc.*, especially if the starting values are zero, as here. Of course an explicit initialisation never does any harm, and there are of course cases in which a starting value of 0 could be dangerous.

* * * *

The final command, `ols iota CG`, is the standard highly desirable check of the first-order conditions.

* * * *

Analytic differentiation of the loglikelihood function of our model with respect to the parameters is not an unduly hard task. Indeed, the explicit form of the logistic function allows a certain simplification. Let F_t denote the function $F(\boldsymbol{X}_t\boldsymbol{\beta})$, and e_t the function $\exp(\boldsymbol{X}_t\boldsymbol{\beta})$. Then

$$\frac{\partial \ell_t}{\partial \beta_i} = \frac{\partial \ell_t}{\partial F_t}\frac{\partial F_t}{\partial e_t}\frac{\partial e_t}{\partial \beta_i} = \frac{y_t - F_t}{F_t(1-F_t)}\frac{1}{(1+e_t)^2}e_t X_{ti},$$

where X_{ti} is the value of explanatory variable i for observation t. Now

$$F_t(1-F_t) = \frac{e_t}{1+e_t}\frac{1}{1+e_t} = \frac{e_t}{(1+e_t)^2},$$

so that

$$\frac{\partial \ell_t}{\partial \beta_i} = X_{ti}(y_t - F_t).$$

Thus the `deriv` lines can be replaced by

```
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
```

But, if we prefer not to trust ourselves to do even this simple differentiation, automatic differentiation is always to hand. In order to use it, all we need do is

```
  deriv a = diff(lhd,a)
  deriv b1 = diff(lhd,b1)
  deriv b2 = diff(lhd,b2)
  deriv b3 = diff(lhd,b3)
```

Exercises:

Try these three methods of estimation, with the derivatives unsimplified, simplified, and computed automatically. The results should be identical, but computing times may be sharply different.

Now let's look at the results.

```
Maximising a Sum of Contributions:

Number of iterations = 22

Parameter  Parameter estimate  Standard error  T statistic

a             -2.535442           1.712046      -1.480943
b1             8.625195           2.306146       3.740090
b2           -13.131199           2.791365      -4.704222
b3             4.078517           1.329014       3.068830

Number of observations = 100   Number of estimated parameters = 4
Maximised value of criterion function = -28.246752

Estimated covariance matrix:

 2.931102  -3.238547   1.800739   0.895411
-3.238547   5.318312  -4.811321   0.502940
 1.800739  -4.811321   7.791716  -1.838118
 0.895411   0.502940  -1.838118   1.766278
```

This table can be compared with the one on page 54 of Man2. The estimated constant is identical, and the other parameters are 100 times greater than those in the other table. This is as it should be: we divided the explanatory variables by 100. The standard errors and t statistics are however a bit different, even if we take account of the division by 100. The estimated covariance matrices are also somewhat different. The reason for this is connected to the use of different estimation techniques. In Man2 we used an **artificial regression** whereas the estimation we have just performed here makes use of the DFP algorithm (see Man2), which gives only an approximation to the Hessian of the loglikelihood function, of which the negative of the inverse can be used as an estimate of the covariance matrix of the parameter estimates. The matrix given by the artificial regression is more reliable.

* * * *

22 iterations were needed here against only 5 for the artificial regression. Three different reasons are behind this phenomenon. First, the artificial regression, being specific to the logit model, is more efficient than the all-purpose `ml` procedure. Next, our starting point, with all parameters set to zero, is less favourable than the one used by the other procedure. Finally, the convergence criterion used by `ml` is stricter than the one used with the artificial regression.

* * * *

The covariance matrix, as estimated by `ml`, is available after the execution of the command in the variable `invhess`. The name is slightly misleading, for the reason we have just seen. The other variables generated by `ml` are: `coef`, the parameter estimates, `stderr`, the standard errors; `student`, the t statistics; `nobs`, the sample size; `nreg`, the number of estimated parameters; `niter`, the number of iterations; `lt`, the vector of contributions to the loglikelihood function, evaluated at $\hat{\boldsymbol{\beta}}$; `lhat`, the maximised loglikelihood function, of which the value is the sum of the elements of `lt`; and `CG`, the CG matrix of contributions to the gradient, of which the (t,i) element is $\partial\ell_t/\partial\beta_i(\hat{\boldsymbol{\beta}})$.

Exercises:

Redo the estimation without dividing the explanatory variable by 100. You will probably run into some numerical difficulties. See if dividing by 10 instead of 100 solves the problem.

As we make our way through the command file `newlogit.ect`, the estimation is redone using a new command, `mlopg`, in place of `ml`. On most computers, this command is quicker. The results are very similar, but the estimated covariance matrix is once again different:

```
Estimated covariance matrix:

 2.883893  -3.140939   2.172086   1.122439
-3.140939   5.613164  -6.148303   0.672352
 2.172086  -6.148303  10.784860  -2.342453
 1.122439   0.672352  -2.342453   2.318406
```

The theory of artificial regressions is laid out in an article of Davidson and MacKinnon (1999). We see there that, for essentially all models that can be estimated by ML, the OPG artificial regression can be used in the algorithm for maximising the loglikelihood function. This artificial regression, which is very straightforward to implement, is what is used by `mlopg`. The very fact of the widespread applicability of the artificial regression often leads to a lack of efficiency. And indeed we see here that the number of iterations used by `mlopg` is even greater than what we had with `ml`. This defect is compensated, in many cases, by the simplicity of the algorithm and the calculations it requires.

The estimated covariance matrix, quite naturally, is the one known as the OPG estimator; see Chapter 8 of DM and Davidson and MacKinnon (1999).

It is available in the variable invOPG. Unfortunately, this estimate is not very reliable. Although use of the mlopg command leads to a short computing time, it is prudent to recompute the covariance matrix by use of some more reliable procedure if one wants to perform inference.

* * * *

The regression, ols iota CG, used to check the first-order conditions, is an OPG regression.

* * * *

There is a difference of sign between the maximised value of the loglikelihood function and the values of crit and newcrit as displayed during the execution of ml and mlopg if the value of showprogress is nonzero. For the former, we have

```
Maximised value of criterion function = -28.246752
```

while for the latter an instance of what is displayed is

```
crit = 28.2471; newcrit = 28.2468
```

The reason is trivial: in the inner workings of ***Ects***, all criterion functions are minimised. In order to conform to this, the negative of the loglikelihood function is minimised. It is the value given in the table of results that is the right one.

Further on still in newlogit.ect, the command mlhess is used to estimate our logit model. As the name suggests, the algorithm used in this command makes use of the **Hessian** of the loglikelihood function. The basic principle that underlies the majority of maximisation/minimisation algorithms is **Newton's method**. In its original form, this method makes use of both first and second derivatives of the function to be optimised. This means that, for Newton's method, we need both the gradient and the Hessian of the loglikelihood function. This fact emerges clearly in the first estimation that uses mlhess:

```
def f = e/(1+e)^2
mlhess lhd
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
second a,a = -f
second a,b1 = -x1*f
second a,b2 = -x2*f
second a,b3 = -x3*f
second b1,b1 = -x1*x1*f
second b1,b2 = -x1*x2*f
second b1,b3 = -x1*x3*f
second b2,b2 = -x2*x2*f
second b2,b3 = -x2*x3*f
second b3,b3 = -x3*x3*f
end
```

The elements of the Hessian matrix are easy enough to express algebraically if we use the macro `f`. The syntax of `mlhess` is not hard. The use of the keyword `second` in place of `deriv` indicates that what follows is a second derivative. After the keyword, we need two parameters instead of just one, since a second derivative involves two differentiations. Given that, for a continuous loglikelihood function, the order of differentiation is unimportant, it is enough to specify the derivative `second a,b1`, for instance: ***Ects*** knows that this is also the derivative `second b1,a`.

Exercises:

Show analytically that the second derivatives used in the program do indeed represent the second derivatives of the function (6).

Where is our much-vaunted automatic differentiation in all this? Nowhere at all, so far, clearly. But we could have obtained the same results using the following code:

```
mlhess lhd
deriv a = y-F
deriv b1 = x1*(y-F)
deriv b2 = x2*(y-F)
deriv b3 = x3*(y-F)
end
```

without any second derivative at all being specified. ***Ects*** is able to formulate the needed second derivatives all by itself. In fact, one can mix and match: If the user provides a subset of the necessary second derivatives, they will be used, and only the missing ones will be generated by ***Ects***.

Why not leave all the work of calculating derivatives, first and second, to ***Ects***? If that is what we wish, then the following code

```
mlhess lhd
deriv a = diff(lhd,a)
deriv b1 = diff(lhd,b1)
deriv b2 = diff(lhd,b2)
deriv b3 = diff(lhd,b3)
end
```

gives exactly the same results as our earlier procedures that use `mlhess`. The only problem is that computing time will be a *lot* longer. Version 3 of ***Ects*** knows perfectly well how to differentiate, but it has no means of *simplifying* the sometimes very long expressions that result from a differentiation. Consequently, although we end up with the same result, the path leading to it is much less direct when automatic differentiation is employed than if short simple expressions for the first and second derivatives are supplied.

The covariance matrix as estimated by `mlhess` is available in the variable named `invhess`. But this time, unlike the situation with `ml`, we get the inverse of the true Hessian rather than of an approximation.

Exercises:

Run the four `mlhess` estimation procedures, as found in the file `newlogit.ect`, on your usual computer, and compare the computing times. If you set `showprogress` to a nonzero value, you will be able to see the time taken by each iteration.

If we look more closely at the results of the OPG regression,

```
    ols iota CG
```

used to check the first-order conditions after the different estimations carried out by means of `ml` or `mlopg`, we see that the t statistics are indeed small, for instance `-1.216695e-10`, but not zero. But the t statistics we find after estimation by `mlhess` are indeed zero to machine precision. This is explained by the fact that Newton's method, the real one that uses second derivatives, converges more quickly in the neighbourhood of the maximum than the approximate quasi-Newton methods employed by `ml` and `mlopg`. In the case we are studying, convergence is globally faster, as can be seen by the much smaller number of iterations used, 8 here, against 22-24 with the other methods, but starting from the same point. The conclusion is thus that, when we want maximum precision, `mlhess` is our best bet. The cost takes the form either of longer computing times, or else laborious hand calculation of second derivatives.

In the command file `logit.ect`, distributed with version 2 of ***Ects***, estimation of our logit model is performed with the help of the artificial regression known as the **BRMR**, for *Binary Response Model Regression* – see Section 15.4 de DM. The same procedure, slightly modified, appears at the end of `newlogit.ect`. As shown in the article by Davidson and MacKinnon (1999), artificial regressions exist for many sorts of model, and they often can be used for efficient estimation routines. We can take advantage of this fact if we make use of the command `mlar`, where the `ar` signifies *Artificial Regression*.

For the general model characterised by (6). the BRMR can be written as

$$\frac{y_t - F_t}{\big(F_t(1-F_t)\big)^{1/2}} = \frac{f_t \boldsymbol{X}_t}{\big(F_t(1-F_t)\big)^{1/2}} \boldsymbol{b} + \text{residual},$$

where $F_t \equiv F(\boldsymbol{X}_t\boldsymbol{\beta})$, and $f_t \equiv f(\boldsymbol{X}_t\boldsymbol{\beta}) = F'(\boldsymbol{X}_t\boldsymbol{\beta})$. For the logit model, F is given by (7), and

$$F'(x) = f(x) = \frac{e^x}{(1+e^x)^2}.$$

We already had a macro in order to evaluate $f(\cdot)$: we used it for the second derivatives needed by `mlhess`. The syntax of an `mlar` command is illustrated by the following code:

```
  def arden = sqrt(F*(1-F))
```

```
mlar lhd
lhs = (y-F)/arden
deriv a = f/arden
deriv b1 = f*x1/arden
deriv b2 = f*x2/arden
deriv b3 = f*x3/arden
end
```

The second line above defines the **regressand** of the artificial regression. The keyword `lhs` signals the definition of the regressand, which is the left-hand side variable of the regression. Next, we have as usual a number of `deriv` lines, used here to specify the **regressors** of the artificial regression. The macro `arden` helps considerably in the expression of these artificial variables. It is important to note that, in general, the regressors, although signalled by the keyword `deriv`, need not be derivatives.

According to the rules of artificial regressions, an estimate of the covariance matrix of the parameter estimates is given by the regressors of the artificial regression. If we write $\boldsymbol{R}(\boldsymbol{\beta})$ for the matrix of regressors, then the estimated covariance matrix is the inverse of $\boldsymbol{R}^\top(\hat{\boldsymbol{\beta}})\boldsymbol{R}(\hat{\boldsymbol{\beta}})$. This inverse matrix is available after an `mlar` command in the variable `XtXinv`.

When we run our `mlar` command, we see that the results are exactly identical to those given by `mlhess`. This is not an accident; rather it is a consequence of a very specific property of the BRMR for a logit model.

Exercises:

Let $\boldsymbol{R}(\boldsymbol{\beta})$ be the matrix of regressors for the logit BRMR. Show that the negative of $\boldsymbol{R}^\top(\boldsymbol{\beta})\boldsymbol{R}(\boldsymbol{\beta})$ is equal to the Hessian of the loglikelihood function. Why does this explain the equivalence of the results of `mlhess` and `mlar`?

Checking the first-order conditions after an `mlar` command has to be done a little differently. What we want to check is that the estimated parameters and t statistics of the artificial regression itself are zero after convergence. We can do this as follows:

```
gen r = (y-F)/arden
ols r CG
```

The artificial regression is run by the command `ols r CG`, because, after an `mlar` command, the matrix `CG` is no longer the usual CG matrix, but rather the matrix of the regressors of the artificial regression, evaluated at $\hat{\boldsymbol{\beta}}$.

Exercises:

Use the `mlar` command to estimate a nonlinear regression model by use of the GNR. For instance, you might use the model we considered in Section 2 of this chapter. Minus the squared residual can serve as the criterion function (why?).

5. The Generalised Method of Moments

The generalised method of moments (GMM), which was thought of as a "sophisticated" technique when Man2 was written, is now a standard tool in econometrics. Chapter 17 of DM remains a solid reference on the subject. In ***Ects***, the old `gmm` command, available since version 2, is now joined by two new commands, `gmmhess` and `gmmweight`. The first of these, `gmmhess`, like `mlhess`, makes use of analytic second derivatives of the criterion function. The second, `gmmweight`, is altogether different, as we will see in due course.

Relative to the `ml` family of commands, the main difference for users is that expressions are evaluated under `gmm` and `gmmhess` as they would be in `mat` command, rather than as in a `gen` command. This means, for example, that the `smplstart` and `smplend` variables, the values of which are assigned by the `sample` command, have no effect. Another trivial, but important, difference is that the criterion function is *minimised*.

We can study how to use `gmm` and `gmmhess` by looking at the command file `gmm.ect`. This file uses the same data and the same models as those used in `nliv.ect`. Of course GMM is not limited to regression models estimated with instrumental variables, but the simplicity of those models makes a suitable vehicle for explaining the ***Ects*** commands.

Recall that the criterion function to minimse is given in expression (3). Expressing this function and its derivatives in proper ***Ects*** syntax, we obtain:

```
sample 1 50
read ivnls.dat y x1 x2 w
gen iota = 1
gen W = colcat(iota, x1, w)
mat WtWinv = (W'*W)inv
def resid = y - b0*iota - b1*x1 - b2*x2
gmm resid'*W*WtWinv*W'*resid
deriv b0 = -2*iota'*W*WtWinv*W'*resid
deriv b1 = -2*x1'*W*WtWinv*W'*resid
deriv b2 = -2*x2'*W*WtWinv*W'*resid
end
sample 1 3
print grad
```

* * * *

It *is* essential to make the constant vector `iota` explicit with `gmm` and `gmmhess` commands. If we don't, then the dimensions of our matrices will be incorrect for matrix multiplication.

* * * *

We can check easily enough that the expression `resid'*W*WtWinv*W'*resid` represents the function $Q(\boldsymbol{\beta})$ of (3), because $\boldsymbol{P_W} = \boldsymbol{W}(\boldsymbol{W}^\top\boldsymbol{W})^{-1}\boldsymbol{W}^\top$. Similarly, the expressions in the `deriv` lines represent the derivatives of Q.

The table of results has much less information than the tables generated by `ols`, `ml`, *etc.* Here it is:

```
Minimising a Criterion Function:

Number of iterations = 13

b0 = -0.166437
b1 = 0.586427
b2 = 2.732630

Number of estimated parameters = 3
Minimised value of criterion function = 0.000000

Estimate of Inverse of Hessian of Criterion Function:

 0.100231  -0.120279   0.108585
-0.120279   0.213099  -0.265844
 0.108585  -0.265844   0.408608
```

We can see that there are a few differences with respect to the table on page 45 of Man2. Some are due simply to the fact that the variables have been divided by 100. The others are pretty minor, and are associated with the approximate nature of the Hessian of Q.

There are not very many variables generated by `gmm`. As usual, the parameter estimates are to be found in the variable `coef`. `nreg` gives the number of estimated parameters, `niter` the number of iterations, and `crit` the minimised value of the criterion function. The inverse of the (approximate!) Hessian of the criterion function is in `invhess`, while the gradient is in `grad`. This last variable is made available so that we can check the first-order conditions – it should be very close to zero if convergence is achieved. Most often, we do not have a suitable simple artificial regression with `gmm`, and so the method we use with other nonlinear commands does not work here.

Here again we can use automatic differentiation. A version that does so can be found in `gmm.ect`, as well as the GMM estimation of the model under the nonlinear constraint $\beta_2 = 1/\beta_1$.

The `gmmhess` command uses exactly the same syntax as `gmm`. Later on in `gmm.ect`, we find it used to redo the estimation of the nonlinear model, with and without automatic differentiation. With `gmmhess`, of course, the Hessian is no longer an approximation. Comparison of the results given by `gmm` and `gmmhess` show that the approximation used by the former is pretty crude. We can see as well that the number of iterations needed by `gmmhess` for convergence is smaller than what is needed by `gmm`, 11 against 16. The variables generated by `gmmhess` are the same as those generated by `gmm`.

Like `mlhess`, `gmmhess` allows the user to define his/her own second derivatives. Further on in `gmm.ect`, we find the following code:

```
  def dresd1 = -x1 + x2/b1^2
  gmmhess residual'*W*WtWinv*W'*residual
  deriv b0 = -2*iota'*W*WtWinv*W'*residual
```

```
deriv b1 = -2*(x1 - x2/(b1*b1))'*W*WtWinv*W'*residual
second b0,b0 = 2*iota'*W*WtWinv*W'*iota
second b0,b1 = -2*iota'*W*WtWinv*W'*dresd1
second b1,b1 = 2*dresd1'*W*WtWinv*W'*dresd1 - \
4*(x2/b1^3)'*W*WtWinv*W'*residual
end
```

where everything is explicit. The results are identical.

In the above code, we made use of the possibility of continuing a command line on to the next line by means of the character \. When ***Ects*** reads a command line, if the final chracter is \, then the contents of the next line are read in and appended to the preceding line. This makes it possible to have ***Ects*** read very long commands, spread out over several lines, with a \ at the end of each line except the very last.

The new `gmmweight` command is quite different from `gmm` and `gmmhess`. First, as with the `ml` family, expressions are evaluated as though with `gen`, and the sample size is respected, as declared in `smplstart` and `smplend`. However, another feature of `gmmweight` is that the command can be applied only to a specific class of models.

The criterion function minimised in nonlinear IV estimation has the form (3), and this is precisely the form required by `gmmweight` if we write out the projection matrix explicitly:

$$Q(\boldsymbol{\beta}) = \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^{\top}\boldsymbol{W}(\boldsymbol{W}^{\top}\boldsymbol{W})^{-1}\boldsymbol{W}^{\top}\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big). \tag{8}$$

In the terminology of GMM, the vector $\boldsymbol{W}^{\top}\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)$ is a vector of **moments**, that is, a vector of which each component is a function of the data and the parameters of the model. If we were to evaluate the function at the true parameter values, the expectations of the functions would be equal to zero. The criterion function is a quadatic form defined by these moments and a **weight matrix**, here the matrix $(\boldsymbol{W}^{\top}\boldsymbol{W})^{-1}$, which is required to be positive definite. The GMM estimator is efficient if the weight matrix is (asymptotically) proportional to the inverse of the covariance matrix of the moments evaluated at the true parameters. It can be seen that, under the assumption that the residuals $\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})$ are homoskedastic and serially uncorrelated, this condition is satisfied by the function (8).

It is often the case in practice that the moments are defined in terms of a vector $\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta})$ that depends on the data $\boldsymbol{y}$ and the parameters $\boldsymbol{\beta}$, whose dimension is the sample size. In the present instance, this vector is just the vector of residuals, $\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})$. The moments are constructed as scalar products of this vector with a set of l variables, that we conventionally call instrumental variables. It is plain that, for a regression model estimated by IV, the columns of the matrix $\boldsymbol{W}$ are indeed the instruments not only in the ordinary sense, but also in the specialised terminology of GMM. It can be shown (see for instance DM, Chapter 17) that, if the moments are defined as the elements of the vector $\boldsymbol{W}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta})$, then, with any positive definite $l \times l$ weight matrix $\boldsymbol{A}$, the minimisation of the criterion function

$$\boldsymbol{f}^\top(\boldsymbol{y}, \boldsymbol{\beta})\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{\beta}) \tag{9}$$

gives a consistent, but inefficient, estimator of the model parameters.

In order to use the `gmmweight` command, we must specify the three ingredients of the structure we have just described, that is, the vector $\boldsymbol{f}$, the instruments $\boldsymbol{W}$, and the weight matrix $\boldsymbol{A}$. The following commands, taken from the end of the command file `gmm.ect`, illustrate the syntax:

```
sample 1 50
def residual = y - b0*iota - b1*x1 - (1/b1)*x2
gen W = colcat(iota, x1, w)
gmmweight residual
instr W
weightmatrix = (W'*W) inv
deriv b0 = -iota
deriv b1 = diff(residual,b1)
end
```

The name of the command, `gmmweight`, is followed by the expression that represents the vector $\boldsymbol{f}$; here we use the macro `residual`. On the following line, just as for the `nliv` command, we put the keyword `instr` followed by a list of instruments, in which we may mix vectors and matrices. Next, on the following line, the keyword `weightmatrix` is followed by the equals sign "=", and an expression that represents the weight matrix $\boldsymbol{A}$. This expression, and *only* this expression, is evaluated according to the rules of a `mat` rather than a `gen` command. After all this, we have the usual series of `deriv` lines, one for each parameter to be estimated, in which we give the partial derivatives of $\boldsymbol{f}$ with respect to the parameters. Finally, as with all multiline commands, we have a final `end`.

Now we may look at the results of this command. Because the model is much more structured than the general models that can be estimated with `gmm`, the table of results is much more complete. We can check that the numerical results are all identical to those we got from `nliv` for the same model, and that the number of iterations is similar to the number used by `nliv`, and consequently less than the number needed by either `gmm` or `gmmhess` in order to achieve convergence.

```
Minimising a Quadratic Form:

Number of iterations = 7

Parameter  Parameter estimate  Standard error  T statistic

b0              -0.014659           0.069393      -0.211248
```

```
b1             0.328674           0.014647      22.439041

Number of observations = 50   Number of estimated parameters = 2
Number of instruments = 3
Sum of squared residuals = 4.316909
Estimate of residual variance
  (with d.f. correction) = 0.089936
Mean of squared residuals = 0.086338
Standard error of regression = 0.299893
Minimised value of criterion function = 0.156803

Estimated covariance matrix:

0.004815  0.000805
0.000805  0.000215

Heteroskedasticity Consistent Estimate:
 (Note: Estimate above valid only with efficient weightmatrix)

0.006719  0.001158
0.001158  0.000262
```

We said earlier that the GMM estimator is efficient if the weight matrix is proportional to the inverse of the covariance matrix of the moments. If it is not, then the first of the two estimated covariance matrices is not correct. This matrix, made available in the variable `vcov`, is given by the following matrix expression:

$$\hat\sigma^2\hat{\boldsymbol{V}} \equiv \hat\sigma^2(\hat{\boldsymbol{X}}^\top\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top\hat{\boldsymbol{X}})^{-1},$$

where

$$\hat\sigma^2 = \frac{1}{n-k}\boldsymbol{f}^\top(\boldsymbol{y},\hat{\boldsymbol{\beta}})\boldsymbol{f}(\boldsymbol{y},\hat{\boldsymbol{\beta}})$$

and $\hat{\boldsymbol{X}} = \boldsymbol{X}(\hat{\boldsymbol{\beta}})$, $\boldsymbol{X}(\cdot)$ being the Jacobian matrix of the components of $\boldsymbol{f}$ with respect to the k components of $\boldsymbol{\beta}$. The value of $\hat\sigma^2$ is saved in the variable `errvar`. It is not too hard to show that the matrix $\hat\sigma^2\hat{\boldsymbol{V}}$ is a consistent estimator of the covariance matrix of $\hat{\boldsymbol{\beta}}$ if the components of $\boldsymbol{f}$ are homoskedastic and serially uncorrelated, and $\boldsymbol{A}^{-1}$ is proportional to the covariance matrix of the moments.

More generally, the components of $\boldsymbol{f}$ may be heteroskedastic, but still serially uncorrelated. Denote the diagonal covariance matrix of the vector $\boldsymbol{f}$ by $\boldsymbol{\Omega}$. In this case, we can show that the correct form for the asymptotic covariance matrix of $\hat{\boldsymbol{\beta}}$ is

$$\boldsymbol{V}\boldsymbol{X}^\top\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top\boldsymbol{\Omega}\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top\boldsymbol{X}\boldsymbol{V},$$

where we have got rid of the hats in order to avoid notational clutter. Observe first that the matrix $\hat{\boldsymbol{V}}$ is available in the variable `XtWAWtXinv`. The second estimated covariance matrix in the table, appearing under the name of `Heteroskedasticity Consistent Estimate`, is indeed the value of the expression

$$\hat{\boldsymbol{V}}\hat{\boldsymbol{X}}^\top \boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top \hat{\boldsymbol{\Omega}}\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top \hat{\boldsymbol{X}}\hat{\boldsymbol{V}}, \tag{10}$$

where $\hat{\boldsymbol{\Omega}}$ is a diagonal matrix of which the typical diagonal element is $f_t^2(\hat{\boldsymbol{\beta}})$. Chapter 17 of DM gives details concerning this estimator. As indicated by the warning in the table of results, this estimator alone is valid in presence of heteroskedasticity or in cases in which the weight matrix is not proportional to the inverse of the covariance of the moments. The matrix (10) is made available in the ***Ects*** variable `HCCME`.

The first-order conditions for the minimisation of the criterion function (9) can be written as

$$\boldsymbol{X}^\top(\boldsymbol{\beta})\boldsymbol{W}\boldsymbol{A}\boldsymbol{W}^\top \boldsymbol{f}(\boldsymbol{\beta}) = \mathbf{0}. \tag{11}$$

In the internal computations of ***Ects***, use is made of an upper triangular matrix $\boldsymbol{B}$ with the property that $\boldsymbol{B}^\top\boldsymbol{B} = \boldsymbol{A}$. The $l \times k$ matrix $\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{X}}$ can be found, after a `gmmweight` command has been run, in the variable `CG`. Similarly, the $l \times 1$ matrix $\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{f}}$ is saved in `grad`. The following commands, taken from `gmm.ect`, serve to check the first-order conditions.

```
sample 1 2
mat t = CG'*grad
print t
sample 1 3
ols grad CG
```

This follows because the matrix product `CG'*grad` is equal to $\hat{\boldsymbol{X}}\boldsymbol{W}\boldsymbol{B}^\top\boldsymbol{B}\boldsymbol{W}^\top\hat{\boldsymbol{f}}$, which, according to (11), is zero when the algorithm converges. If one wants to perform the check by means of an artificial regression, then regressing `grad` on `CG` will work.

For the other variables generated by a `gmmweight` command, we have the scalars `nobs`, `nreg`, `ninst`, and `niter`, which have their usual meanings, that is, n, k, l, and the number of iterations, respectively. The $k \times 1$ vectors `coef`, `stderr`, and `student` contain $\hat{\boldsymbol{\beta}}$, the standard errors, and the t statistics. The last two are computed using the covariance matrix estimate `vcov`, which is not necessarily the right one to use. If the other estimate, `HCCME`, is preferred, then the standard errors and t statistics have to be computed explicitly. The n–vector `res` is the analogue of the vector of residuals, that is, $\hat{\boldsymbol{f}} \equiv \boldsymbol{f}(\boldsymbol{y}, \hat{\boldsymbol{\beta}})$. The scalar `ssr` is the sum of the squares of the elements of `res`. Finally, the scalar `crit` is the minimised value of the criterion function.

Exercises:

Estimate the model dealt with in `nliv.ect` and `gmm.ect` using a `gmmweight` command, and then construct the matrix $\hat{\boldsymbol{X}}$ explicitly. The matrix $\boldsymbol{B}$ can be computed by the command `mat B = (lowtriang(A))'`. Show that the definitions given above of `CG`, `grad`, `vcov`, `XtWAWtXinv`, and `HCCME` are correct.

Redo all the estimations in `nliv.ect` et `gmm.ect` using only the last 25 observations. For the commands, `nliv`, `gmmweight`, and those in the `ml` family, it is enough to make the declaration `sample 26 50`. For `gmm` and `gmmhess`, it is necessary to extract the relevant blocks of the matrices that contain the variables. In all cases, you will know if your manipulations are correct if you get the same results for everything.

6. Procedures

If you look yet again at the command file `newlogit.ect`, you'll notice that there are several computations that are done many times. We begin by defining the macros `dldF` and `dFde`, which depend in turn on another set of macros, `F`, `e`, and `X`. Subsequently, in the nonlinear estimation routines, for each derivative of the loglikelihood function we call on almost all of these macros. For instance, in the first estimation by `ml`, the expression `dldF*dFde*e` is evaluated four times for each iteration of the algorithm. If we have only a small number of estimations to do, this is no big deal. But, when there are lots of nonlinear estimations to be done, as in a simulation experiment with many replications, it is grossly inefficient to do and redo identical evaluations several times, and computing times are needlessly long.

In version 3 of ***Ects***, there is a mechanism that lets us avoid such uselessly repeated computations. The mechanism is illustrated in the command file `proclogit.ect`, in which we perform some of the estimations in `newlogit.ect` more efficiently. The key is the use the `procedure` command, as follows.

```
procedure logit 4
   def X = arg1*iota + x1*arg2 + x2*arg3 + x3*arg4
   gen lf = lhd
   answer lf
   gen D0 = dldF*dFde*e
   answer D0
   gen D1 = D0*x1
   answer D1
   gen D2 = D0*x2
   answer D2
   gen D3 = D0*x3
   answer D3
end
```

The name of the command, `procedure`, is followed by the name of the procedure, here `logit`, and the number of *arguments* needed, here 4. The arguments, which must be scalars, usually correspond to parameters to be estimated. After the first line, there are several lines, many of which contain ordinary ***Ects*** commands. The end of the definition of the procedure is signalled, as usual, by `end`.

The purpose of the procedure is to replace the following command, as found in `newlogit.ect`,

```
ml lhd
deriv a = dldF*dFde*e
deriv b1 = dldF*dFde*e*x1
deriv b2 = dldF*dFde*e*x2
deriv b3 = dldF*dFde*e*x3
```

by a command that makes use of the procedure:

```
ml logit(a,b1,b2,b3,1)
deriv a = logit(a,b1,b2,b3,2)
deriv b1 = logit(a,b1,b2,b3,3)
deriv b2 = logit(a,b1,b2,b3,4)
deriv b3 = logit(a,b1,b2,b3,5)
end
```

We will now see how this works, and why it lets us avoid repeated computations.

The only effect of a `procedure` command, along with all the lines that follow it up until the final `end`, is to *define* the procedure. None of the ***Ects*** commands found inside the definition is executed at this stage. But later, when the `ml` command is run, the content of the definition is executed at the moment when the expression

```
deriv b1 = logit(a,b1,b2,b3,3)
```

is to be evaluated in the course of the iterations of the nonlinear algorithm used by `ml`. Expressions of this sort are to be evaluated 5 times per iteration, once for the criterion function, of which the ***Ects*** representation is given immediately after the command name `ml`, and 4 times for the four derivatives of the function. Normally, the function itself is evaluated before the derivatives. When this occurs, execution of the commands found in the procedure definition begins. The first of these defines a macro, `X`, that uses 4 variables that have been defined nowhere that we can see, namely, `arg1`, ..., `arg4`. As we might suppose, these are the 4 arguments to the procedure. When it sees the expression `logit(a,b1,b2,b3,1)`, ***Ects*** assigns to the scalar variables `arg1`, ..., `arg4` the values of the arguments `a,b1,b2,b3`.

The main interest in using a procedure is that it serves to evaluate several expressions simultaneously. That is why, in each call to the procedure, we see a fifth argument after the four main ones. This last argument is the index of the expression we want. Thus, the criterion function is the first expression to be evaluated, the derivative with respect to `a` is the second, the derivative with respect to `b1` the third, and so on. In the procedure definition, after the macro, we find a `gen` command that creates a variable `lf`. The value of this variable is given by the macro `lhd`, which represents the criterion function. The following line is something new. It makes use of the `answer` command. This command is used *only* inside a procedure: Elsewhere an attempt to use it would lead to a syntax error. Here, its function is to return one of the expressions that the procedure evaluates. As illustrated by the command

```
answer lf
```

a single argument is needed, which must be the name of a variable that has been created. It may be a scalar, or a vector, or a matrix, but it *must* exist at the moment when the `answer` command is executed. A procedure may contain an arbitrary number of `answer` commands. Each one defines one of the expressions evaluated by the procedure. The variable returned by the first `answer` in the procedure receives an index of 1, the second an index of 2, and so on. In the `logit` procedure, the variable with index 1 is `lf`, which is the evaluation of the criterion function.

Logically, then, the variable with index 2, given by the second `answer` command, should be the derivative with respect to `a`, since this derivative is given by the expression `logit(a,b1,b2,b3,2)`. Indeed, the variable `D0`, which is returned by the second `answer` command, is generated by

```
gen D0 = dldF*dFde*e
```

exactly as the derivative is generated in `newlogit.ect`. So far, we may well have the impression that all this is much ado about nothing. However, the next variable, `D1`, generated by

```
gen D1 = D0*x1
```

does not recompute the expression `dldF*dFde*e`, because it can use the variable `D0` that has already been evaluated. The same applies to the other two derivatives, given by `D2` and `D3`.

If a procedure is really to save computing time, it must be executed only once per iteration of the `ml` nonlinear algorithm. Each time a value is requested from the procedure, the values of the arguments are compared with those in force at the previous call. If the argument values have not changed, then no computation is done, and the procedure simply returns the previously computed variable. But if any of the arguments is different from those of the previous call, then all the commands in the procedure definition are run once more. In nonlinear estimation, when the arguments are the model parameters, the values of these parameters change only at the end of an iteration. Consequently, we have just what we want: Recomputation occurs when needed, and not when not, and we have successfully avoided those repeated computations.

* * * *

The *arguments* of a procedure are evaluated each time that the procedure is called. They are evaluated according to the rules of a `set` command, logically enough, since the arguments must be *scalars*. The final argument, that is, the index of the expression desired, is computed in the same way, and is then interpreted as an *integer*. If this integer is not positive, then a syntax error occurs. Once the final argument is evaluated, if the preceding arguments are fewer in number than the number of arguments prescribed in the procedure definition, there will also be a syntax error. But if there are too many, those given after the necessary ones are thrown away by ***Ects***, with no comment.

* * * *

It is only after all the commands in the procedure definition have been executed that it is possible to see whether an answer exists corresponding to the index requested. If not, an error message is displayed. If the answer does exist, it is returned to the command that called the procedure. If this command is a `set` command, or a command that operates under the same rules, like `procedure` itself when it is evaluating its arguments, only the first element of the answer is used. With `gen` and commands operating under its rules, only the rows of the currently declared sample, as defined by the `sample` command, are modified. With `mat`, the matrix provided by the procedure is used as is.

The next estimation found in `proclogit.ect`, along with the procedure definition that precedes it, illustrates some of the points made in the last paragraph. Here are the commands:

```
procedure argt 4
   sample 1 2
   answer a
   answer b1
   gen bb2 = b2
   answer bb2
   answer b3
end
```

```
mlopg logit(argt(a,b1,b2,b3,1), argt(a,b1,b2,b3,2), \
argt(a,b1,b2,b3,3), argt(a,b1,b2,b3,4),1)
deriv a = logit(a,b1,b2,b3,2)
deriv b1 = logit(a,b1,b2,b3,3)
deriv b2 = logit(a,b1,b2,b3,4)
deriv b3 = logit(a,b1,b2,b3,5)
end
```

The procedure called `argt` is just for the purposes of this illustration, and has no other use. It simply returns its 4 arguments `a`, `b1`, `b2`, and `b3`. It also shows how a procedure can call another procedure.

Given that the `logit` procedure, when it reads its arguments, operates under the rules of `set`, only the first element of the third answer, `bb2`, is retained. In fact, `bb2` has two elements, on account of the command `sample 1 2` that we see in the procedure `argt`. It is important to note that when a `sample` command is used inside a procedure, its effect is purely *local*. Once the procedure is executed, the command is forgotten, and the declared sample reverts to what was in force beforehand. But the effects of all the other commands persist outside the procedure. There is one notable exception to that last statement. It would be ridiculous to execute a `quit` command inside a procedure. If one is encountered, then its only effect is an error message.

We can ascertain that the command `sample 1` inside the procedure really has had its proper effect by means of the following commands:

```
set i = rows(bb2)
show i
```

The `rows` function takes one argument, which must be the name of a variable in existence. The value of the function is the number of rows of the matrix associated with the variable name. Similarly, the `cols` function returns the number of columns of its argument. Used in `set` or `mat`, the values of these functions are scalars; with `gen`, we get a vector of which each element within the declared sample is equal to that scalar value.

Although the `procedure` command yields some significant advantages, it can also lead to some disadvantages, as we can see from the following code, taken from what follows in `proclogit.ect`.

```
set i = 3

procedure logithess i-2
   sample 1 100
   gen lf = lhd
   answer lf
   gen D0 = dldF*dFde*e
   answer D0
   gen D1 = D0*x1
   answer D1
   gen D2 = D0*x2
   answer D2
   gen D3 = D0*x3
   answer D3
   gen D00 = -f
   answer D00
   gen D01 = x1*D00
   answer D01
   gen D02 = x2*D00
   answer D02
   gen D03 = x3*D00
   answer D03
   gen D11 = x1*D01
   answer D11
   gen D12 = x2*D01
   answer D12
   gen D13 = x3*D01
   answer D13
   gen D22 = x2*D02
   answer D22
   gen D23 = x3*D02
   answer D23
   gen D33 = x3*D03
   answer D33
```

```
end

set a = 0
set b1 = 0
set b2 = 0
set b3 = 0
def X = a*iota + x1*b1 + x2*b2 + x3*b3

mlhess logithess(a,1)
deriv a = logithess(a,2)
deriv b1 = logithess(a,3)
deriv b2 = logithess(a,4)
deriv b3 = logithess(a,5)
second a,a = logithess(a,6)
second a,b1 = logithess(a,7)
second a,b2 = logithess(a,8)
second a,b3 = logithess(a,9)
second b1,b1 = logithess(a,10)
second b1,b2 = logithess(a,11)
second b1,b3 = logithess(a,12)
second b2,b2 = logithess(a,13)
second b2,b3 = logithess(a,14)
second b3,b3 = logithess(a,15)
end
```

When we look at the first two lines of this code fragment, we can see that the argument that specifies the number of arguments needed by a procedure is not necessarily a simple numerical constant. This argument, like those corresponding to the index of a requested answer, is evaluated under the rules of `set`, and the value is interpreted as an integer. If this integer is negative, an error message is displayed. There is however nothing wrong with a procedure that takes no arguments, other than that it can never be executed more than once.

The disadvantage we mentioned just above manifests itself in the `mlhess` command. If the derivatives of a loglikelihood function are specified normally, then ***Ects*** is capable of calculating second derivatives as needed. But if they are defined by a procedure, automatic differentiation is no longer possible. In such cases, *all* the second derivatives must be explicitly specified. This can be done, of course, as we see in the above code in the procedure `logithess`, in which the loglikelihood function is generated along with its first and second derivatives. As usual, doing this saves a great many repeated computations.

Three more remarks on the above code. First, we see that the parameters are reinitialised before the nonlinear estimation, in order that the performance of the different commands may be compared directly. If we did not reinitialise, each command would converge in one single iteration, since the parameters would already be equal to the ML estimates. Second, the `logithess` procedure is defined with just one argument. In general, the values of all the parameters vary from one iteration to the next, and so it is unnecessary to take account of more than one of them in order to trigger the recomputation at the start of each iteration. This allows us to simplify the code significantly. We must note, however, that this simplification comes at the cost of our no longer being able to use the variables `arg2`, *etc.*, because only `arg1` is defined. Here, this is of no concern, since `logithess` does not use these variables, preferring to use the parameters `a`, `b1`, *etc* directly. Third, it was necessary to redefine the macro `X`. The definition given inside the procedure `logit` is given in terms of the `arg` variables, which are not used by `logithess`.

We now move on to the last estimation found in `proclogit.ect`. It is carried out using `mlar`. Some more points of general interest are illustrated here:

```
procedure logitar 1
   sample 1 100
   gen D = 1/sqrt(F*(1-F))
   gen Dl = (y-F)*D
   answer Dl
   gen D0 = f*D
   answer D0
   gen D1 = x1*D0
   answer D1
   gen D2 = x2*D0
   answer D2
   gen D3 = x3*D0
   answer D3
   answer D
end
```

We see that it is perfectly possible to generate a variable, here `D`, at the beginning of the procedure, to use it in generating other variables, and then not to declare it as an answer until the end. Thus we have complete liberty as to the ordering of the answers. We could even do all the computations first, and then declare the answers in what seems the most logical order for the user of the procedure.

Then we have

```
equation diffb1 b1 = logitar(a,3)
equation diffb2 b2 = logitar(a,4)
mlar lhd
lhs = logitar(a,1)
deriv a = logitar(a,2)
deriv diffb1
deriv diffb2
deriv b3 = f*x3*logitar(a,6)
end
```

Here we see the use of a new command, `equation`. This command brings with it no new functionality; its purpose is just to simplify the writing of an ***Ects*** program, and also perhaps to clarify its structure. After the command name `equation`, we specify a name that can be used subsequently to refer to the equation that follows, by which we mean anything that can be expressed as

<variable> = *<expression>*

Thus the name `diffb1` is associated with the equation

```
b1 = logitar(a,3)
```

It is an equation of this type that is required by the commands `set`, `gen`, `mat`, `def`, `nls`, `nliv`, and in all the `deriv` or `second` lines of nonlinear estimation commands. Such an equation can also appear in an `equation` command itself. If we want to give one and the same equation two different names, we can do

```
equation name2 name1
```

for instance, if `name1` is already defined.

Exercises:

In the `mlar` command that we have just looked at, we have the following line:

```
lhs = logitar(a,1)
```

This line has the form of an `equation`. Define an appropriate equation, and use it in the `mlar` command in place of `lhs`.

In Chapter 14 of DM there is presented a double-length artificial regression, that is, one that has twice the number of artificial observations as the number of real observations in the original sample. Implement one of the double-length regressions found in that chapter. This exercise should demonstrate clearly just why it is important to allow for different sample sizes inside and outside a procedure.

Finally, observe that the very last command in `proclogit.ect` before `quit` is the simple command `showall`. This command allows us to inspect the current state of all the internal tables maintained by ***Ects***. First we see a list of all the variables in existence at the moment the `showall` command is run. The elements of the list appear as follows (this is just a small excerpt from the full list):

```
Variables currently defined are
Dl:  dimensions 100 x 1
PI: dimensions 1 x 1
R2:  dimensions 1 x 1
TOL: dimensions 1 x 1
XtXinv:  dimensions 4 x 4
```

The names of existing variables are given, along with their matrix dimensions.

* * * *

All ***Ects*** variables are matrices. A scalar is just a 1×1 matrix, and a single variable is an $n \times 1$ vector, for some integer $n > 1$.

* * * *

Next comes a list of the macros defined by the `def` command (again, we show here only one macro):

```
Macros currently defined are
F:
e 1 e +(2) /(2)
```

The first line gives the name of the macro, and the next line gives its internal representation, in the same format as the one given by the `differentiate` command. If you like figuring out riddles, you can have a go at the meaning of this internal representation.

Then comes a list of defined procedures, of which we again give only an excerpt:

```
Procedures currently defined are
argt
logit
```

and, finally, the equations:

```
Equations currently defined are
diffb1
diffb2
```

* * * *

If you really do want to study the internal representation of expressions, there exists a command `expand` that displays on demand the representation of a single macro. Thus if you do

```
def X = a + b1*x1 + b2*x2
expand X
```

Ects displays

```
X is:
a b1 x1 *(2) +(2) b2 x2 *(2) +(2)
```

This command is among those that are more useful to the developers of ***Ects*** than to its users.

* * * *

Everything that appears in the lists displayed by `showall` can be deleted from memory by the `del` command. Current computers very often have vast amounts of RAM, but, on older machines that have less, it is sometimes necessary to perform a cleanup using `del`.

Exercises:

Append a `del` command at the end of the file `proclogit.ect`, for example

```
del diffb1 lhd logit
```

and rerun `showall`. You will see that the objects on `del`'s list are no longer recognised by ***Ects***.

Chapter 3
Simulation

1. Recursive Simulation

The enormous power of modern computing machinery has led simulation to become a tool of great importance for many disciplines, econometrics not least among them. Monte Carlo experiments are discussed in Chapter 6 of Man2, and these are simulation based. In this chapter, I consider the numerous new possibilities offered by version 3 of ***Ects*** for implementation of simulations of many kinds.

Virtually all simulations make use of a **loop**, which serves to perform a series of operations many times, with different realisations of the random elements each time. ***Ects*** operates by reading each line of a command file and then executing the command found on that line[3]. This means that the execution if a loop is unavoidably slower that if one had a similar loop in a C++ program, or indeed a program written in any decent low-level language. In the first versions of ***Ects***, the commands in a loop were read and reread each time the loop was executed, thereby aggravating the intrinsic slowness. With version 3, loop execution is considerably faster than with earlier versions, since the contents of a loop are now read only once, but a good deal of slowness relative to low-level languages remains.

Even outside a simulation context, loops are sometimes needed for the purpose of generating a series *recursively*. boucles pour générer des séries de manière *récursive*. The simplest example is that of an AR(1) series, that is, a series generated by an autoregressive process of order 1. This type of series is dealt with in Chapter 6 of Man2. The defining equation for the AR(1) series y_t is

$$y_t = \rho y_{t-1} + u_t,$$

where u_t is white noise and ρ is a parameter of which the absolute value is less than 1.

Let's look at the command file `argen.ect`. There we find four different ways to generate an AR(1) series. The first is the one proposed in Man2:

[3] Not true with version 4, but explicit loops still slow down execution.

```
set N = 1000
set NREP = 100
set rho = 0.5
sample 1 N
gen u = random()
set i = 0

while i < NREP
   set i = i+1
   gen t = time(-1)
   gen rhot = rho^t
   gen y = conv(rhot, u)
end
```

This program fragment creates an AR(1) series with 1,000 observations for $\rho = 0.5$. The loop, which causes the same code to be executed 100 times, serves only to measure computing time. On a Pentium 166, this loop takes around 5 seconds with version 3, against 15 seconds with version 2. The speedup due to reading the loop only once is quite evident.

The method laid out above is not perhaps completely transparent, on account of the use of the `conv` function. A more naive method would be to use an explicit loop in place of that function:

```
set NREP = 20
set i = 0
while i < NREP
   set i = i+1
   set j = 1
   set y(1) = u(1)
   while j < N
      set j = j+1
      set y(j) = rho*y(j-1) + u(j)
   end
end
```

This time, the generating procedure is repeated only 20 times, but computing time is already up to 30 seconds. Thus this computation takes about 30 times as long as the first method!

Exercises:

Check that the two methods do indeed generate exactly the same series. Waste no time in doing so: do `set NREP = 1` before running the program.

It is unfortunately not always possible to find a trick like the one used in the first method for performing a recursive simulation. What we would really like is to be able to write a command like

```
gen y = rho*lag(1,y)+u
```

However, this command is not implemented recursively. If the variable `y` exists before the command is run, it is first lagged one period by the `lag` function. The result is then multiplied by `rho` and added to `u`. If the variable has not been previously defined, then all we get is a syntax error. In either case, there is no recursive operation.

The result we want can be obtained by running the command inside a `recursion` command. To illustrate, in the simplest case, we would have

```
set NREP = 100
set i = 0
while i < NREP
   set i = i+1
   recursion
   gen yy = rho*lag(1,yy)+u
   end
end
```

Computing time is now around 80 seconds, or twice as fast as the explicit loop, but still much slower than the method using `conv`. The conclusion from this is clear: If we can find a trick that avoids an explicit loop or the `recursion` command, then we should use it.

Like all other multiline commands, `recursion` must be explicitly terminated by `end`, because – as we will see in due course – we can have several `gen` commands inside one `recursion`. This feature is very useful in cases in which we cannot find a way to avoid explicit looping.

2. ARMA Processes

In the analysis of **time series**, the most frequently encountered processes are **ARMA processes**. Such processes are treated briefly in Chapter 10 of DM, and much more fully in Hamilton (1994). An ARMA(p, q) process can be characterised as follows:

$$y_t = \sum_{i=1}^{p} \rho_i y_{t-i} + u_t + \sum_{j=1}^{q} \alpha_j u_{t-j}, \tag{12}$$

where the series u_t is white noise. The ρ_i and the α_j, along with the variance σ^2, are the parameters of the process.

It is convenient to denote an ARMA(p, q) process in terms of two polynomials, A and B, as follows:

$$A(L)y_t = B(L)u_t, \tag{13}$$

where A and B are defined as

$$A(x) = 1 - \rho_1 x - \rho_2 x^2 - \ldots - \rho_p x^p = 1 - \sum_{i=1}^{p} \rho_i x^i, \text{ et}$$

$$B(x) = 1 + \alpha_1 x + \alpha_2 x^2 + \ldots + \alpha_q x^q = 1 + \sum_{j=1}^{q} \alpha_j x^j.$$

The notation L signifies the **lag operator**, which can be defined by the equation $Ly_t = y_{t-1}$. Similarly, $L^i y_t = y_{t-i}$. It is easy to check that (12) and (13) are equivalent.

An AR(p) process is a specialisation of an ARMA(p, q) process, for which $q = 0$. It is defined by an equation of the form $A(L)y_t = u_t$. In the same way, an MA(q) process is an ARMA with $p = 0$, defined by the equation $y_t = B(L)u_t$. Simulating an MA(q) presents no difficulty, since no recursion is needed. It is enough to do

```
gen u = random()
gen y = u + a1*lag(1,u) + ... + aq*lag(q,u)
```

There does exist another more complicated way ot doing the same thing, which we present here, not because it is in any way preferable for the current problem, but because it lets us do quite interesting things in other circumstances.

The following program, in which we generate an MA(3) process using a white noise process bruit blanc `u` with variance 1, illustrates the two approaches.

```
sample 1 20
set a1 = 0.2
set a2 = 0.3
set a3 = -0.1
gen u = random()
gen y = u + a1*lag(1,u) + a2*lag(2,u) + a3*lag(3,u)
lagpoly 1 a1 a2 a3
gen yy = polylag(u)
```

The second approach makes use of the command `lagpoly`.[4] The name of the command is inspired by *lag operator polynomial*. The polynomial $B(L)$ of the MA(3) process is here $B(L) = 1 + \texttt{a1}L + \texttt{a2}L^2 + \texttt{a3}L^3$. The command

```
lagpoly 1 a1 a2 a3
```

serves to store this polynomial in memory.

[4] Note for version 4 : This command does not at present exist in version 4. It is my intention to incorporate it in a **loadable module** along with a richer set of commands than those in version 3.

* * * *

There is a slight danger of falling into a trap when defining polynomials with `lagpoly`. To illustrate it, consider this example:

```
lagpoly 1 0.2 0.3 -0.1
```

which would be intereted by ***Ects*** as

```
lagpoly 1 0.2 0.2
```

since, indeed, $0.3 - 0.1 = 0.2$. The problem can be circumvented either by using named variables, or macros, in place of explicit numbers, or else by separating the arguments by commas. Thus the command

```
lagpoly 1, 0.2, 0.3, -0.1
```

or even

```
lagpoly 1 0.2 0.3, -0.1
```

works just fine.

* * * *

In order to use the polynomial stored in memory by `lagpoly`, one uses the function `polylag`. This function is available only inside a `gen` command, not in `set` or `mat` commands, where its use would lead to a syntax error. `polylag` takes just one argument, here the series `u`. The value of the function can be written as

$$\texttt{polylag}(u_t) = \texttt{a0}u_t + \texttt{a1}u_{t-1} + \texttt{a2}u_{t-2} + \texttt{a3}u_{t-3},$$

which is just the series denoted by $B(L)u$.

* * * *

The *arguments* of `lagpoly` are parsed as though in a `mat` command, for a reason given shortly.

* * * *

For an AR(p) process, we have $A(L)y_t = u_t$. Formally, this becomes $y_t = \big(A(L)\big)^{-1}u_t$. The constant term of the polynomial A is always 1; thus we can write $A(L) = 1 + a(L)$. The binomial theorem lets us obtain an explicit expression for $\big(A(L)\big)^{-1}$:

$$\big(A(L)\big)^{-1} = \sum_{k=0}^{\infty} \big(a(L)\big)^k. \tag{14}$$

In this equation, it must be the case that $(a(L))^0 = 1$. It is therefore possible to evaluate the right-hand side of (14) explicitly. This leads to a result of the form

$$\big(A(L)\big)^{-1} = 1 + \sum_{i=1}^{\infty} \alpha_i L^i. \tag{15}$$

The apparently infinite sum in the above expression is not so in practice, because, for $i > n$, n being the sample size, $L^i y_t = 0$ for all $t = 1, \ldots, n$ and for any vector y_t. Thus the computation of the right-hand side of (15) is a simple algebraic exercise, that can be solved in various ways.

For an ***Ects*** user, the easiest of these ways is to hand the task over to ***Ects*** itself. To do so, we can use the command `invertlagpoly`, of which the syntax is identical to that of `lagpoly`. Thus if we do

```
invertlagpoly 1 0.2 0.3, -0.1
```

this serves to store in memeory the inverse of the polynomial $A(L) \equiv 1{+}0,2L{+}0,3L^2 - 0.1L^3$. Of course, the infinite sum in equation (15) is truncated after the first `smplend` terms. If later on we do

```
gen y = polylag(u)
```

then the series `y` satisfies the autoregression

$$y_t = -0.2y_{t-1} - 0.3y_{t-2} + 0.1y_{t-3} + u_t.$$

Now in order to initialise such an autoregression, we need the values of the first three elements of the vector `y`. If we do

```
sample 1 20
invertlagpoly 1 0.2 0.3, -0.1
gen u = random()
gen y = polylag(u)
```

we have implicitly set $y_0 = y_{-1} = y_{-2} = 0$. On the other hand, if we do

```
sample 1 20
gen y = 0
set y(1) = 3
set y(2) = 4
set y(3) = 5
lagpoly 1 0.2 0.3, -0.1
gen u = polylag(y)
sample 4 20
gen u = random()
sample 1 20
invertlagpoly 1 0.2 0.3, -0.1
gen y = polylag(u)
```

explicitly setting the first three components of `y`, and starting the random part of the generating process only at the fourth element, we obtain the desired result. Note that specifying the first three elements of `y` implicitly serves to define the first three elements of `u`. To make the implicit definition explicit, we can use the `lagpoly` command to generate `u` as a function of `y`. Then we generate the unspecified random elements of `u` starting from the fourth component. All this can be checked by the commands

```
gen uu = y + 0.2*lag(1,y) + 0.3*lag(2,y) - 0.1*lag(3,y)
print u uu
```

We should find that the vectors `u` and `uu` are identical.

Exercises:

Use equation (14) to evaluate the inverse of the polynomial $A(L)$ with coefficients $1, 0.2, 0.3, -0.1$ directly. If necessary, ***Ects*** can help you multiply polynomials. The inverse polynomial can be represented by a series of 20 éléments, which are the first 20 coefficients. Let this series be denoted by `r`. Then show that the command

```
gen y = conv(r,u)
```

generate the same series as the one generated by the `polylag` command.

An ARMA(p, q) series can be generated in two steps. First, we generate the MA part, that is, the series $B(L)u_t$. Let `b1`$, \ldots,$ `bq` be the coefficients of the polynomial $B(L)$. After obtaining the MA part, we can obtain the full ARMA series by applying the inverse of the polynomial $A(L)$. Let `a1`$, \ldots,$ `ap` be the coefficients of $A(L)$. Then we have

```
gen u = random()
lagpoly 1 b1 ... bq
gen u = polylag(u)
invertlagpoly 1 a1 ... ap
gen u = polylag(u)
```

Alternatively and sometimes more simply, we may generate the MA part directly, without using the `lagpoly` function. This would give

```
gen u = u + b1*lag(1,u) + ... + bp*lag(p,u)
```

in place of the `lagpoly` command and the `gen` command immediately following.

A simple AR(1) series can of course be created using the command `invertlagpoly`. At the end of the command file `argen.ect`, we find the following two commands:

```
invertlagpoly 1, -rho
gen y = polylag(u)
```

In the first implementations of `invertlagpoly`, the above code took a very long time to execute, around 15 seconds. However, this was due to what turned out to be a grossly inefficient implementation. It now executes almost as fast is the method that uses the `conv` function.

3. ARMAX and VAR Processes

In econometrics, it is unusual for a univariate model[5] to have only lags of the dependent variable as explanatory variables. A more frequently encountered

[5] A model is called **univariate** if the dependent variable for each observation is a scalar. Models in which the dependent variable is a vector are called **multivariate**, or **bivariate** if there are just two components.

case is to have a model of the form

$$y_t = \boldsymbol{X}_t\boldsymbol{\beta} + \sum_{i=1}^{p} \rho_i y_{t-i} + u_t + \sum_{j=1}^{q} \alpha_q u_{t-q},$$

where the row vector $\boldsymbol{X}_t$ contains exogenous explanatory variables. Such a model defines an **ARMAX process**, or, more precisely, an ARMAX(p, q) process. The 'X' corresponds to the notation $\boldsymbol{X}$ for the matrix of exogenous variables.

For the purposes of numerical simulation, there is no need to distinguish the exogenous term $\boldsymbol{X}_t\boldsymbol{\beta}$ from the random terms $u_t + \ldots$. The process can thus be simulated as follows:

```
gen u = random()
lagpoly 1 a1 ... aq
gen u = polylag(u)
gen Xu = x1*beta1 + ... + xk*betak + u
invertlagpoly 1 rho1 ... rho2
gen y = polylag(Xu)
```

If $p = 0$, the process can be simulated directly without use of the commands `lagpoly` and `invertlagpoly`. If $p = 1$, a much more rapid computation can be obtained by use of the function `conv`:

```
gen u = random()
gen u = u + a1*lag(1,u) + ... + aq*lag(q,u)
gen Xu = x1*beta1 + ... + xk*betak + u
gen t = time(-1)
gen rhot = rho^t
gen y = conv(rhot,Xu)
```

Exercises:

Generate an ARMAX(p, q) series according to the procedure described above, and check that it gives the correct result by comparing the random vector u_t with the expression $y_t - \sum_{i=1}^{p} \rho_i y_{t-i} - \boldsymbol{X}_t\boldsymbol{\beta}$.

An estimation procedure used a good deal in econometrics makes use of the concept of a **vector autoregression**, or **VAR**. A VAR can be written down as follows:

$$\boldsymbol{Y}_t = \boldsymbol{\alpha} + \sum_{i=1}^{p} \boldsymbol{A}_i \boldsymbol{Y}_{t-i} + \boldsymbol{U}_t, \tag{16}$$

where $\boldsymbol{Y}_t$ is an $m \times 1$ vector, as also are the vector $\boldsymbol{\alpha}$ of constants and the random vector $\boldsymbol{U}_t$. The matrices $\boldsymbol{A}_i$ are square $m \times m$ matrices. It would be possible to add to the right-hand side a term $\boldsymbol{B}\boldsymbol{X}_t$, where $\boldsymbol{X}_t$ is a $k \times 1$ vector of exogenous variables and $\boldsymbol{B}$ is an $m \times k$ matrix of parameters. Such a model is however rarely used in practice, except for cases in which $\boldsymbol{X}_t$ contains nothing but purely deterministic variables, like seasonal dummies or time trends.

The parameters of a VAR model are actually easy to estimate, since all that is needed is OLS. This is illustrated in the command file `var.ect`. We use the data in the data file `ols.dat`, which provides 100 observations on 4 variables. A VAR model with these 4 variables and 3 lags ($p = 3$) can be estimated by use of the following commands:

```
sample 1 100
read ols.dat y1 y2 y3 y4
gen Y = colcat(y1,y2,y3,y4)
gen Y1 = lag(1,Y)
gen Y2 = lag(2,Y)
gen Y3 = lag(3,Y)
ols Y c Y1 Y2 Y3
```

It is worth recalling here that the command `ols` accepts a matrix of several columns as the dependent variable, here the matrix `Y` of which the 4 columns are the 4 variables of the model. Each column of `Y` is regressed on a constant and the first three lags of the four variables, for a total of 13 regressors.

If we wish, we can then extract the vector $\hat{\boldsymbol{\alpha}}$ and the three matrices $\hat{\boldsymbol{A}}_i$, $i = 1, 2, 3$, from the complete matrix `coef` of estimated parameters.

```
mat alpha = coef(1,1,1,4)
mat A1 = coef(2,5,1,4)
mat A2 = coef(6,9,1,4)
mat A3 = coef(10,13,1,4)
```

A VAR model can be expressed in terms of a **matrix polynomial** in the lag operator. The polynomial corresponding to the model (16) can be written as

$$\boldsymbol{A}(L) \equiv \mathbf{I} - \sum_{i=1}^{p} \boldsymbol{A}_i L^i,$$

so that (16) becomes

$$\boldsymbol{A}(L)\boldsymbol{Y}_t = \boldsymbol{\alpha} + \boldsymbol{U}_t. \tag{17}$$

In order to simulate a VAR process, we can once more use the command `lagpoly`. In `var.ect` we find the following commands:

```
gen iota = 1
mat Alpha = iota*alpha
mat U = Alpha+res
mat I = A1^0
lagpoly I, -A1, -A2, -A3
gen UU = polylag(Y)
mat tt = UU-U
mat tt = tt'*tt
show tt
showlagpoly
```

The rows of the matrix `U`, transposed, represent the right-hand side of (17). Note also that each row of the matrix product `Alpha`, of dimension 100×4, is equal to $\hat{\boldsymbol{\alpha}}$ transposed. Then, so as to get an identity matrix quickly, we use the fact that any 4×4 matrix raised to the power 0 is a 4×4 identity matrix. If you run `var.ect`, you should see that all the elements of the matrix `tt` are zero. This demonstrates that the matrix `UU`, created by `polylag`, and the matrix `U` are identical.

* * * *

We can now see why the arguments of the commands `lagpoly` and `invertlagpoly` are parsed as though in a `mat` command: It is this that lets us use matrix polynomials.

* * * *

The inverse operation, which allows us to simulate the matrix $\boldsymbol{Y}$, is more useful in practice. In fact, what we have to do is just to use `invertlagpoly` in place of `lagpoly`.

```
invertlagpoly I, -A1, -A2, -A3
gen YY = polylag(U)
mat tt = YY-Y
mat tt = tt'*tt
show tt
showlagpoly
```

Once more all the elements of `tt` are zero. We can thus reproduce `Y` exactly from the matrix `U` that contains the constants and the random elements. It follows from this observation that the VAR process defined by the parameters $\hat{\boldsymbol{\alpha}}$ and $\hat{\boldsymbol{A}}_i$ can be simulated by replacing the matrix `res` by a random matrix, which could be generated by `random`, for example.

After each one of the commands `show tt`, we find the command `showlagpoly`. In truth, this command exists more for the purposes of the programmer than for those of the user of ***Ects***, but it can be used here in order to see the nature of the lag operator polynomials generated by `lagpoly` and `invertlagpoly`, whether scalar or matrix polynomials. The polynomial stored in memory by the command `lagpoly` is represented in the output file as follows:

```
The number of terms in the lag polynomial is 4:

At lag 0:
  1.000000    0.000000    0.000000    0.000000
  0.000000    1.000000    0.000000    0.000000
  0.000000    0.000000    1.000000    0.000000
  0.000000    0.000000    0.000000    1.000000
At lag 1:
  0.227360   -0.038240   -0.070657   -0.064302
 -1.327870   -0.526401   -0.049297   -0.211917
 -1.272313   -0.227526    0.263922   -0.369891
 -1.273761    0.022772    0.190848   -0.654683
```

```
At lag 2:
  0.348915    0.046788   -0.049145    0.016212
  1.066620    0.277487    0.028368    0.168480
  0.254965    0.213882    0.012838   -0.116361
 -0.127309    0.013068    0.123233    0.542982
At lag 3:
 -0.015650    0.032638    0.017103   -0.014088
 -0.969398   -0.278550   -0.038755   -0.093459
 -0.592363    0.085799    0.173043   -0.164176
  0.174623    0.068046   -0.000404   -0.256209
```

As we would expect, the polynomial has four terms, the first being an identity matrix, and the other three the matrices $\boldsymbol{A}_i$, $i = 1, 2, 3$. The polynomial generated by `invertlagpoly`, on the other hand, is much more complicated, since, mathematically speaking, it has an infinite number of terms. Fortunately, we can forget about those terms that would have no effect until we are past the end of the sample, here of size 100. This fact is stated explicitly in the output file:

```
The number of terms in the lag polynomial is 100:

At lag 0:
  1.000000    0.000000    0.000000    0.000000
  0.000000    1.000000    0.000000    0.000000
  0.000000    0.000000    1.000000    0.000000
  0.000000    0.000000    0.000000    1.000000
At lag 1:
 -0.227360    0.038240    0.070657    0.064302
  1.327870    0.526401    0.049297    0.211917
  1.272313    0.227526   -0.263922    0.369891
  1.273761   -0.022772   -0.190848    0.654683
```

In the output file, the list continues inexorably all the way through the 100 matrices. We can see, though, that these matrices finally tend to zero. For instance, we see

```
At lag 43:
 -0.000000   -0.000000   -0.000000   -0.000000
 -0.000000    0.000000    0.000000   -0.000000
  0.000000   -0.000000   -0.000000    0.000000
  0.000000    0.000000    0.000000    0.000000
```

and, from the 43rd lag up to the 100th, all the matrices are zero. This is due to the fact that the matrix polynomial that is inverted is *stationary*: in other cases, the successive matrices could tend to infinity.

Exercises:

Invert some simple scalar polynomials by use of the command

```
invertlagpoly 1 rho
```

for different values of `rho`. Can you characterise the values that give rise to stationary polynomials, where the terms tend to zero, and those that give rise to explosive terms?

We have seen that the commands `lagpoly` and `invertlagpoly` work correctly by checking that the series contained in the matrix `Y` can be reproduced starting from the residuals. In a more realistic simulation, there is another aspect of the problem to consider. Even if the residuals are serially uncorrelated, we may well expect to find that the m residuals in the vector $\boldsymbol{U}_t$ are correlated among themselves. We call such a phenomenon **contemporaneous correlation** of the residuals.

* * * *

Contemporaneous correlation is explicitly accounted for by the procedure called SUR estimation, for Seemingly Unrelated Regressions: see section 3.3 of Man2 and Chapter 9 of DM.

* * * *

The contemporaneous covariance matrix can be estimated by the $m \times m$ matrix $\hat{\boldsymbol{\Sigma}}$, given by

$$\hat{\boldsymbol{\Sigma}} = n^{-1}\hat{\boldsymbol{U}}^\top\hat{\boldsymbol{U}},$$

where $\hat{\boldsymbol{U}}$ is the $n \times m$ matrix of residuals from the vector regression (16). In order to simulate a DGP contained in the model (16), we need to generate a matrix $\boldsymbol{U}^*$ of simulated disturbances, of which each row $\boldsymbol{U}_t^*$ is a drawing from the multivariate normal distribution $\mathrm{N}(\mathbf{0}, \hat{\boldsymbol{\Sigma}})$.

* * * *

Here and henceforth the star $(^*)$ indicates a simulated quantity.

* * * *

How can we do that? Let $\boldsymbol{A}$ be an $m \times m$ matrix such that

$$\boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{\Sigma}. \qquad (18)$$

If the $m-$vector $\boldsymbol{z}$ is a drawing from the $\mathrm{N}(\mathbf{0}, \mathbf{I})$ distribution, we may calculate as follows:

$$\mathrm{Var}(\boldsymbol{A}\boldsymbol{z}) = \mathrm{E}(\boldsymbol{A}\boldsymbol{z}\boldsymbol{z}^\top\boldsymbol{A}^\top) = \boldsymbol{A}\mathrm{E}(\boldsymbol{z}\boldsymbol{z}^\top)\boldsymbol{A}^\top = \boldsymbol{A}\mathbf{I}\boldsymbol{A}^\top = \boldsymbol{A}\boldsymbol{A}^\top = \boldsymbol{\Sigma}.$$

Thus $\boldsymbol{A}\boldsymbol{z}$ is a drawing from the $\mathrm{N}(\mathbf{0}, \boldsymbol{\Sigma})$ distribution.

The first step, then, is to find a matrix $\boldsymbol{A}$ satisfying (18). In Man2 it is explained that the function `uptriang` does what is needed: the command

```
mat A = uptriang(Sigma)
```

creates an upper-triangular matrix `A` that satisfies the relation (18). In version 3.3 of ***Ects***, another function, `lowtriang`, is available, identical to `uptriang` except that the generated matrix is lower-triangular. The following commands, taken from `var.ect`, illustrate the use of these two functions:

```
mat Sigma = (res'*res)/100
mat B = uptriang(Sigma)
mat A = lowtriang(Sigma)
show A B
mat M1 = B*B'
mat M2 = A*A'
show Sigma M1 M2
```

If you run `var.ect`, you will see that `A` and `B` have the required triangularity properties, and that the matrices `M1` and `M2` are equal to the estimated covariance matrix `Sigma`.

Now that we have a suitable matrix $\boldsymbol{A}$, the simulation takes only three lines of code:

```
gen Us = colcat(random(), random(), random(), random())
mat Us = Us*A'
gen Ys = polylag(Alpha+Us)
```

The 100×4 matrix `Us`, is generated by four calls to the `random` function. Next, the matrix is postmultiplied by `A'`, the transpose of $\boldsymbol{A}$. Finally, the simulated matrix `Ys` is generated by the `polylag` function.

Exercises:

Why must we transpose the matrix $\boldsymbol{A}$ in the simulation?

How can we generate triangular matrices $\boldsymbol{A}$ and $\boldsymbol{B}$, upper and lower, such that $\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{B}^\top\boldsymbol{B} = \boldsymbol{\Sigma}$?

Generate graphs that let you compare the 4 simulated series with the four original series found in the data file `ols.dat`. What is the main difference between the simulated and original set of series? Can you explain this difference?

4. ARCH and GARCH Processes

It is impossible here to do justice to the ARCH and GARCH processes that are used extensively in econometrics nowadays, especially in financial econometrics. Readers can refer to Chapter 16 of DM and the very abundant literature cited there. The formal definition of an ARCH(p) process is as follows:

$$u_t = h_t^{1/2} v_t, \quad v_t \sim \text{IID}(0,1), \quad h_t = \sigma_t^2 = \alpha + \sum_{i=1}^{p} \gamma_p u_{t-i}^2. \tag{19}$$

Starting from a white noise process v_t, we construct a series u_t in such a way that the variance σ_t^2 of u_t is a function of the first p lags of u_t. This construction is what gave rise to the acronym **ARCH**, which stands for *Auto-Regressive Conditional Heteroskedasticity*.

* * * *

A **white noise** process is one of which the elements are mutually independent, of expectation zero, and homoskedastic, that is, all with the same variance σ^2. A white noise process is often normal, or Gaussian, especially in simulation contexts, but this is not required.

* * * *

In the file `archdemo.ect` we find several methods for generating an ARCH(1) process. The first makes use of the `recursion` command, which we saw already in the context of the generation of AR(p) series. After the following preliminary commands,

```
set n = 100
sample 1 n
set gamma = 0.4
set alpha = 1
```

which serve to define the sample size and the parameters α and γ of the process, we begin by initialising the series h_t, the white noise process v_t, and the process u_t that will be our ARCH(1) series:

```
gen h = 1
set h(1) = alpha/(1-gamma)
gen v = random()
gen u = 0
set u(1) = sqrt(h(1))*v(1)
```

The command `set h(1) = alpha/(1-gamma)` is not strictly necessary to our implementation. It is nonetheless quite reasonable, since it sets `h(1)` equal to the unconditional expectation of the series h_t, assuming that this series is stationary.

* * * *

See Chapter 16 of DM for more on this point

* * * *

Next we proceed to the recursion corresponding to (19), in a quite explicit manner:

```
sample 2 n
recursion
   gen h = alpha + gamma*(lag(1,u))^2
   gen u = sqrt(h)*v
end
```

Here we can see one of the major advantages of the `recursion` command: It may contain an arbitrarily great number of `gen` commands, which are applied recursively, in the order in which they appear. It is important to note that *only* the `gen` command is allowed inside a `recursion`. Anything else provokes an error message. It is also important to note the effect of a sample size declaration prefomed by the `sample` command. In order not to undo the initialisation of `h` and `u`, we have to start the recursion at the second observation.

The way that `gen` works inside a `recursion` is quite different from the usual way. It begins with observation `smplstart`, that is, the first observation of the sample declared by `sample`. Then each `gen` command is applied to that single observation, as though we were in a `set` command. Having run through all the `gen` commands contained in the `recursion`, it passes to the next observation, and repeats all of the `gen` commands, and so on until getting to observation `smplend`, the last observation of the declared sample. It is on account of this way of working that the `lag` function, so essential to a recursion, finds the correct values. If we tried to run a recursion backwards, starting from the last observation, and using leads instead of lags, we would not get what we wanted, since the observations are evaluated in order from `smplstart` to `smplend`.

GARCH(p, q) processes are defined by the equation

$$u_t = h_t^{1/2} v_t, \quad v_t \sim \mathrm{IID}(0,1), \quad h_t = \sigma_t^2 = \alpha + \sum_{i=1}^{p} \gamma_i u_{t-i}^2 + \sum_{j=1}^{q} \delta_j h_{t-j}.$$

As with simple ARCH processes, we start with a white noise process v_t, that is then multiplied by the standard deviation $h_t^{1/2}$, which is now defined in a more complicated manner. In fact, $h_t$ñow depends not only on the past of the u_t process, but also on its own past. Note that an ARCH(p) is also a GARCH$(p, 0)$. In fact the 'G' of GARCH denotes *Generalised*.

Exercises:

In the above program, the series `u` was initialised to have exactly 100 elements. What would be the consequences if the series were initialised with a smaller or a greater number of elements? Show first that, if there are fewer elements, the missing ones are created, and that, if there are more, those that are not in the declared sample remain unchanged.

Write a program for generating an ARCH(p) process for $p = 3$, and a GARCH(p, q) process with $p = 3$, $q = 4$. Take care to initialise correctly, and to start the recursion at the correct observation after the initialisation.

It is quite possible to generate a (G)ARCH process by means of an explicit loop. Such a loop is in fact necessary with older version of ***Ects***, in which the `recursion` command did not exist. For an ARCH(1) process, you can find in the command file `archdemo.ect` some code similar to what follows:

```
sample 1 n
gen v = random()
gen h = 0
gen u = 0
set h(1) = alpha/(1-gamma)
set u(1) = sqrt(h(1))*v(1)
set j = 2
while j < n
     set h(j) = alpha + gamma*(u(j-1))^2
     set u(j) = sqrt(h(j))*v(j)
     set j = j+1
end
```

The logical structure is identical to that used in the code with a `recursion`. Here though, we must define the series `h` and `u` (although their contents do not matter) for the full sample before embarking on the loop. If we did not, the `set` commands would have no effect outside whatever sample was in effect when `h` and `u` were first generated.

Exercises:

Just as we did for the different ways of generating an AR(1) process, set up loops that generate an ARCH(1) process many times, first using a `recursion`, and then with an explicit loop, in order to compare execution times. The advantage of the `recursion` over the explicit loop is greater for the ARCH(1) than for the AR(1) process.

For ARCH(1) processes, there is a trick that can be used to generate then very fast. Sadly the trick applies only to ARCH(1) and does not generalise. It is found in `archdemo.ect`, where it is the third generating method in the command file. The relevant code is this:

```
sample 1 n
gen v = random()
gen x = gamma*v*v
set x(1) = x(1)/(1-gamma)
gen b = lag(1,x)
set b(1) = 1
gen tmp = product(1/b)
gen h = alpha*sum(tmp)/tmp
set h(1) = alpha/(1-gamma)
gen u = sqrt(h)*v
```

Note the use of the `product` function, available for the first time in version 3.3. Its syntax is identical to that used by `sum`, but a product is computed rather than a sum.

* * * *

Although there is no reason to use older versions of ***Ects***, you can get the result of the command

```
gen tmp = product(1/b)
```

by the command

```
gen tmp = exp(sum(-log(b)))
```

Since all the elements of the series `b` are positive, the logarithms exist. But, in other applications, it could be necessary to make sure that there are no negative elements.

* * * *

Exercises:

Use the `product` function to compute the factorials of the integers from 1 to 10.

The program above that generates an ARCH(1) can be used as a black box. The distinctly tricky calculations that follow are only for those who wish to understand in detail how and why it works. From the definition (19), for the case with $p = 1$, we can see that

$$h_t = \alpha + \gamma u_{t-1}^2 = \alpha + \gamma h_{t-1} v_{t-1}^2.$$

If we solve this relation recursively, we find

$$\begin{aligned} h_2 &= \alpha + \gamma h_1 v_1^2, \\ h_3 &= \alpha + \alpha\gamma v_2^2 + \gamma^2 v_2^2 v_1^2 h_1, \\ h_4 &= \alpha + \alpha\gamma v_3^2 + \alpha\gamma^2 v_3^2 v_2^2 + \gamma^3 v_3^2 v_2^2 v_1^2 h_1, \ldots \end{aligned}$$

For $t > 2$, let x_t be equal to γv_t^2. For $t = 1$, $x_1 = \gamma v_1^2 h_1/\alpha$. It follows that, for $t > 1$,

$$h_t = \alpha \left(1 + \sum_{s=1}^{t-1} \prod_{u=0}^{s-1} x_{t-1-u} \right).$$

WE can simplify this expression by adopting the rule that any product of no factors at all is equal to 1. For $t > 1$, then, we obtain

$$h_t = \alpha \sum_{s=0}^{t-1} \prod_{u=t-s}^{t-1} x_u,$$

and this, after some more algebraic manipulations, can be written as

$$h_t = \alpha \left(\prod_{u=1}^{t-1} \frac{1}{x_u} \right)^{-1} \sum_{s=1}^{t} \prod_{u=1}^{s-1} \frac{1}{x_u}.$$

In the ***Ects*** program, the series `x` corresponds to x_t. Consequently the series `tmp` represents $\prod_{u=1}^{t-1}(1/x_u)$, in the sense that the element t of `tmp` is equal to this product, the first element being just equal to 1. It is important that the first element of the series `b`, that is, `x` lagged, should be equal to 1, in order to avoid divisions by zero if we kept the default value of 0. Now that `tmp` is defined, we can see that the element t of `sum(tmp)` is

$$\sum_{s=1}^{t} \texttt{tmp}_s = \sum_{s=1}^{t} \prod_{u=1}^{s-1} \frac{1}{x_u}.$$

It is now not too hard to see that `alpha*sum(tmp)/tmp` represents h_t, except for the first element. But since we know that that element is just `alpha/(1-gamma)`, we can just set this value explicitly.

* * * *

In the command file `archdemo.ect`, there is yet another method for generating an ARCH(1) process, but you are advised not to use it. Its presence is in fact just to serve as a horrid warning. Many things that are possible are not very useful!

* * * *

EXERCISES:

Compare the computing time of the method we have described with the recursion procedure. You will see that the trick, even if it not very elegant, is very efficient.

5. Resampling and the Bootstrap

The **bootstrap** is a very general method which enables us to perform statistical inference based on simulations. The idea at the root of the method is that, if we know the distribution of a test statistic only approximately, under the null hypothesis being tested, we can often obtain a better approximation by doing simulations under the null hypothesis. An introduction to the bootstrap can be found in Efron and Tibshirani (1993).[6] We will now consider examples that

[6] I no longer recommend this book. Even when it was published, it was not quite up to date; now it is decidedly out of date. Davidson and MacKinnon's (2004) textbook is better, in my opinion.

will illuminate the method and will at the same time illustrate the features of ***Ects*** that facilitate its application.

We know that the t statistics calculated during estimation by nonlinear least squares (NLS) follow Student's t distribution under the null hypothesis that they test only asymptotically. Consequently, inferences based on these t statistics are only approximate. In order to see how the bootstrap can improve this, let us take a concrete example. Consider the nonlinear regression

$$\boldsymbol{y} = \alpha\boldsymbol{\iota} + \beta\boldsymbol{x}_1 + (1/\beta)\boldsymbol{x}_2 + \boldsymbol{u}. \tag{20}$$

where we write $\boldsymbol{\iota}$ for the constant vector of which every element is 1. The file `gv.dat` contains 10 observations on three variables, $\boldsymbol{y}$, $\boldsymbol{x}_1$, and $\boldsymbol{x}_2$. Based on this very small sample, we wish to test the hypothesis that model (20) is correctly specified, against the linear alternative

$$\boldsymbol{y} = \alpha\boldsymbol{\iota} + \beta\boldsymbol{x}_1 + \gamma\boldsymbol{x}_2 + \boldsymbol{u}, \tag{21}$$

with an unconstrained γ.

The model (20) can be estimated by NLS, giving estimates $\tilde{\alpha}$ and $\tilde{\beta}$ of the parameters. The Gauss-Newton regression (GNR) which corresponds to the the model (20) can be written as

$$\boldsymbol{y} - \alpha - \beta\boldsymbol{x}_1 - (1/\beta)\boldsymbol{x}_2 = b_\alpha\boldsymbol{\iota} + b_\beta\big(\boldsymbol{x}_1 - (1/\beta^2)\boldsymbol{x}_2\big) + \text{residuals}; \tag{22}$$

see chapter 6 of DM. The GNR corresponding to model (21) is written as

$$\boldsymbol{y} - \alpha - \beta\boldsymbol{x}_1 - \gamma\boldsymbol{x}_2 = b_\alpha\boldsymbol{\iota} + b_\beta\boldsymbol{x}_1 + b_\gamma\boldsymbol{x}_2 + \text{residuals}. \tag{23}$$

In order to obtain a test statistic, we evaluate all the variables of the GNRs (22) and (23) at the values obtained by estimation of the null hypothesis, which means, $\alpha = \tilde{\alpha}$, $\beta = \tilde{\beta}$, $\gamma = 1/\tilde{\beta}$. Both GNRs will thus have the same regressand. Then we express the GNR (23) in the form of an augmented regression with respect to (22). We check without trouble that the regressors of the regression

$$\boldsymbol{y} - \tilde{\alpha} - \tilde{\beta}\boldsymbol{x}_1 - (1/\tilde{\beta})\boldsymbol{x}_2 = b_\alpha\boldsymbol{\iota} + b_\beta\big(\boldsymbol{x}_1 - (1/\tilde{\beta}^2)\boldsymbol{x}_2\big) + b_\delta\boldsymbol{x}_2 + \text{residuals} \tag{24}$$

span the same linear space as do those of (23), and that (24) is simply the GNR (22), evaluated at $\tilde{\alpha}$ and $\tilde{\beta}$, with the additional regressor $\boldsymbol{x}_2$, to which we associate the artificial parameter denoted by b_δ, because it no longer corresponds to the parameter γ.

If we run both artificial regressions, we can compute an F statistic in the usual way using the two sums of squared residuals. But, as the test only has one degree of freedom, we can also use the t statistic associated with the artificial parameter b_δ given by (24), and thus need not run the GNR (22) corresponding to the null hypothesis. There exists yet another possibility, because of the fact that the GNR (24) is evaluated at the constrained estimates $\tilde{\alpha}$ and $\tilde{\beta}$. This third possibility is the nR^2 of (24), where $n = 10$ is the sample size, and the R^2 is uncentred.

The first commands of the file `gv.ect` are used to perform the operations described above. Here they are:

```
set n = 10
sample 1 n
read gv.dat y x1 x2
set beta = 1
nls y = alpha + beta*x1 + x2/beta
deriv alpha = 1
deriv beta = x1 - x2/(beta^2)
end
set sigma = sqrt(errvar)
gen Rb = x1 - x2/(beta^2)
ols res c Rb x2
set t = student(3)
set nR2 = n*R2
set Pt = 2*(1 - tstudent(abs(t),n-3))
set PnR2 = 1 - chisq(nR2,1)
show Pt PnR2
```

To avoid trouble, it is prudent to assign a nonzero value to the parameter β before running the `nls` command. Otherwise, we may have numerical problems due to zero denominators. The command that runs the GNR is the following:

```
ols res c Rb x2
```

and here are the results found in the output file:

```
ols res c Rb x2

Ordinary Least Squares:

Variable  Parameter estimate  Standard error  T statistic

constant    -31.165705          17.686371      -1.762131
Rb            5.511348           3.566145       1.545464
x2            6.505508           3.586173       1.814053

Number of observations = 10   Number of regressors = 3
R squared (uncentred) = 0.319780  (centred) = 0.319780
```

The last two commands are used to calculate the asymptotic P values associated with both tests, or in other words, the marginal significance levels. The values displayed on the screen are:

```
Pt = 0.112545
PnR2 = 0.073737
```

These values are quite different. The difference is due to the fact that an asymptotic approximation can behave badly if the sample size is only 10.

Before going any further, a remark about the R^2 of a linear regression: The coefficient of determination, the formal name for the R^2, can be defined in different ways. Older versions of ***Ects*** recognized only one of them, the so-called **uncentred** R^2, which is the ratio of the explained sum of squares (in ***Ects***, `sse`) to the total sum of squares (in ***Ects***, `sst`). We can access the uncentred R^2 through the `R2` variable. Although the R^2 does not have a formal statistical interpretation in general, it ca be useful, as here, for the calculation of some test statistics. If the constant is included in the regressors of an OLS regression, we might be interested in the so-called **centred** R^2. This is the R^2 given by a regression in which all the nonconstant variables, including the dependent variable, are replaced by deviations from their respective means, and the constant is eliminated. In version 3.3 of ***Ects***, if the constant `c` is included in the regressors of an `ols` command, the resulting table contains two R^2s. The first one is uncentred, and the second one is centred, as we see in the listing above. The uncentred R^2 is available under the name `R2c`.

* * * *

For the centred R^2 to be available, the constant must be called `c`. If we define the constant explicitly by another name, for instance by `gen iota = 1`, then the command `ols y iota x` does not give a centred R^2.

* * * *

EXERCISES:

Explain why the two R^2s are identical for the GNR that we just estimated.

Asymptotically and under the null hypothesis, the t statistic we just computed is drawn from the standard normal distribution, and the nR^2 is drawn from the χ^2 distribution with one degree of freedom. With a finite sample size, the test statistics are drawings from different distributions, generally unknown analytically. But we can study them by simulation. The procedure is as follows: We first establish a data generating process (DGP) called **bootstrap DGP**. This bootstrap DGP must absolutely satisfy the null hypothesis. Then, we use the bootstrap DGP to generate by simulation artificial samples, that have the same size as the true sample size. For each of these simulated samples, we calculate one or more test statistics, exactly like those we calculated using the original data. Finally, we construct the **empirical distribution function** of the simulated statistics. The empirical distribution function is the simulated version of the true unknown distribution function, of the statistics for the given finite sample size and under the null hypothesis. When the sample size tends to infinity, the difference between the empirical function and the true unknown function tends to zero.

A practical difficulty can arise due to the fact that the null hypothesis is in general not limited to only one DGP. In our example, under the null hypothesis, there are three unknown parameters, α, β, and σ^2, the variance of the error terms. In order to specify the bootstrap DGP, we must assign specific values to these parameters. The easiest and most efficient solution is to use estimates of the parameters under the null hypothesis. Another aspect that the null hypothesis does not specify completely is the distribution of the error terms. We can solve this last problem in two very different ways, which however often yield very similar results. The first approach is to assume that the error terms are generated by the normal distribution. The second is to do a **resampling** of the residuals obtained when estimating the null hypothesis. The first approach constitutes what is called the **parametric bootstrap** , the second is the **nonparametric** or **semiparametric bootstrap**.

The parametric bootstrap

We first look at the the parametric bootstrap, which is easier to set up. The bootstrap DGP can be written in the following way:

$$\boldsymbol{y}^* = \tilde{\alpha}\boldsymbol{\iota} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2 + \tilde{\sigma}\boldsymbol{v}^*, \quad \boldsymbol{v}^* \sim N(\mathbf{0}, \mathbf{I}).$$

Here, $\tilde{\alpha}$, $\tilde{\beta}$, and $\tilde{\sigma}$ are the restricted estimates. The stars denote simulated values. The regressors $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$, assumed to be exogenous, are the same for each simulation. It is easy to see that this DGP satisfies the null hypothesis. Next, in `gv.ect`, we find:

```
set B = 999
sample 1 B
gen tt = 0
gen nRR2 = 0

def fnreg = a + b*x1 + x2/b
def dbeta = x1 - x2/(b{̂}2)

sample 1 n
gen rfn = alpha + beta*x1 + x2/beta
set i = 0
silent
noecho
while i < B
   set i = i+1
   gen ys = rfn + sigma*random()
   set a = alpha
   set b = beta
   nls ys = fnreg
   deriv a = 1
   deriv b = dbeta
   end
   ols res CG x2
   set tt(i) = student(3)
   set nRR2(i) = n*R2
```

```
    end
    echo
    restore
```

Here, we write `B` to denote the number of simulations. For theoretical purposes, we prefer to choose `B` such that `B` + 1 is a nice round number, rather than `B` itself, which explains the choice of `B = 999`. The vectors `tt` and `nRR2` are created to contain the `B` simulated statistics. As the expression $\tilde{\alpha} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2$ is the same for all simulations, we calculate it once and for all in the `rfn` vector. Furthermore, we define the macros `fnreg` and `dbeta` so as to accelerate the calculations done inside the loop. We thus do once and for all the work of interpreting the expressions which represent the regression function and its derivative with respect to β.

The simulations themselves are located inside the loop. The `ys` vector represents $\boldsymbol{y}^*$. In the context of these simulations, the "true" values of the parameters, in other words, the DGP values, are known: They are given by the constrained estimates. As the nonlinear estimation algorithm converges faster if the starting point is close to the estimates, it is prudent to initialize the parameters of the simulated regressions (`a` and `b` instead of `alpha` and `beta`) with the true values. For the GNR, we use the fact that the NLS command creates the variables `c` and `Rb` automatically, as the two columns of the `CG` variable. The command `ols res CG Rb` is then used to run the GNR, after which we save the two simulated statistics in the vectors kept for that purpose.

At this point, we are ready to construct the empirical distribution functions of the two statistics. This is done easily if we make use of the `cdf` function, available for the first time in version 3.3 of ***Ects***. The name of this function comes from the expression *Cumulative Distribution Function*. The rest of the program demonstrates how to use it.

```
  sample 1 101
  set linestyle = 1
  gen x = -4 + 8*time(-1)/100
  mat edft = cdf(tt,x)
  gen lstud = tstudent(x,n-3)
  plot (x edft lstud)
  gen x = 10*time(-1)/100
  mat edfR2 = cdf(nRR2,x)
  gen lchi2 = chisq(x,1)
  plot (x edfR2 lchi2)
```

We want to construct the graphical representation of the empirical distribution functions for the 999 realizations of both statistics. These realizations are contained in the vectors `tt` and `nRR2`. In order to graph these two distribution functions, we first need to choose the points at which the function will be evaluated. The choice must be very different for the two statistics, because the t statistic can take on negative values, whereas the nR^2 cannot. For the t statistic, we choose a grid of 101 points, regularly spaced in the interval $[-4, 4]$. For the nR^2, the grid extends from 0 to 10. The best way to create these grids is to use the **time** function, because, formally, grids are just time trends.

The **cdf** function is used in a **mat** command and takes on two arguments.

* * * *

This fact means that the rule laid down in Man2 by which all functions used in a **mat** command take only one argument, at the cost of a syntax error, has been made more flexible in the current version of the software. In the next chapter, we will find other new features, which also break the old rule.

* * * *

The command

```
mat fre = cdf(stat,abscissa)
```

where the first argument, **stat**, is a $B \times m$ matrix, and the second, **abscissa**, is an $n \times 1$ matrix, creates the **fre** matrix, an $n \times m$ matrix, with as many rows as **abscissa** and as many columns as **stat**. We denote x_i, $i = 1, \ldots, n$, the i^{th} element of **abscissa**. Then, the (i, j) element of **fre**, for $i = 1, \ldots, n$, $j = 1, \ldots, m$, is the proportion of the elements of the j^{th} column of **stat** that are less than or equal to x_i.

Let us denote by b_k, $k = 1, \ldots, B$, the k^{th} element of the first column of **stat**. The formal definition of the distribution function $\hat{F}$ of the B elements of this column is as follows:

$$\hat{F}(x) = \frac{1}{B} \sum_{k=1}^{B} I(b_k \le x), \tag{25}$$

where the value of the **indicator function** $I(b_k \le x)$ is equal to 1 if the inequality which serves as argument is true, and 0 otherwise. The sum in the definition (25) is thus the number of elements b_k that are less than or equal to x. If we divide by B, we obtain the proportion. The consequence of the definition (25) is that the (i, j) element of **fre** is the value of the distribution function of the elements of column j of **stat** at x_i, element i of **abscissa**.

It is illuminating to compare the empirical distributions with the distributions given by asymptotic theory. These distributions are Student's t, with $n-3 = 7$ degrees of freedom, and χ^2, with one degree of freedom. In the program, we evaluate these theoretical distribution functions at the same points as the empirical distributions, and we display the results using the **plot** command. These can be seen in Figure 5, where the dotted lines are the plots of the empirical functions and the continuous lines are the theoretical functions. It seems that the $t(7)$ distribution is a good approximation, but the $\chi^2(1)$ less so. This could explain the difference in the asymptotic P values given by both statistics.

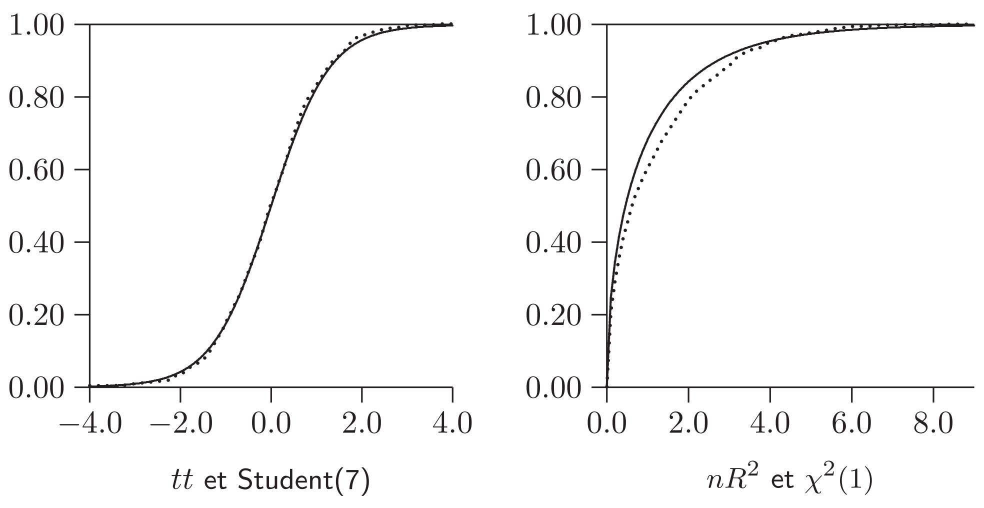

Figure 5 Empirical and theoretical distribution functions

Most often, we are not interested in all the details of the empirical distribution function of a statistic. To make inferences, we need only the bootstrap P value. Any bootstrap P value is a measure of the probability area under the tail(s) of a distribution, beyond the realized value of the test statistic. For the χ^2 test, this area is 1 minus the value of the distribution function of the χ^2 at the realized statistic. For the t statistic, it is twice 1 minus the value of the $t(7)$ distribution function at the absolute value of the realized test statistic, for a two-tailed test that is. The calculation of bootstrap P values for the example in `gv.ect` is done as follows:

```
sample 1 B
gen Pt = abs(tt) > abs(t)
gen PnR2 = nRR2 > nR2
gen iota = 1
mat Pt = iota'*Pt/B
mat PnR2 = iota'*PnR2/B
show Pt PnR2
```

The `Pt` and `PnR2` vectors contain a value of 1 only if the bootstrap test statistic is further away in the distribution tail than the test statistic calculated with the real data, and 0 otherwise. Then we add up the elements of these vectors and divide the result by `B` in order to get bootstrap P values.

* * * *

Recall: the use of the equality signs (`=`) or the inequalities (`<` and `>`) in a **`gen`** command creates boolean values, in other words, 1 if the equality or inequality is satisfied, 0 otherwise.

* * * *

Practically, if we are only interested in the bootstrap P value, it is of no interest to store the values of all the bootstrap test statistics, as we did in the vectors `tt` and `nRR2`. It is enough to define scalar variables, initialized to 0, and to increment them each time the bootstrap statistic is further away in the distribution tail than the true statistic. However we choose to do the computation, the P values calculated for our example are

```
Pt = 0.090090
PnR2 = 0.090090
```

It is just luck that we find two identical values, but this fact indicates clearly that an inference based on either of the bootstrap P values is better than an asymptotic inference.

The nonparametric bootstrap

Let us now consider the nonparametric bootstrap. This version of the bootstrap does not make any precise assumption about the distribution of the error terms. Instead, we resample the residuals given by the estimation of the null hypothesis. The term "resampling" can be explained in terms of the empirical distribution function of the residuals. Whereas the parametric bootstrap draws simulated residuals from the normal distribution, resampling draws them from the empirical distribution function of the residuals. We say "resampling" because each bootstrap error term is equal to a residual from the original estimation. What changes is that the order of the residuals is no longer respected, and that the same residuals can be drawn exactly once, more than once, or not all. In order for each bootstrap error term to be a drawing from the same empirical distribution, the drawing is done *with replacement*. This means that, if the drawing was done by pulling a piece of paper out of a hat, we would have to put that piece of paper back into the hat after each drawing.

We write $\hat{F}(\hat{\boldsymbol{u}})$ for the empirical distribution of the residuals $\hat{\boldsymbol{u}}$ given by estimating regression (20). Then the nonparametric bootstrap DGP can be written as

$$\boldsymbol{y}^* = \tilde{\alpha}\boldsymbol{\iota} + \tilde{\beta}\boldsymbol{x}_1 + (1/\tilde{\beta})\boldsymbol{x}_2 + \boldsymbol{u}^*, \quad \boldsymbol{u}^* \sim \hat{F}(\hat{\boldsymbol{u}}).$$

Modern users prefer an improvement to this classical bootstrap DGP, where the residuals are multiplied by $n/(n-k) = 10/7$ before being subject to resampling. The reason for this is that the variance of the empirical distribution of the modified residuals is equal to $\hat{\sigma}^2 = \text{SSR}/(n-k)$, the unbiased estimator of the variance of the error terms.

The nonparametric bootstrap by resampling is carried out using the following commands, found in `gv.ect`:

```
gen us = res*sqrt(n/(n-3))
while i < B
   set i = i+1
   gen ys = rfn + us(random(0.9,n+0.9))
   set a = alpha
   set b = beta
```

```
    nls ys = fnreg
    deriv a = 1
    deriv b = dbeta
    end
    ols res CG x2
    set tt(i) = student(3)
    set nRR2(i) = n*R2
 end
```

The `us` vector contains the modified residuals, and the resampling is done through the following command

```
gen ys = rfn + us(random(0.9,n+0.9))
```

and the rest of the procedure is identical to the one for the parametric bootstrap. Let us now see how and why the expression

```
us(random(0.9,n+0.9))
```

constitutes a drawing with replacement in "the hat" of the modified residuals.

The result of the command

gen B = A($\langle expn \rangle$)

where `A` is a variable that is already present in the memory of the computer, is a matrix of which the number of rows corresponds to the current sample size and the number of columns is equal to the number of columns of `A`. Each row of `B` is one of the rows of `A` chosen in the following way. The $\langle expn \rangle$ argument is evaluated, according to the rules of the `gen` command, with, as a result, a `smplend` $\times\, m$ matrix, $m \geq 0$, of which only the first column will be taken into account. Corresponding to each element of this column, we calculate a real number according to the rules stated in Man2 for the evaluation of indices: Let x_i be element i of the column; the subscript is then the biggest integer n_i such that $n_i \leq (x_i + 0.1)$. Then, we assign to the i^{th} row of `B` the n_i^{th} row of `A`.

If we use two indices, as for instance in the command

gen C = A($\langle expn_1 \rangle, \langle expn_2 \rangle$)

the `C` variable will be a column vector, of which the i^{th} element is the element A(n_i, m_i), where n_i and m_i are respectively indices calculated from elements i of the first columns of the matrices obtained on evaluating $\langle expn_1 \rangle$ and $\langle expn_2 \rangle$.

* * * *

Each time we ask for as element that doesn't exist, or if we use a zero or negative index, the elements of the corresponding row of the result will be zero. The syntax described in the above two paragraphs is not included in previous versions of ***Ects***.

* * * *

When the expression

```
random(0.9,n+0.9)
```

is evaluated, the result is a vector with each element a drawing from the uniform distribution on an the interval that extends from 0.9 to $\texttt{n}+0.9$. We can see that all the realizations from the uniform distribution on the [0.9, 1.9[interval give rise to an index equal to 1, that all realizations in [1.9 2.9[give rise to an index equal to 2, and so on. The last segment, $[\texttt{n}-1+0.9,\ \texttt{n}+0.9[$, gives rise to an index equal to `n`. The uniform distribution allocates to each segment, of length 1, the same probability, equal to $1/\texttt{n}$. It follows that the probability that each index $1, 2, \ldots, \texttt{n}$ is selected is equal to $1/\texttt{n}$. Combining this fact with the way the `gen` command interprets indices means that each element of

```
us(random(0.9,n+0.9))
```

is an independent drawing from the empirical distribution of the elements of the `us` vector, as we wished.

The bootstrap P values given by the nonparametric bootstrap are once again identical for both test statistics that we are considering, and very similar to those given by the parametric bootstrap:

```
Pt = 0.100100
PnR2 = 0.100100
```

On the basis of 999 bootstrap simulations, the values of 0.090090 and 0.100100 are not significantly different.

As all simulation requires us to generate random numbers, we discuss this here in more detail. How can a *deterministic* machine like a computer generate *random* numbers? The simplest answer is that the computer uses what is called in mathematics **deterministic chaos**, which generates numbers that appear to be random. For simulation purposes, appearances are more than enough. But the generating process is deterministic, and so can be reproduced. If we want to generate twice the same set of "random" numbers, we have only to select the same **seed** as starting point for the random number generator. The starting point for the ***Ects*** generator consists of two numbers, contained in the `seed` variable. The two elements of this variable are updated after each use of the `random` function; we can look at it at any moment by doing

```
show seed
```

At the moment ***Ects*** is launched, the `seed` variable contains default values of `20000` et `987654321`. To change the starting point of the random number generator, we use the `setseed` command. The syntax is straightforward:

`setseed` $\langle \mathit{expn}_1 \rangle$ $\langle \mathit{expn}_2 \rangle$

Ects allocates to the first element the value of $\langle \mathit{expn}_1 \rangle$, calculated under the rules of the `set` command, and to the second element, the value of $\langle \mathit{expn}_2 \rangle$. Meanwhile, the `seed` variable is updated. It is important to note that it is not enough to change the values of `seed`, because the change is not echoed inside the generator itself. Only `setseed` enables us to change the values inside the random number generator.

Exercises:

The goal of this exercise is to demonstrate both aspects of deterministic chaos. First the deterministic aspect. Generate a series of random numbers using the `random` command after having written down the values in the `seed` variable. Then, reset the seed to the same values using `setseed` and generate a second series of random numbers. Check that both series are identical. Then the chaotic aspect. Redo the same exercise, but change the value of `seed(1)` by adding 1 before generating the second seed. Calculate the correlation between both series: it will be close to zero.

References

Abramowitz, M. and I. A. Stegun (1964). *Handbook of Mathematical Functions*, National Bureau of Standards, Washington.

Davidson, R. and J. G. MacKinnon (1993), (DM). *Estimation and Inference in Econometrics*, Oxford University Press, New York.

Davidson, R. and J. G. MacKinnon (1999). “Artificial Regressions”, in *A Companion to Theoretical Econometrics*, ed. B. H. Baltagi, Blackwell, Oxford, pp 16-37.

Davidson, R. and J. G. MacKinnon (2004). *Econometric Theory and Methods*, Oxford University Press, New York.

Efron, B. and R. J. Tibshirani (1993). *An Introduction to the Bootstrap*, New York, Chapman and Hall.

Golub and Reinsch (1970). *Numerische Mathematik*, 14, pp 403-470.

Gouriéroux, C. and A. Monfort (1989). *Statistique et Modèles Économétriques*, Economica, Paris.

Hamilton, J. D. (1994). *Time Series Analysis*, Princeton University Press.

Moshier, S. L. (1989). *Methods and Programs for Mathematical Functions*, Prentice-Hall.

Plauger, P. J. (1995). *The Draft Standard* C++ *Library*, Prentice-Hall, New Jersey.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling (1986). *Numerical Recipes*, Cambridge University Press, Cambridge.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling (1992). *Numerical Recipes in C*, Cambridge University Press, Cambridge.[7]

[7] Other versions of this book can be found, with the programs written in Fortran or Pascal.

Stroustrup, B. (1991). *The* C++ *Programming Language*, Second Edition, Addison-Wesley, New-York.

General Index

<, 79
=, 79
>, 79
#, 6
%, 6
\, 42

Access path, 4
Algol (programming language), 2
answer, 47–49
AR(1) (autoregressive process), 55–57
ARCH(1) (ARCH process of order 1), 68–72
archdemo.ect, 68–72
arg, 47, 52
argen.ect, 55–57, 61
Artificial regression, 34, 38–39
  BRMR, 38–39
Artificial regression, 73
asin, 20
Automatic derivatives, 3, 15–19
Automatic differentiation, 3, 15–19, 23–24, 37–38, 41, 51

Bono, Pierre-Henri, 1
Boolean values, 79
Bootstrap, 72–82
  nonparametric or semiparametric, 76, 80–82
  parametric, 76–80
BRMR, 38–39

c, 17, 75
C (programming language), 2
C++ (programming language), 2, 55
cdf, 77–78
Cephes Mathematical Function Library, 2
CG, 25, 28, 35, 39, 45, 77
Chaos, 13
chisq, 16, 20
coef, 29, 35, 41, 45, 63
Collett, Brian, 2
cols, 50
conv, 56, 57, 62
Correlation
  contemporaneous, 66
cos, 20
crit, 29, 36, 41, 45
Cumulative distribution function, 31

Data Generating Process (DGP)
  bootstrap, 75–82
def, 15, 53, 54
del, 54
deriv, 24, 27, 37, 39, 40, 43, 53
Deterministic Chaos, 82–83
diff, 15–19, 24
differentiate, 19, 54
Directory
  root, 4
  tmp, 4, 10, 11
Distribution function
  empirical, 75–82

end, 43, 46, 47, 57
equation, 53
errvar, 29, 44
Estimation
  nonlinear, 23–46
Evaluation
  of indices, 81
expand, 54

fit, 5, 30
Floating point processor, 4
Free Software Foundation, 2
Function
  incomplete gamma, 16
  logistic, 31

Gauss-Newton regression (GNR), 25, 73–77
gen, 16–18, 32, 40, 49, 50, 53, 57, 59, 69–70, 79, 81–82
Generalised Method of Moments (GMM), 40–46
GMM, 40–46
gmm, 3, 23, 26, 32, 40–41, 46
gmm.ect, 40–46
gmmhess, 40–46
gmmweight, 40, 42–46
GNR, 25, 73–77
GNU, 2
gnuplot, 3–15
gnuplot.n, 10

`gnuplot.gnu`, 10
`GNUTERM`, 4
Golub and Reinsch, 2
`grad`, 18–19, 41, 45
Gradient
  of a scalar, 18–19
Graphics, 3–15
`gv.dat`, 73–74
`gv.ect`, 74–82

`HCCME`, 45
`hess`, 18–19
Hessian
  of a loglikelihood function, 36
  of a scalar, 18–19

Incomplete gamma function, 16
Indicator function, 78
Indices
  evaluation, 81
Initialisation
  of the parameters of a nonlinear model, 24
`instr`, 27, 43
Instrumental variables (IV), 26–30, 42
`int`, 19–22
`integral.ect`, 20–22
Internal representation
  of expressions, 19
Internal representation interne
  of expressions, 54
Internal tables, 53–54
`INTTOL`, 20–22
`invertlagpoly`, 60–66
`invhess`, 35, 37, 41
`invOPG`, 36
`iv`, 26, 27, 30
IVGNR (GNR with IV), 28
`ivnls`, 30
`ivnls.dat`, 26, 27
`ivnls.ect`, 26

Jacobian matrix, 25, 28

Kelley, Colin, 3

`lag`, 57, 69
lag operator (L), 58
`lagpoly`, 58–66
LaTeX, 14
Least squares
  Nonlinear (NLS), 23–26, 73
`lhat`, 35
`lhs`, 39
`linestyle`, 6, 77
Linux, 1–2, 4
Logit, 31–39
`logit.ect`, 38
Loops, 55–57
`lowtriang`, 67
`lt`, 35

MacKinnon, James G., 2
Macro, 15
Maple, 19
`mat`, 17, 18, 33, 40, 49, 50, 53, 59, 78
Mathematica, 19
Maxima, 19
Maximum likelihood (ML), 31–39
`maxintiter`, 20–22
`maxiter`, 24–25
Mean
  of dependent variable, 25–26
ML, 31–39
`ml`, 3, 23, 32–35, 37, 38, 46–48
`mlar`, 38–39, 52–53
`mlhess`, 36–40, 51
`mlopg`, 35–36, 38, 49
Model
  binary choice, 31
  bivariate, 61
  logit, 31–39
  multivariate, 61
  univariate, 61
Moshier, Stephen L., 2

`newcrit`, 29
`newlogit.ect`, 31–39, 46
Newton's method, 36
`ninst`, 30, 45
`niter`, 30, 35, 41, 45
`nliv`, 26–30, 43, 45, 53
`nliv.ect`, 26–30, 40, 45
NLS, 23–26, 73
`nls`, 3, 23–26, 53, 74
`nls.ect`, 25
`nobs`, 30, 35, 45
`noquestions`, 25
`nreg`, 30, 35, 41, 45
Number of iterations, 24–25
Numerical integration, 19–22

`oir`, 30
`ols`, 5, 18, 26, 63, 74–75
`ols.dat`, 5, 10, 17, 32, 63, 67
Operating system
  DOS, 1
  Linux, 1, 2, 4
  multitasking, 10
  Unix, 4
  Windows, 1, 4
Overidentification statistic, 30

P value, 74
  bootstrap, 79–82
PI, 6
Plauger, P.J., 2
plot, 5–11, 77–78
polylag, 58–67
Polynomial
  in lag operator, 58–61
  matrix, 63
PostScript, 14
  encapsulated, 14
Press *et al* (1986), 1
Press *et al* (1992), 2
Printer
  HP LaserJet II, 14
  PostScript, 14, 15
Probability distribution
  chi-squared (χ^2), 16, 75
  logistic, 31
  multivariate normal, 66–67
  standard normal (N(0,1)), 75
proclogit.ect, 46–54
procedure, 4, 46–53
Procedures, 3–4, 46–53
Process
  ARCH, 68–72
  ARCH(p), 68
  ARMA, 57–61
  ARMA(p, q), 57–58
  ARMAX, 61–62
  AR(p), 58
  autoregressive
    of order 1, 55–57
  GARCH, 68–72
  GARCH(p, q), 69
  MA(q), 58–61
  VAR, 62–67
product, 71
Programming language
  Algol, 2
  C, 2
  C++, 2
PwX, 30

quit, 49

R^2, 26, 75
  centred, 75
  uncentred, 75
R2, 26, 75
R2c, 75
random, 6, 64, 67, 81–83
Random number generator, 82–83
recursion, 57, 68–70
Regressand
  of an artificial regression, 39
Regressor
  of an artificial regression, 39
rem, 6
res, 5, 18, 25, 28, 30, 45, 63–64
Resampling, 72, 76, 80–82
rows, 50

sample, 40, 49, 69
savegnu, 10
second, 37, 53
seed
  of random number generator, 82
seed, 82–83
Segmentation fault, 4
set, 6, 16, 19, 24, 48–51, 53, 59, 69, 82
setseed, 82–83
showall, 53–54
showint, 20, 22
showlagpoly, 63–66
showprogress, 24, 28, 36, 38
Simulation, 55–57, 72–83
  recursive, 55–57
sin, 20
Singular value decomposition, 2
smplend, 40, 42, 60, 69
smplstart, 40, 42, 69
sort, 13
sse, 30, 75
ssr, 18, 30, 45
sst, 30, 75
stderr, 29, 35, 45
Stroustrup, Bjarne, 2
student, 29, 35, 45
sum, 71
SVGA, 4

testplot.ect, 5–9
time, 11, 77–78
Time series, 57–72
tmp, 4, 10, 11

Unix, 4
uptriang, 67

VAR (vector autoregression), 62–67
var.ect, 63–67
Variable
  binary, 31
  limited dependent, 32
vcov, 29, 44, 45
Vector autoregression (VAR), 62–67

Weight matrix for GMM, 42
weightmatrix, 43
White noise, 68

Williams, Thomas, 3
Windows, 4, 5

`x11`, 4
$\boldsymbol{X}(\boldsymbol{\beta})$, 25, 28, 44–45
`xlabel`, 11, 13
`xrange`, 11
`XtPwXinv`, 29
`XtWAWtXinv`, 44–46
`XtXinv`, 39

`ybar`, 26, 30

Ects Index

Command Files
- archdemo.ect, 68–72
- argen.ect, 55–57, 61
- gmm.ect, 40–46
- gv.ect, 74–82
- integral.ect, 20–22
- ivnls.ect, 26
- logit.ect, 38
- newlogit.ect, 31–39, 46
- nliv.ect, 26–30, 40, 45
- nls.ect, 25
- proclogit.ect, 46–54
- testplot.ect, 5–9
- var.ect, 63–67

Data Files
- gv.dat, 73–74
- ivnls, 30
- ivnls.dat, 26, 27
- ols.dat, 5, 10, 17, 32, 63, 67

Ects Commands
- #, 6
- %, 6
- \, 42
- answer, 47–49
- def, 15, 53, 54
- del, 54
- deriv, 24, 27, 37, 39, 40, 43, 53
- differentiate, 19, 54
- end, 43, 46, 47, 57
- equation, 53
- expand, 54
- gen, 16–18, 32, 40, 49, 50, 53, 57, 59, 69–70, 79, 81–82
- gmm, 3, 23, 26, 32, 40–41, 46
- gmmhess, 40–46
- gmmweight, 40, 42–46
- invertlagpoly, 60–66
- iv, 26, 27, 30
- lagpoly, 58–66
- lhs, 39
- mat, 17, 18, 33, 40, 49, 50, 53, 59, 78
- ml, 3, 23, 32–35, 37, 38, 46–48
- mlar, 38–39, 52–53
- mlhess, 36–40, 51
- mlopg, 35–36, 38, 49
- nliv, 26–30, 43, 45, 53
- nls, 3, 23–26, 53, 74
- ols, 5, 18, 26, 63, 74–75
- plot, 5–11, 77–78
- procedure, 4, 46–53
- quit, 49
- recursion, 57, 68–70
- rem, 6
- sample, 40, 49, 69
- second, 37, 53
- set, 6, 16, 19, 24, 48–51, 53, 59, 69, 82
- setseed, 82–83
- showall, 53–54
- showlagpoly, 63–66
- sort, 13
- time, 11

Ects Functions
- <, 79
- =, 79
- >, 79
- asin, 20
- cdf, 77–78
- chisq, 16, 20
- cols, 50
- conv, 56, 57, 62
- cos, 20
- diff, 15–19, 24
- grad, 18–19
- hess, 18–19
- int, 19–22
- lag, 57, 69
- lowtriang, 67
- polylag, 58–67
- product, 71
- random, 6, 64, 67, 81–83
- rows, 50
- sin, 20
- sum, 71
- time, 77–78
- uptriang, 67

Ects Variables
- arg, 47, 52
- c, 17, 75
- CG, 25, 28, 35, 39, 45, 77
- coef, 29, 35, 41, 45, 63

crit, 29, 36, 41, 45
errvar, 29, 44
fit, 5, 30
grad, 41, 45
HCCME, 45
INTTOL, 20–22
invhess, 35, 37, 41
invOPG, 36
lhat, 35
linestyle, 6, 77
lt, 35
maxintiter, 20–22
maxiter, 24–25
ninst, 30, 45
niter, 30, 35, 41, 45
nobs, 30, 35, 45
noquestions, 25
nreg, 30, 35, 41, 45
oir, 30
PI, 6
PwX, 30
R2, 26, 75
R2c, 75
res, 5, 18, 25, 28, 30, 45, 63–64
savegnu, 10
seed, 82–83
showint, 20, 22
showprogress, 24, 28, 36, 38
smplend, 40, 42, 60, 69
smplstart, 40, 42, 69
sse, 30, 75
ssr, 18, 30, 45
sst, 30, 75
stderr, 29, 35, 45
student, 29, 35, 45
vcov, 29, 44, 45
XtPwXinv, 29
XtWAWtXinv, 44–46
XtXinv, 39
ybar, 26, 30