Ects

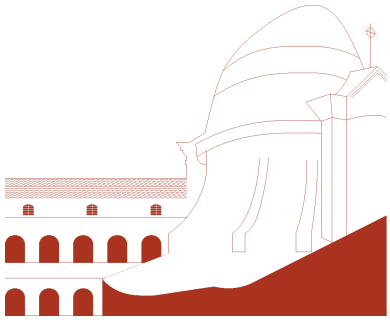
Software for Econometrics

Versions 2 and 3

Russell Davidson

March 2004
Revised, September 2005

Ects, Versions 2 and 3
© Russell Davidson, March 2004.
All rights reserved for reproduction, adaptation, translation, and execution for all countries



Foreword

This econometrics software package was originally designed for the use of the students of the Magistère Ingénieur-Économiste at the Université d'Aix-Marseille II (now called the Université de la Méditerranée) and for the graduate students at the research group GREQE (now GREQAM), in Marseille, France. Its purpose is however not exclusively pedagogical, since, right from its beginnings, the software proved itself useful, not only for class exercises, but also for a number of empirical applications. It remains true that the original conception was formulated in order to help students learn econometrics. For this reason, the software does not try to do everything conceivable. It is quite intentional that it is often necesary to do a little programming after estimating a model in order to compute, for instance, a particular test statistic.

For many years, I made use of successive versions of a textbook I wrote with my colleague J. G. MacKinnon, entitled *Estimation and Inference in Econometrics*. I refer to it throughout this manual simply as DM. In 2004, another textbook by the two of us appeared, entitled *Econometric Theory and Methods*. I do not refer to it here, since this documentation antedates the new book. Both books are published by Oxford University Press, New York.

To the many people, students and others,
who helped make this software useful.

Table of Contents

Table of Contents v

Introduction 1

1 How to use *Ects* 3

1 Installation 3
2 The `ols` Command 5
3 Data Reading and Writing 9

2 Linear Regression 13

1 Variable Transformation 13
2 Collinearity 17
3 Instrumental Variables 19

3 Matrix Operations 22

1 Simple Operations 22
2 Matrix Elements and Blocks 27
3 Other Operations 31

4 Nonlinear Estimation 35

1 Nonlinear Regression 35
2 Maximum Likelihood 39
3 Generalised Method of Moments 43

5 Interactive mode; File management 47

1 The Command and Output files 47
2 Work Contexts 48
3 Control of the Contents of the Output File 52

6 Monte Carlo Experiments 57

1 The Programming of Loops 57
2 Conditional Execution of a Command Block 60
3 (Pseudo-)Random Numbers 62
4 Monte Carlo 63

7 Everything else **68**

1 Other Functions, Other Rules 68
2 Arithmetic Operations: Useful Rules 71
3 Exceptional Functions 74
4 Unix and the Source Code 76

References **77**

General Index **78**

***Ects* Index** **80**

Introduction

This is the English translation of the first volume of documentation for ***Ects***. It first appeared in French with a date of March 1993, and has been revised on several occasions since, although the cover page of the French version still bears the original date, so as to identify the version of the software referred to, namely version 2. I have taken the opportunity provided by the translation to make a number of corrections, to remove some material concerned with the operating systems of more than ten years ago, chiefly DOS, something that will no doubt always be part of the history of computing, but is hardly used at all anymore. In some cases, I have added some footnotes in order to correct statements that no longer apply to any extant version of the software. For the most part, though, I have resisted the temptation to update and to tinker, and have left the manual as faithful a translation of the original French as was compatible with telling the truth (more or less), and not actively misleading readers.

Version 3 of ***Ects*** came out in 1999, and, with it, a second volume of documentation covering the considerable number of new features added progressively since 1993. I hope that a translation of that volume will soon be forthcoming. This first volume remains necessary for version 3, since the second volume covers only the new features. A good deal of effort went into keeping version 3 compatible with the previous version, so that this volume applies equally well to versions 2 and 3, and should apply with only minor changes to later versions.

The very first version of ***Ects*** was released around 1989 or 1990; I forget the exact date. It was much less ambitious than subsequent versions, having been undertaken on account to the lack of suitable econometric software at the Faculté des Sciences Économiques in Marseille. It was intended solely as a teaching aid, and, although its documentation was just adequate, it was pretty sparse. Almost as soon as it was completed, I moved on to the programming of a more complete package, using C++ instead of C as the programming language. The added features, most importantly the matrix operations introduced in version 2, made it possible for me to use ***Ects*** myself, not only as a substitute for other econometrics packages like TSP or Shazam, but also as a matrix language suitable for performing serious simulations.

I am engaged at present (March 2004) in the coding of the projected version 4 of ***Ects***. It too is programmed in C++, but not at all the same sort of C++ as what I used in previous versions. This is in some measure due to the fact that the specification of the language itself evolved fairly fast during the 1990s, so that things that were allowed back then no longer are, and many new things then undreamt of are now possible. But a more important reason

for the change of programming style is that, over the period of the existence of C++, the programming community has become more sophisticated in its use of object-oriented programming, and the use of templates. Several valuable books have appeared in the last five or six years, in which radically new programming techniques are explained. These new techniques make it possible to write better code, that is, code that is clearer, terser, and, above all, less error-prone. Computer programming is intrinsically error-prone, and anything that reduces the time wasted hunting down obscure bugs is always welcome.

As computer programming becomes more of a science and less of an arcane art, it is a pity that so few economists and econometricians seem to be attracted by what has always been a necessary aspect of empirical work and work based on simulations. The "point-and-click" generation may be more familiar with computers than its predecessors, but it's a great deal less skilful in its use of them. ***Ects*** has different goals from C++, but it still needs programming skills if it is to be used fruitfully. I myself find it very convenient to develop simulation experiments with ***Ects***, and to translate the result into C++ if industrial-strength simulations are required – which is by no means always. If ***Ects*** can help students recover programming skills possessed by their predecessors, but since all too often lost, I will be more than delighted.

Many people have helped and supported me in the development of ***Ects***, with suggestions for new or improved features, and, in a few cases, suggestions for the actual programming. These people have all come up through the graduate program at GREQAM in Marseille, and many are now distinguished economists and econometricians, academic or otherwise. Now that this translation is available, making it possible for ***Ects*** to be used in the English-speaking community, and more especially at McGill University, it is fitting that I should record the names of the students and ex-students who helped the most. I express here my gratitude to Pierre-Henri Bono, Dirk Eddelbüttel, Emmanuel Flachaire, Christophe Flachot, Laurence Lasselle, Stéphane Luchini, and Christian Raguet, in alphabetical order, and also to all my students and colleagues at GREQAM, for creating an environment in which serious research, as well as the development of econometrics software, is such a pleasure.

It is my hope that McGill econometricians, students and others, will also contribute to further developments of ***Ects***. It is my pleasant duty to thank the bilingual team of four students from my Economics 467 class of 2003-4 who did the translation of this volume, Ram Lokan, Serge Mouracade, Amjad Patel, and Martin Viger. I was pleasantly suprised by how little work was left for me to make this volume ready for distribution.

Chapter 1
How to Use *Ects*

1. Installation

Ects was created as a small, but practical, piece of software rather than a complex but powerful tool. It began more than ten years ago as an executable file of no more than 100KB, but it has grown in size since those days, more, I think, because of the way software and compilers have developed generally than because lots of extra functionality has been added. The newer aspects of the software are treated in the second volume of documentation; this volume deals with the older, more basic, procedures.

In 2003, the long-surviving DOS version of ***Ects*** was finally abandoned. A Linux version has been available for at least eight years, and is, in a sense, the canonical version, since all development of the software has taken place on the Linux platform for many years. However, a version that runs under Windows as a command-line application has recently been compiled, and it is that version which, along with the Linux version, is available on my websites in France and Canada:

`http://russell.vcharite.univ-mrs.fr/ects3`

`http://russell-davidson.arts.mcgill.ca/ects3`

The instructions given there may vary with time and subsequent developments, and so here I do no more than sketch the simple installation procedure. The zip file (at present called `Ects3_cygwin.zip`) is small enough to fit on a floppy disk. It should be unzipped in a suitable directory, after which the directory will contain the executable file `ects3.exe` and three dynamic libraries, which should be kept in the same directory as the executable. ***Ects*** can then be run in the same way as any other command-line application. When it is started, a command-line window opens in which commands can be typed. Alternatively, and often preferably, such a window can be opened beforehand. After changing to the ***Ects*** directory, one can type `ects3`, by itself or followed by the name of a command file. If you work with Linux, then things are simpler. It should in most cases be enough to download the executable file `ects3`, install it somewhere suitable like `/usr/local/bin`, and then run it from wherever you please.

Throughout this volume and the second volume of documentation, reference is made to command files that contain ***Ects*** code used to illustrate and explain many things, most to do with ***Ects***, but others concerning econometrics generally. These files can be downloaded as a zip file, called (at least at this time of writing) `demofiles.zip`, or a gzipped tar, `demofiles.tar.gz`. The files contained in these archives are as follows, in alphabetical order:

```
2sls.dat
2sls.ect
archdemo.ect
ar.dat
ar.ect
argen.ect
arnls.ect
gen.ect
gmm.ect
gv.dat
gv.ect
integral.ect
ivnls.dat
ivnls.ect
logit.ect
ml.ect
mrs.ect
nearunit.ect
newlogit.ect
nliv.ect
nls.dat
nls.ect
nlsols.ect
ols.dat
ols.ect
pitch.ect
proclogit.ect
season.ect
sur.dat
sur.ect
testplot.ect
var.ect
```

You may verify that all `*.dat` files contain data, and all `*.ect` are programs.

The `*.ect` files are provided as examples, so that you may use ***Ects*** right away. However, notice that files containing the programs cannot be created by ***Ects*** itself. Usually, a text editor, such as Notepad (for Windows users) is used for such purpose. Word processors (such as Microsoft Word) can also

be used *as long* as they can create ASCII files. Indeed, an ***Ects*** executable file is simply an ASCII file.

If you are working with Macintosh, you can edit your files directly and then transfer them by the utility AFE (Apple File Exchange).

To execute `ols.ect`, type `ects3 ols`. If the extension of the command file is not `.ect`, it is necessary to specify it. If we type `ects3` without a file name, we enter the interactive mode, which we will discuss later. (This mode did not exist in the first version.)

2. The ols Command

The linear regression model is the most frequently used model in econometrics. It is frequently written as

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{u}.$$

* * * *

See Chapter 1 of DM for a theoretical explanation.

* * * *

The **dependent variable**, sometimes called the **regressand**, is $\boldsymbol{y}$. It is represented algebraically by a vector of n scalar components, and geometrically by this same vector considered as a point in space $\mathbb{R}^n$. Similarly, the k columns of the $n \times k$ matrix $\boldsymbol{X}$ represent the **explanatory variables**, or the **regressors**.

The **parameters** $\boldsymbol{\beta}$ of the model are estimated by the **Least Squares** method, here **ordinary least squares**, or OLS.

Now look at the command file, `ols.ect`. The file reads as follows:

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
quit
```

Even though it looks very simple, this file allows us to see four ***Ects*** commands.

The first one, `sample`, defines the sample size. From the computer's viewpoint, two integers are defined by the `sample` command. The first one, noted by t_1, is the first index; it is the first observation to take into account in the operations that will follow. The second one, t_2, is the index of the last one. In our case, we start with the first observation for each variable, and finish with the hundredth. Usually, the command is written `sample` $\langle expn_1 \rangle$ $\langle expn_2 \rangle$. The two expressions, $\langle expn_i \rangle$, $i = 1, 2$, may be, as here, explicit integers, or can also be expressed algebraically (which we will consider further on) to which the computer can give a numerical value. We must have that $\langle expn_1 \rangle \geq 1$, and

that $\langle expn_2 \rangle \geq \langle expn_1 \rangle$, or else the command's results may be unpredictable. But watch out! The initial value determined by the computer for t_1 and for t_2, is 1 in both cases. This implies that our sample contains only one observation. In general, there will be $t_2 - t_1 + 1$ observations.

At this stage, the sample size is defined. There are 100 observations. Up to now, no variables have been created. This is done by executing the `read` command. The command's syntax is simple: the word `read` is followed by a file name accessible to ***Ects***, and then by the names of one or more variables. If the file doesn't exist, or if it cannot be found, an error message will appear. The variable names must be strings of alphanumeric characters (that is, one of the $26+26+10 = 62$ characters A–Z, a–z, 0–9), with no blank space, starting with an alphabetical character. Names such as `x1` are then admissible, whereas `1x` would not be.

Before looking at the `ols` command in more detail, we see that the `quit` command is used to stop the execution of the command file. If other commands follow the `quit` command, they will not be executed. However, using `quit` is not mandatory. We could have done without it in our case; ***Ects*** stops when it realizes it has come to the end of the file.

Now, consider `ols`. Yet again, the syntax is very simple. After `ols` we first find the name of the dependent variable, the regressand, here `y`. Then, in addition to the names of the three variables found in the `ols.dat` file, there is the constant, noted `c`. This variable is defined automatically by ***Ects*** each time a command such as `ols` is executed. For each observation of the defined sample, the value of `c` equals 1. But watch out! It is possible to redefine `c`, but not recommended. The consequences of such redefinition would be hardly predictable and conceivably disastrous.

The algebraic version of the command

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

is then

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_3 + \boldsymbol{u}, \tag{1}$$

where we denote the constant by α, the parameter associated to the "variable" `c`.

Now it is time to try the `ols.ect` program. Type in the instruction

```
ects ols
```

followed by enter. Errors aside, you should see on your screen all the lines of the `ols.ect` file, followed by `Ects terminated`, meaning that the program has been executed. You will find the results in a file named `ols.out`. This file can be viewed on the screen, printed, or sent to a word processor. Like a command file, it is a simple ASCII file. Here is what it contains:

```
sample 1 100
Sample set from observation 1 to observation 100
read ols.dat y x1 x2 x3
Variables y x1 x2 x3 read from file ols.dat
ols y c x1 x2 x3

Ordinary Least Squares:

Variable    Parameter estimate  Standard error  T statistic

constant       87.053652          24.944936       3.489833
x1              1.579244           0.224144       7.045661
x2             -3.116123           0.194432     -16.026823
x3              1.169272           0.161331       7.247679

Number of observations = 100   Number of regressors = 4
Mean of dependent variable = 133.296804
Sum of squared residuals = 210392.502407
Explained sum of squares = 2.644800e+06
Estimate of residual variance
  (with d.f. correction) = 2191.588567
Standard error of regression = 46.814406
R squared (uncentred) = 0.926312  (centred) = 0.804901

Estimated covariance matrix:

622.249817  -5.188451   0.364278   3.064586
 -5.188451   0.050241  -0.007466  -0.019113
  0.364278  -0.007466   0.037804  -0.000765
  3.064586  -0.019113  -0.000765   0.026028
```

We see the answers to the commands `sample` and `read`: these are simple checks that the commands have been executed. Concerning `ols`, we can read in the first table, corresponding to each regressor, the **estimated parameter** (`Parameter estimate`), the **standard error** (`Standard error`), and the ***t* statistic** (`T statistic`). Then come a few scalar values associated to the regression: `Number of observations`, the sample size ($t_2 - t_1 + 1$), the `Number of regressors`, the number of regressors in the regression, the sum of squared residuals, `Explained sum of squares`, the explained sum of squares, `Estimate of residual variance`, the $\hat{\sigma}^2$ of the regression, calculated taking into account the degrees of freedom "used up" by the parameter estimation, and finally `R squared`, the regression R^2. This last one is `uncentred`. We then find a second table that contains the estimated covariance matrix, which we usually denote as $\hat{\sigma}^2(\boldsymbol{X}^\top \boldsymbol{X})^{-1}$.

The results in the `ols.out` file are also available inside ***Ects***. As we will see later on, it is possible to use these data to construct test statistics. After each `ols` command, a certain number of variables are created or updated. ***Ects*** defines four variables, corresponding to the four lines that define the first table above. First, `nobs` is a scalar variable that contains the number of

observations used by `ols`. (Quickly, a **scalar variable** is like a variable whose definition is preceded by the command `sample 1 1`, a 1×1 matrix. For more details, see section 2.1.) In a similar way, the scalar variable `nreg` contains the number of regressors. Then come `ssr` and `sse`, variables which respectively contain the sum of squared residuals and the explained sum of squares. A fifth variable, defined as `sst`, contains the total sum of squares. Furthermore, the scalar variables `R2` and `errvar` equal the R^2 and the $\hat{\sigma}^2$ of the regression.[1]

Those scalar variables aside, ***Ects*** updates three series after each `ols`. (A **series**, contrary to a scalar variable, has many components. If it has n components, the series is represented by a $n \times 1$ column vector.) `fit` is a series that contains the fitted values, whereas `res` contains the residuals. Those two series each have t_2 elements. The series `hat` contains the $t_2 - t_1 + 1$ diagonal elements of the $\boldsymbol{P_X} \equiv \boldsymbol{X}(\boldsymbol{X}^\top \boldsymbol{X})^{-1}\boldsymbol{X}^\top$ matrix – see first chapter of DM; more specifically section 1.6.

Three generally shorter series are also created. They are `coef`, which has `nreg` elements, and contains the estimated parameters, `stderr`, of the same size and containing the `nreg` standard errors, and `student`, also of the same size, containing the t statistics. Finally, two *matrices*[2] are created; `XtXinv` contains the matrix $(\boldsymbol{X}^\top \boldsymbol{X})^{-1}$, and `vcov` contains the same matrix multiplied by $\hat{\sigma}^2$, the variance matrix of the estimated parameters.

EXERCISES:

Try different `sample` commands in the `ols.ect` file. For example, what are the results of the following commands: `sample 50 100`, `sample 1 50`, `sample 1 200`? Do you understand why the results are what they are?

Input the data in two increments. It is important to know why

```
sample 1 50
read ols.dat y x1 x2 x3
sample 51 100
read ols.dat y x1 x2 x3
sample 1 100
```

and

```
sample 51 100
read ols.dat y x1 x2 x3
sample 1 50
read ols.dat y x1 x2 x3
sample 1 100
```

yield different results. Before trying it on the computer, guess the outcome of

[1] As we will see later on, these "scalar" variables will be matrices if the `ols` command is used to do many regressions at the same time.

[2] We will see how we can do matrix operations with ***Ects*** in chapter 3.

```
sample 1 100
read ols.dat y x1 x2 x3
sample 1 50
read ols.dat y x1 x2 x3
```

Even though the data are input normally, it is possible to split the regression itself in two. Try

```
sample 1 45
ols y c x1 x2 x3
sample 46 100
ols y c x1 x2 x3
```

or even

```
sample 1 33
ols y c x1 x2 x3
sample 34 67
ols y c x1 x2 x3
sample 68 100
ols y c x1 x2 x3
```

3. Data Reading and Writing

We have seen in the last section how ***Ects*** can read data with the `read` command. This command should be used with caution, as it is not very flexible. Indeed, we must be careful that the data should be correctly organized within the file. Each file line must have one observation on each variable whose name is on the list given to the command `read`. For example, the very first line of the file `ols.dat` reads as follows:

```
231.3227     91.52508     -7.879150     -22.82027
```

and the following 99 have the same formatting. Note however that blank lines, or lines that begin with the character `#` or `%`, are skipped without being counted. Further, the file can contain anything at all after the last line of data, since ***Ects*** does not read past the end of the current sample size.

The rule is the following: if the last `sample` command was `sample` t_1 t_2, the content of the first line of the file will be given to observation t_1 of each of the variables on the list. The same will apply to the other observations: the content of line i will become observation $t_1 + i - 1$. The reading of the file stops either at the end of the file, or after reading observation t_2, if we come to it before the end of the file.

What will happen if we try:

```
read ols.dat y x1 x2 x3 x4
```

Ects will simply say that the *five* variables `y x1 x2 x3 x4` were input. This is true – ***Ects*** rarely lies – but variable `x4` will have 100 observations (if the command `read` is preceded by `sample 1 100`), each one equal to zero. It is probably not very bothersome.

If we try

```
read ols.dat y x1 x2
```

what will turn up? This time, ***Ects*** will answer correctly that the variables `y x1 x2` were input. This is quite simple: if we ask for only three variables, ***Ects*** will go to the next line after having found three numbers.

If ***Ects*** comes to the end of a data file before finding all the observations in the sample defined by `sample`, the missing observations will be replaced by zeros. The commands

```
sample 1 200
read ols.dat y x1 x2 x3
```

create 4 variables of 200 components each, the last 100 ones being zeros.

The data must absolutely be organized by *observation* and not by *variable*. Some econometrics software allows the user to choose the format of the data file, but this is not the case with ***Ects***. Usually, a file whose format is not suited for ***Ects*** can be easily modified with a text editor. Otherwise, now is the time to put to work your knowledge of C, or even C++, in order to create a new data file that can be read by ***Ects***.

When the time comes to do a `read`, it is possible that, within the list of variables submitted to `read`, there are variables that already exist. In that case, two things may happen. If an existing variable has already at least t_2 observations, the observations from 1 to $t_1 - 1$, and from $t_2 + 1$ on, will be kept as is, whereas observations t_1 to t_2 will be replaced by numbers in the file. In the case where an existing variable has fewer than t_2 observations, it will be entirely erased and replaced.

EXERCISES:

Do some tests on the computer to be sure that you understand all what we have seen so far. To see the effect of the commands that you will enter in your command file, you can use the `write` command, which will be discussed in the next paragraph. To see the first 10 observations of the variables `y x1`, for example, we can do

```
sample 1 10
write temp.dat y x1
```

A `temp.dat` file will be created (if it exists already, it will be written over) to contain the observations 1 to 10 of `y` and `x1`.

The `read` command is the only one that reads data. However, there are four commands to write or to display data. The first one, mentioned in the above

exercise, is `write`. The syntax is identical to `read`, and its result the exact opposite. For example, the command

```
write ols2.dat y x1 x2
```

creates an `ols2.dat` file and writes into it the $t_2 - t_1 + 1$ observations of `y`, `x1`, and `x2` of the sample. As we mentioned earlier, if a file named `ols2.dat` already exists, it will be overwritten by the `write` command. If $t_1 = 1$ and $t_2 = 100$, the information within the file `ols2.dat` would be the same as that in our initial file, `ols.dat`, but without the last column. By use of the `sample` command, we can also delete lines from `ols.dat`.

The other writing commands are `print`, `show`, and `put`. The four writing commands are implemented by a single function within ***Ects***, which allows for two arguments. It is the choice of those two arguments that distinguishes the commands. The first argument takes only two possible values, TRUE and FALSE. If the value is TRUE, the variable's name will be printed before its components whereas if the values is FALSE, only the components will be printed. The value is TRUE for `print` and for `show`, whereas it is FALSE for `write` and for `put`. If we do

```
sample 1 4
write crap y x1
print y x1
```

(`crap` is a file name) the result of the command `write` is

```
231.322700  91.525080
168.226300  58.321080
  5.600930  71.192640
121.004900  85.375360
```

while the result for `print` is

```
     y                 x1
231.322700         91.525080
168.226300         58.321080
  5.600930         71.192640
121.004900         85.375360
```

The results are slightly different if the variables to print are scalar variables.

The second argument is more interesting, as it specifies the location where the result will be saved. If we use `write`, it is necessary to specify explicitly a file name that will receive the result of the command. As we have seen with the last example, it is not necessary to specify the destination of a `print` command. Indeed, the result will appear in the output file, the same one that receives the results of an `ols` command, for example. By default, the output file has the same name as the command file with the `.out` extension

(see section 1.2).[3] Should we wish that the results should simply be displayed, the command is

```
show y x1
```

whose effect is identical to

```
print y x1
```

except that what was in the output file is now displayed on the screen. The command `put` also writes to the output file. We will see later the usefulness of this command.

EXERCISES:

Try to learn the varied uses of the commands `show` and `write`. Input variables from `ols.dat`, or from another file, by `read`, and make them appear on the screen by the use of `show`. Then create a new file by `write` to hold the variables. But watch out! Should you choose an existing file name, it will be erased. View the `ols.dat` file and your new file to ensure that the variables are the same.

Notice the consequences if, within a `print`, `write`, or `show` command, you mix up scalar variable or series.

[3] The output file can be chosen differently. See section 5.1.

Chapter 2
Linear Regression

1. Variable Transformation

For many reasons, we often want to transform variables that we have input, before or after running a regression. Consequently, ***Ects*** provides the capability of doing many algebraic manipulations, with the commands `gen` (for *generate*) and `set`. The distinction is easy to understand: we use `gen` if we have a **series**, that is, a variable having a value for each observation between t_1 and t_2, and we use `set` if we have a simple **scalar variable**.

For example, let us study the command file `gen.ect`. Here is its content:

```
sample 1 100
read ols.dat y x1 x2 x3
ols y c x1 x2 x3
set ssr0 = ssr
print ssr0
gen yy1 = y - 2*x1
ols yy1 c x2 x3
set ssr1 = ssr
set F1 = 96*(ssr1 - ssr0)/ssr0
print ssr1 F1
gen xx1 = x1 - x3
gen xx2 = x2 - x3
ols y c xx1 xx2
set ssr2 = ssr
set F2 = 96*(ssr2 - ssr0)/ssr0
print ssr2 F2
gen yy2 = y + 2*x2
ols yy2 c x1 x3
set ssr3 = ssr
set F3 = 96*(ssr3 - ssr0)/ssr0
print ssr3 F3
gen yy3 = y - 2*x1 + 2*x2 - 2*x3
ols yy3 c
set ssr4 = ssr
set F4 = 96*(ssr4 - ssr0)/(3*ssr0)
print ssr4 F4
quit
```

We start by reading the data from the file `ols.dat`, and we then redo the OLS estimation of regression (1):

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \beta_3 \boldsymbol{x}_3 + \boldsymbol{u}$$

The idea of the rest of the file is to implement F tests of the following hypotheses:

$$\begin{array}{rl} \beta_1 = 2; & \text{statistic } \texttt{F1} \\ \beta_1 + \beta_2 + \beta_3 = 0; & \text{statistic } \texttt{F2} \\ \beta_2 = -2; & \text{statistic } \texttt{F3}, \text{ and} \\ \beta_1 = 2, \beta_2 = -2, \beta_3 = 2; & \text{statistic } \texttt{F4} \end{array} \tag{2}$$

* * * *

Recall that the statistic used for the F test of a set of constraints has the form

$$F = \frac{(\mathrm{SSR}_c - \mathrm{SSR}_{nc})/r}{\mathrm{SSR}_{nc}/(n-k)},$$

where SSR_c is the sum of squared residuals from the restricted regression, SSR_{nc} is the same for the unrestricted regression, n is the sample size, k is the number of regressors in the *unrestricted* regression, and r is the number of restrictions.

* * * *

The command

```
set ssr0 = ssr
```

creates a new scalar variable, named `ssr0`, to which is given the value of the variable `ssr`. As we have seen in section 1.2, `ssr` is a scalar variable that is updated after each `ols` command. Its value is, of course, the sum of the squared residuals of the regression. The following command

```
print ssr0
```

will make the line

```
ssr0 = 210392.502407
```

appear in the output file `gen.out`.

The next group of commands is used to test the first hypothesis of (2). A new regressor is created by

```
gen yy1 = y - 2*x1
```

It is regressed on the constant, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$. Then the F statistic is computed and written in the output file. The other hypotheses are tested in the same way. Notice that the multiplication symbol is `*`, and that it cannot be deleted, contrary to usual algebraical notation.

The manipulations and transformations made with `gen.ect` are very simple, but it is possible to ask for more complicated algebraic operations. Before

considering the numerous functions known to ***Ects***, notice that the four arithmetic operations are easily obtained by the use of the `+`, `-`, `*`, and `/` symbols. The operations `*` and `/` have **precedence** over `+` and `-`. By this, we mean that, for example, in the expression

```
y + x*z
```

the product `x*z` will be calculated first, and the result then added to `y`. The result is simply $y + xz$. ***Ects*** respects the usual algebraical conventions.

The command `gen` computes the arithmetical operations observation by observation. For example, the command

```
gen y = x + z
```

has the following effect:

$$y_t = x_t + z_t \text{ for } t = t_1, \ldots, t_2.$$

The operation can also be written using vector notation:

$$\boldsymbol{y} = \boldsymbol{x} + \boldsymbol{z}.$$

Similarly, the effect of

```
gen y = x*z
```

is that

$$y_t = x_t z_t \text{ for } t = t_1, \ldots, t_2. \tag{3}$$

Notice that this does not correspond to a standard vector operation. However, we sometime speak of the **Schur product**, denoted $\boldsymbol{y} * \boldsymbol{z}$, and defined by (3). The operation given by the command

```
gen y = x/z
```

is just

$$y_t = x_t / z_t \text{ for } t = t_1, \ldots, t_2,$$

and, although unknown to standard algebra, this operation is very useful in econometrics.

There exists another operation that has a precedence even higher then the four base ones. It is the "power" operation. If we write

```
gen z = y^w
```

we obtain the result

$$z_t = y_t^{w_t}.$$

It is not necessary for the exponent w_t to be a positive integer, but we may have problems if y_t is negative. In such cases, z_t would be equal to zero.

Let us now look at a few functions known to ***Ects***. We will first look at functions of a single argument, which may either be a scalar variable or a

series, according as we use `set` or `gen`. The result of the application of each of these functions is a new scalar variable or a new series, as the case may be. The functions are the following: `log`, `sqrt`, `sign`, `abs`, `exp`, `sin`, `cos`, `tan`, `sum`, `time`, `Phi`, and `phi`. The meaning of most of these operations is fairly clear. Here are the algebraic definitions:

$$\begin{aligned}
\texttt{log}(\texttt{z}) &= \log z;\\
\texttt{sqrt}(\texttt{z}) &= \sqrt{z};\\
\texttt{sign}(\texttt{z}) &= 1 \text{ if } z \geq 0, \text{ and } -1 \text{ otherwise};\\
\texttt{abs}(\texttt{z}) &= z \text{ if } z \geq 0, \text{ and } -z \text{ otherwise};\\
\texttt{exp}(\texttt{z}) &= e^z;\\
\texttt{sin}(\texttt{z}) &= \sin z;\\
\texttt{cos}(\texttt{z}) &= \cos z;\\
\texttt{tan}(\texttt{z}) &= \tan z.
\end{aligned}$$

But watch out! If you try to use the `log` and `sqrt` functions with a negative or zero argument, the result will be zero, and an error message will follow.

The function `Phi` is the CDF of the standard normal distribution, $\mathrm{N}(0,1)$. The formal definition is

$$\Phi(x) = \frac{1}{(2\pi)^{1/2}} \int_{-\infty}^{x} e^{-y^2/2}\, dy. \tag{4}$$

The integral cannot be expressed in closed form using elementary functions. The integrand, which is the standard normal density, is available through the ***Ects*** function `phi`. We have

$$\phi(x) = \frac{1}{(2\pi)^{1/2}} e^{-y^2/2}.$$

The function `sum` is used to compute a series of cumulated values. If `z` is defined by

```
gen z = sum(y)
```

the observation z_t is equal to $\sum_{i=1}^{t} y_i$.
`time` is usually used to define time trends with a constant argument. If

```
gen z = time(0)
```

then $z_t = t$. In general, no matter y, the result of the

```
gen z = time(y)
```

command is that $z_t = y_t + t$.

An operation of the uttermost importance in the manipulation of time series is the **lag** operation. Indeed, the `lag` function exists. The syntax is as follows:

`gen z = lag(`$\langle expn_1 \rangle$`,`$\langle expn_2 \rangle$`)`

The result of the command is that, for all $t = t_1, \ldots, t_2$,

$$z_t = \begin{cases} y_{t-n} & \text{if } t - n > 0, \\ 0 & \text{if } t \leq n. \end{cases} \tag{5}$$

The meaning of equation (5) is that $\boldsymbol{y}$ is the *series* that results from the evaluation of the expression $\langle expn_2 \rangle$. n, on the other hand, is the positive or negative integer that results from the following operation: Let $\boldsymbol{n}$ be the series that results from the evaluation of the expression $\langle expn_1 \rangle$. n is then the biggest integer such that $n < (n_1 + 0.1)$, where n_1 is the first component of $\boldsymbol{n}$.

* * * *

For example, if we write `lag(n,y)`, and if `n` represents a series whose first component equals 4.5678, n is equal to 4, which is the biggest integer such that $n < 4.5678 + 0.1 = 4.6678$.

* * * *

Usually, we choose $\langle expn_1 \rangle$ as an explicit integer, like 2 or -3, or a scalar with an integer value.

EXERCISES:

What is the result of the `gen` command on series that exist already? Try for example

```
sample 1 100
read ols.dat y x1
sample 1 50
gen z = y
sample 45 70
gen z = x1
```

and find how `gen` and `sample` interact.

The file `ar.ect` presents an extended example of manipulations by `gen` and `set`. It is a nonlinear estimation performed using the artificial Gauss-Newton regression. Note that we can use a single observation of a series as if it were a scalar variable. Thus `beta(2)` is observation 2 of the series `beta`.

2. Collinearity

If in the linear regression model

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{u},$$

there exists a linear combination of the columns of the matrix $\boldsymbol{X}$ equal or almost equal to $\mathbf{0}$, the model is no longer identified. Computers are not able

to do arithmetic operations to an infinite precision, and consequently, even though the matrix $\boldsymbol{X}$ has full rank, an approximate collinearity can be as bothersome as an exact one.

For this reason, before running an `ols` command, ***Ects*** deletes all regressors that can be expressed up to a "small" error as a linear combination of already defined regressors. The details of this "small" error are quite technical, but there are scalar variables pre-defined by ***Ects***, called `TOL` (for tolerance), whose default value equals 10^{-8}, used to give a precise meaning to the term "small". Even though the value of `TOL` can be modified by the `set` command, it is strongly recommended not to go too far from the pre-defined value, so as to avoid incorrect results, and also not to give rise to floating-point exceptions with the numeric co-processor.

A way to create an exact collinearity is to use four seasonal variables simultaneously with a constant. If we then try to run the following regression:

```
ols y c x s1 s2 s3 s4
```

where we use the four variables `s1`, `s2`, `s3`, and `s4`, we expect that ***Ects*** deletes `s4`. Why `s4`? Because `s4` is the first variable of the list of regressors able to be expressed as a linear combination of already defined regressors.

Now look at the file `season.ect`. After reading a sub-sample of the data contained in `ols.dat` you will see the following commands:

```
gen s1 = seasonal(4,1)
gen s2 = seasonal(4,2)
gen s3 = seasonal(4,3)
gen s4 = seasonal(4,4)
```

It is like this that we can create seasonal variables. The `seasonal` "function" takes two arguments, each of which must be a positive integer. In reality, the arguments are evaluated exactly like the first argument of the `lag` function, in the sense that we use the biggest integer that is smaller than the first component's argument, plus 0.1. The expression `seasonal(`$\langle n \rangle$`,`$\langle m \rangle$`)` means a variable of which each observation equals zero, except for those whose indices take the form $m+kn$, for $k = 0, 1, \ldots$, for which the value is 1. m is the index of the first observation for which the value is 1, and n is the periodicity of the seasonality. If the first argument n is smaller than 1, it is replaced by 1. It follows that `s1` is different from zero precisely for observations $1, 5, 9, \ldots$. For `s2` it is observations $2, 6, 10 \ldots$, *etc.* You can insert the command

```
print s1 s2 s3 s4
```

in the command file after the generation of the seasonal variables to ensure that they were correctly generated.

Then come the commands

```
ols y c x s1 s2 s3
ols y c x s1 s2 s3 s4
ols y x s1 s2 s3 s4
```

If you now run the file `season.ect`, you will see the result. The first and last regressions have nothing unusual, but the second one, where there is indeed a redundant variable, produces results identical to the first, with the exception of the last line. Here, ***Ects*** makes the following remark:

```
The variable s4 was excluded as collinear with other variables.
```

Observe that the t statistics, and all the elements of the covariance matrix, are identical in the results of the first two regressions. This implies that ***Ects*** has correctly computed the estimation of $\hat{\sigma}^2$, taking into account only five regressors.

EXERCISES:

Try to have an idea of how approximate a collinearity can be and still cause the deletion of a regressor by ***Ects***. Generate two almost collinear variables, and then use them as regressors in a regression as follows:

```
gen x2 = x1 + TINY
ols y c x1 x2
```

where `TINY` is a size that you can modify by the command `set`. Exceptionally, play with the the value of the predefined variable `TOL`. The default value is 10^{-8}, that you can express as `1.0E-8` in "computer language." Try bigger values of `TOL` and find, with respect to `TOL`, the critical value of `TINY` that allows ***Ects*** to distinguish the two regressors.

3. Instrumental Variables

There are many reasons that lead us to use instrumental variables (see Chapter 7 of DM). Recall now the elements of the procedure. The instrumental variables estimator (IV) of the linear regression

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{u}$$

for a matrix of instruments $\boldsymbol{W}$ can be written as

$$\hat{\boldsymbol{\beta}}_{\mathrm{IV}} = (\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{y}. \tag{6}$$

It is necessary that the number l of instruments should be at least equal to the number k of regressors. Suppose that $k = 3$, and $l = 5$. Then the translation of (6) that must be done so that ***Ects*** can read it is:

```
iv y x1 x2 x3 (w1 w2 w3 w4 w5)
```

If l is less than k, an error message will appear, and ***Ects*** will go to the next command.

It is often the case that the set of instruments $\boldsymbol{W}$ contains regressors. This is quite in order; `iv` is not bothered by this. Indeed, all exogenous regressors should also be instruments.

* * * *

In particular, the constant and other artificial variables, such as seasonal and dummy variables...

* * * *

The definition of $\hat{\sigma}^2$ used by `iv` is the right one, namely,

$$\hat{\sigma}^2 = \frac{1}{n-k}\sum_{t=0}^{n}(y_t - X_t\hat{\boldsymbol{\beta}}_{\mathrm{IV}})^2.$$

This expression is *not* what is produced by the two-stage least squares (2SLS) procedure, which gives a wrong answer.

The command file `2sls.ect` is used to show how to use `iv`. With the data contained in `2sls.dat`, we estimate, by IV, the system of simultaneous equations

$$\begin{aligned} q_t &= \alpha_d - \beta_{11}p_t + \beta_{12}Y_t + u_{1t}, \text{ and} \\ q_t &= \alpha_o + \beta_{21}p_t - \beta_{22}C_t + u_{2t}. \end{aligned} \tag{7}$$

The variables q and p are respectively the quantities traded and the prices; the exogenous variables Y and C are consumers' incomes and production costs.

EXERCISES:

Redo the estimation of the (7) system by 2SLS. Try to compute the right covariance matrix of the estimated parameters without using `iv`.

A collinearity problem can appear in the context of an estimation by instrumental variables, as in the OLS context. Indeed, the deletion of regressors or redundant instruments by ***Ects*** is done in two steps. First, if there is collinearity in the instruments, the useless instruments will be excluded from the list. Secondly, after the projection of the regressors on the space of the instruments, (creation of the matrix $\boldsymbol{P}_W\boldsymbol{X}$), the possibly redundant columns of this matrix will be deleted.

The `iv` command, like the `ols` command, creates or updates a series of variables. The scalars `ssr`, `sse`, `sst` have the same meaning as in a regression run by `ols`. In general, the relation `sse` + `ssr` = `sst`, always satisfied with `ols`, is not true in the case of `iv`, because the projection used is oblique, different from the orthogonal projection used by `ols`: see chapter 7 of DM.

EXERCISES:

Repeat the estimation of the model of section 1.2 by `iv`. Show that, if the instruments and the regressors are the same variables, all results would be identical to those given by `ols`.

The scalar `errvar` contains $\hat{\sigma}^2$, and the series `coef`, `stderr`, `student` contain the estimated parameters, their standard errors, and the t statistics, exactly as in the case of `ols`.

In addition to the scalars `nobs`, (n = the sample size), `nreg`, (k =the number of regressors), we also have `ninst`, the number of instruments, l. The series `fit` and `res` contain the fitted values and the residuals from the instrumental variables estimation.

Instead of the matrix `XtXinv` that has no interest in the IV context, a matrix `XtPwXinv` is defined. As the notation shows, it is the $(\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1}$ matrix. Again, the matrix `vcov` contains the estimate of the covariance matrix of the estimated parameters, $\hat{\sigma}^2(\boldsymbol{X}^\top \boldsymbol{P}_W \boldsymbol{X})^{-1}$.

Chapter 3
Matrix Operations

1. Simple Operations

In everyday econometrics, it is often necessary to perform **matrix operations**. For such needs, ***Ects*** has the `mat` command, which works like `gen` and `set`, but which operates on matrices. Within ***Ects***, all the variables are matrices. A scalar is simply a 1×1 matrix, and a series is a column matrix, that is, a $n \times 1$ matrix, for a given positive integer n. Depending on the choice of `mat`, `gen`, or `set`, the arithmetic operations contained in the rest of the command are treated differently.

Regarding `gen`, everything depends on the current values of the internal variables `smplstart` and `smplend`. These are the variables that we denoted as t_1 and t_2 in section 1.2. We can directly access these variables (for example, by `print` or `show`) through "external" variables which have the same name. But watch out! To change the values of the internal variables, it is necessary to use the `sample` command. A `set` will indeed change the external variables, but not the internal values used to determine the sample size.

EXERCISES:

To understand the consequences of these manipulations fully, run the following commands:

```
sample 1 10
gen y = time(0)
sample 3 3
gen i = y
set ii = y
set iii = y(3)
print i ii iii
```

Explain the obtained results.

For the duration of a `mat` command, neither the internal values nor the external ones are taken into consideration. Every variable is considered as a matrix, which implies that each variable is assigned two dimensions, n, the number of rows, and m, the number of columns. We should always make sure that

these two dimensions are compatible with the requested operations. Thus, if we want to add up two matrices, the two dimensions of the two matrices must be the same; to multiply them, it is necessary for the number of columns of the left matrix to equal the number of rows of the right matrix.

* * * *

The general rule is not much more complex: for details, refer to section 7.2. For additions and subtractions, the dimensions of the first matrix are used. This means that if the second matrix has fewer rows than the first one, the missing rows are replaced by zeros; if it has more rows, these rows are not retained in the result. For matrix multiplications, the number of rows in the product matrix is equal to the number of rows of the leftmost factor, and the number of columns is equal to the number of columns of the rightmost factor. If a row or column does not exist, it is replaced by a row or column of zeros.

* * * *

How to generate matrices other than scalars or series? A simple way would be to use an outer product of two vectors, that is to say, of two series. For example, if two series `x` and `y` have been generated for a sample of size n, the operation

```
mat Z = x*y'
```

will give an $n \times n$ matrix `Z`. We will indeed have

$$\boldsymbol{Z} = \boldsymbol{x}\boldsymbol{y}^{\top},$$

Observe that `*` indicates matrix multiplication here. In the above example, we have made use of another very useful operation, namely, **transposition** of a matrix.

We often write $\boldsymbol{Z}'$ for the transpose of $\boldsymbol{Z}$.[4] Therefore, within a `mat` command, the prime `'` can be used to carry out the transposition of a matrix. But it is obvious that most matrices are not outer products. Two commands allow the construction of a matrix based on its columns or its rows. If we reuse the example of the linear regression considered in the `ols.ect` file, we will be able to construct a 100×4 matrix by the following command:

```
mat X = colcat(c, x1, x2, x3)
```

The function `colcat` is used to concatenate columns.

* * * *

The above command will work only *after* the `ols` command. This is due to the fact that the constant `c` is created only after a first regression is run.

* * * *

[4] This notation is not used in this manual, we prefer the notation $\boldsymbol{Z}^{\top}$.

The arguments of the `colcat` function can be matrices. For example, we could have proceeded in two or three steps. In fact, the matrices `X1`, `X2`, and `X3` defined by the following commands are identical:

```
mat X1 = colcat(c, x1, x2, x3)
mat Y = colcat(c, x1, x2)
mat X2 = colcat(Y, x3)
mat Y1 = colcat(c, x1)
mat Y2 = colcat(x2, x3)
mat X3 = colcat(Y1, Y2)
```

The `rowcat` function operates in a similar way. Here are three equivalent methods for generating the transpose of the matrix `X` created above.

```
mat ctr = c'
mat x1tr = x1'
mat x2tr = x2'
mat x3tr = x3'
mat X1tr = rowcat(ctr, x1tr, x2tr, x3tr)
mat X2tr = X1'
mat X3tr = rowcat(c', x1', x2', x3')
```

* * * *

The functions `colcat` and `rowcat` generate matrices as big as possible. If all the columns of a `colcat` don't have the same dimensions, the dimension of the result will be the biggest of the dimensions of the arguments. A similar property is true for `rowcat`.

* * * *

We should note that `colcat` and `rowcat` can also be used within a `gen` command. The results will be identical except for the following minor detail: the number of rows of a `colcat` can never exceed the `smplend` under `gen`. Thus, if we replace `mat` by `gen`, and if we define a smaller sample, we will obtain a smaller matrix. Suppose that the current sample size is given by

```
sample 1 100
```

Consider the following:

```
sample 1 10
mat X1 = colcat(c, x1, x2, x3)
gen X2 = colcat(c, x1, x2, x3)
```

The matrix `X1` will have 100 rows, but `X2` will have only 10 rows.

There does not exist a matrix division operation. However, a nonsingular square matrix can be **inverted**. To obtain the inverse of a nonsingular square matrix, we use the following command:

```
mat XX = X inv
```

EXERCISES:

The function `inv` finally enables us to obtain the OLS estimator without the use of the `ols` command! Indeed, we can check that the result of the following operations:

```
mat X = colcat(c, x1, x2, x3)
mat betahat = (X'*X) inv * (X'*y)
```

is identical to

```
ols y X
```

The previous exercise demonstrated that the matrices are recognised by the `ols` command. As in the case of `colcat`, we can mix series and matrices in the arguments of `ols`. This is also true for the `iv` command.

EXERCISES:

Run the following commands:

```
mat X = colcat(c, x1, x2)
ols y X x3
```

And notice how ***Ects*** denotes the three regressors of the matrix `X`. Confusion can arise if another variable `X1` exists! Try the same kind of exercise for `iv`. You will see that the instruments as well as the regressors can be grouped within a matrix.

What would happen if we asked ***Ects*** to invert a matrix which is not square, or which is square but singular? We can answer both questions simultaneously. We obtain what we call a **generalised inverse** of the matrix. Let $\boldsymbol{A}$ denote an $n \times m$ matrix. A generalised inverse of $\boldsymbol{A}$ is an $m \times n$ matrix $\boldsymbol{A}^+$ which satisfies the following properties (see, for example, Gouriéroux-Monfort (1989)):

$$\begin{aligned} \boldsymbol{A}\boldsymbol{A}^+\boldsymbol{A} &= \boldsymbol{A}; \\ \boldsymbol{A}^+\boldsymbol{A}\boldsymbol{A}^+ &= \boldsymbol{A}^+ \end{aligned} \tag{8}$$

The symmetry of these two conditions clearly shows that $\boldsymbol{A}$ is a generalised inverse of $\boldsymbol{A}^+$. Furthermore, we see that $(\boldsymbol{A}^+)^\top$ is a generalised inverse of $\boldsymbol{A}^+$. In addition, $(\boldsymbol{A}^+)^\top$ is a generalised inverse of $\boldsymbol{A}^\top$. It is also clear that, if $\boldsymbol{A}$ is a nonsingular square matrix, $\boldsymbol{A}^{-1}$ satisfies the two conditions (8). Now consider an $n \times k$ matrix of regressors, $\boldsymbol{X}$. We can easily show that the generalised inverse of $\boldsymbol{X}$ is the matrix $(\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top$. It follows at once that the vector `betahat` defined earlier by the command

```
mat betahat = (X'*X) inv * (X'*y)
```

could have been defined in a simpler way using

```
mat betahat = X inv * y
```

EXERCISES:

Consider again the example of section 2.2 on collinear variables. Generate a matrix containing the constant and the four seasonal variables, and compute the "parameter estimates" using the generalised inverse of the matrix. Then compare your results with those provided by the `ols` command. It is unlikely that the parameter estimates are the same. But if you use the two parameter vectors to compute the fitted values, or the residuals, the results should be identical.

To invert a matrix `A`, we have until now used the notation `A inv` rather than `A^(-1)`. But the second notation is also valid: Both the commands

```
mat Z = A inv
mat ZZ = A^(-1)
```

give the same matrix, provided `A` is a square matrix. But the notation is more general. In fact, if the value of the scalar variable `n` is a positive, negative, or zero integer, the value of expression `A^n`, if `A` is a square matrix, is the matrix `A` to the power `n`. Thus, `A^0` is an identity matrix, `A^1` is the matrix `A` unchanged, `A^2` is the matrix `A` squared, `A^(-2)` is the square of `A inv`, and so on. If the power `n` is not an integer, the biggest integer (algebraically) smaller than `n + 0.1` will be used.

If `A` is a series, or, equivalently, a single column matrix with more than one row, then `A^n` is a series composed of the elements of `A` raised to the power of `n`. In other words, the effects of the commands `mat Z = A^n` and `gen Z = A^n` are identical.

* * * *

In particular, if we try to compute the non-integer power of a negative element, the result will be zero.

* * * *

If we replace the power `n` by a series, or a matrix, rather than a scalar, the element $(1, 1)$ of the series or of the matrix will be used. Finally, if, in the command `mat Z = A^n`, `A` is neither a square matrix nor a series, an error message will be displayed and the command will not be executed.

The command `gen Z = A^B` always has a meaning, even if `A` and `B` are matrices. The general rule is quite simple: Every column of `A` is treated separately. Consequently, the commands:

```
mat A = colcat(a1, a2, a3)
gen Z = A^n
```

are equivalent to the commands:

```
gen z1 = a1^n
gen z2 = a2^n
gen z3 = a3^n
mat Z = colcat(z1, z2, z3)
```

* * * *

Similarly, `set Z = A^B` always has a meaning. However, every series and every matrix will be treated as having a single row.

* * * *

EXERCISES:

Generate a matrix of seasonal variables:

```
sample 1 50
gen s1 = seasonal(4,1)
gen s2 = seasonal(4,2)
gen s3 = seasonal(4,3)
gen s4 = seasonal(4,4)
mat S = colcat(s1, s2, s3, s4)
```

What will be the results of the following operations:

```
mat Z = S^3
gen Z = S^3
gen Z = S^(-3)
gen Z = S^0
gen Z = S^s1
gen Z = S^s2
gen Z = S^S
```

and why?

2. Matrix Elements and Blocks

It is often necessary to assign numerical values to matrix elements on a one-to-one basis. Or to pull out an element from a matrix or a series. Or even to pull out a block, that is, a sub-matrix, of a bigger matrix. All this can be done using the `set` and `mat` commands.

First, if we wish to change the value of a particular element of a series, `y` for example, the appropriate command is

```
set y(i) = x
```

The subscript `i`, as well as the value `x`, will be computed under the rules of every `set` command, that is to say, as if both of the variables `smplstart` and `smplend` were equal to 1. Normally, the result of the computation of the subscript `i` has to be a positive integer. If this is not the case, the result will be replaced by $\max(1, [\texttt{i} + 0.1])$, where $[x]$ denotes the biggest integer smaller than the argument between the brackets. But watch out! Unlike the `set y = x` command, which can quite easily be used to generate scalar variables `y`, the series `y` must exist before we can assign values to its elements. If the series `y` does not exist, an error message will be displayed. However, if we try to assign a value to an element that does not exist, the command will merely have no effect.

A similar syntax is used for the elements of a matrix. To change the value of the element (i, j) of a matrix, we can use the following command:

```
set A(i,j) = x
```

The rules stated above also apply when evaluating the subscripts `i` and `j`. If we forget about the first subscript, it is the element i of the first row that will be changed. If we put too many subscripts, as for example in

```
set A(i,j,k) = x
```

the effect is similar to that of

```
set A(i) = x
```

or to that of

```
set A(i,1) = x
```

The elements of series and matrices can be used exactly like scalar variables in algebraic expressions. For example, in a `set` command,

```
set a = y(i)
```

a scalar variable `a` is generated, after having been deleted if it already existed, and receives the value of `y(i)`. Once again, the computation of the subscript `i` is done according to the rules stated above. Obviously, to avoid syntax errors, it is necessary for the variable `y` to exist. If `y` has several columns, it is the first one that will be selected by the command, and if the element does not exist, the assigned value will be zero. To access an element of a column other than the first one, the appropriate command is:

```
set a = A(i,j)
```

The notations `y(i)` and `A(i,j)` can also be used in the `gen` and `mat` commands wherever a scalar variable can be expected.

* * * *

We can obtain an unexpected result if we use a single subscript to access a variable that has several columns. Such results may be useful, but normally there exists a better way of obtaining the same results: refer to the next paragraph.

* * * *

Thus, the commands

```
sample 1 20
gen y = A(2,3)
```

generate a vector `y`, with 20 elements, each of which is equal to the value of element `A(2,3)` if it exists, and to zero otherwise.

It is possible to extract submatrices of a pre-existing matrix. For example, if we wish to eliminate the first column of an $n \times 3$ matrix, we can use this command:

```
mat A = A(1,n,2,3)
```

What this means is that the matrix `A` will be replaced by the block of the pre-existing matrix defined from the first to the n^{th} rows, and from the second to the third columns.

* * * *

Obviously, it is not necessary to write over the pre-existing matrix: one could just as easily have

```
mat B = A(1,n,2,3)
```

which will result in the construction of a new matrix.

* * * *

The four subscripts of this sort of expression are evaluated according to the usual rules. The first one corresponds to the first row of the submatrix to extract, the second to the last row, the third to the first column, and the fourth to the last column. We could have chosen another system, but alas, some choices are arbitrary!

If the subscript of the last row is less than that of the first, a single row, the one corresponding to the subscript of the first row, will be selected. The same holds for the columns. As usual, rows and columns that do not exist will be replaced by zeros.

Expressions of the form `A(i,j,k,l)` can only be used in the *right-hand side* of a `mat` command. These expressions are recognised by neither the `set` nor the `gen` command, nor when entered in the left-hand side. Thus, the command

```
mat A(i,j,k,l) = ...
```

will result in a syntax error. To insert submatrices into an existing matrix, we must use the functions `colcat` and `rowcat`. It is not always very convenient; in a later version there will perhaps be other possibilities ...

EXERCISES:

The file `sur.dat` contains 50 observations for the five variables $\boldsymbol{y}_1$, $\boldsymbol{y}_2$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$. We wish to estimate the system of equations

$$\begin{aligned} \boldsymbol{y}_1 &= \alpha_1 + \beta_{11}\boldsymbol{x}_1 + \beta_{12}\boldsymbol{x}_2 + \boldsymbol{u}_1 \\ \boldsymbol{y}_2 &= \alpha_2 + \beta_{22}\boldsymbol{x}_2 + \beta_{23}\boldsymbol{x}_3 + \boldsymbol{u}_2. \end{aligned} \tag{9}$$

In section 9.8 of DM, we see that we can make use of a **Stacked regression**, which can be written in matrix notation as

$$\begin{bmatrix} \boldsymbol{y}_1 \\ \boldsymbol{y}_2 \end{bmatrix} = \begin{bmatrix} \boldsymbol{\iota} & \boldsymbol{x}_1 & \boldsymbol{x}_2 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{\iota} & \boldsymbol{x}_2 & \boldsymbol{x}_3 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \beta_{11} \\ \beta_{12} \\ \alpha_2 \\ \beta_{22} \\ \beta_{23} \end{bmatrix} + \begin{bmatrix} \boldsymbol{u}_1 \\ \boldsymbol{u}_2 \end{bmatrix}. \tag{10}$$

(The vector $\boldsymbol{\iota}$, of which every element is equal to 1, corresponds to the constant.) According to the theory of stacked regressions, the estimates given by ordinary least squares applied to (10) are identical to the ones given by OLS applied separately to the two equations of (9). Construct the matrices of (10) by hand in order to demonstrate the result. It is useful to know that, in order to generate a 50×1 vector, of which all elements are zero, it is sufficient to run

```
sample 1 50
gen zero = 0
```

In practice, we might have to regress several variables on the same set of regressors. For example, in the previous exercise, we could have regressed the two variables $\boldsymbol{y}_1$ et $\boldsymbol{y}_2$ on all of the regressors, that is, the constant, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$. Similarly, if we wish to carry out an estimation by 2SLS, it is necessary to regress beforehand all the explanatory variables on all the instruments. To decrease the amount of programming, the `ols` and `iv` commands recognise matrices as dependent variables. For each column of the matrix, estimation will be carried out, either by OLS or by IV.

EXERCISES:

Run the following program:

```
sample 1 50
read sur.dat y1 y2 x1 x2 x3
mat Y = colcat(y1, y2)
gen c = 1
mat colcat(c, x1, x2, x3)
ols Y X
```

to see the result of an `ols` with a matrix of dependent variables. Note that it is necessary to generate a constant explicitly in this exercise. Indeed, a constant is automatically generated only if a *regression* by `ols` or by `iv` is carried out.

If several estimations are carried out simultaneously,[5] as in the preceding exercise, some of the variables generated or updated by ***Ects*** will have dimensions differing from those of the ordinary case. Let us first consider the case of the `ols` regressions. Let n denote the sample size, k the number of regressors, and m the number of dependent variables. The variable `coef` now becomes a $k \times m$ matrix, the m columns being the estimated parameters of the m regressions. The variables `fit` and `res` become $n \times m$ matrices: once again, the m columns correspond to the m regressions.

* * * *

This observation does not fully hold if `smplstart` is different from 1. The number of rows of `fit` and `res` is equal to `smplend`, but the first `smplstart` − 1 rows are zero.

* * * *

The three variables `ssr`, `sse`, and `sst`, which usually are scalars, are now $m \times m$ matrices. If we let $\boldsymbol{Y}$, $\hat{\boldsymbol{U}}$, and $\hat{\boldsymbol{Y}}$ denote respectively the $n \times m$ matrices of dependent variables, of residuals, and of fitted values, in ***Ects*** notation `Y`, `res`, and `fit`, then the variables `ssr`, `sse`, and `sst` are defined as follows:

$$\begin{aligned} \texttt{ssr} &= \hat{\boldsymbol{U}}^\top \hat{\boldsymbol{U}}; \\ \texttt{sse} &= \hat{\boldsymbol{Y}}^\top \hat{\boldsymbol{Y}}; \quad \text{and} \\ \texttt{sst} &= \boldsymbol{Y}^\top \boldsymbol{Y}. \end{aligned}$$

[5] We then refer to these estimations or regressions as **multivariate** regressions.

The variable R2 is transformed into an $m \times 1$ vector, with the successive elements being the successive R^2 of the m regressions. errvar, which is simply an estimate of $\hat{\sigma}^2$ in the univariate case, becomes an $m \times m$ matrix defined exactly as in the univariate case:

$$\texttt{errvar} = \frac{\texttt{ssr}}{n-k}.$$

Finally, stderr and student are, exactly as coef, $k \times m$ matrices, defined in an analogous way.

Some ***Ects*** variables keep their ordinary aspects, even in the multivariate case. An example is the XtXinv matrix, because the regressors are precisely the same in all the regressions. For the same reason, the hat vector remains unchanged relative to a simple regression.

Trap! For purposes of compatibility with the first version of ***Ects***, the matrix vcov corresponds in the univariate case to the expression $\hat{\sigma}^2(\boldsymbol{X}^\top\boldsymbol{X})^{-1}$. In the multivariate case, $\boldsymbol{X}^\top\boldsymbol{X}$ is the same matrix for all the regressions, but $\hat{\sigma}^2$ is different. Because it would be difficult to define a whole vector of matrices, the vcov variable in the multivariate case corresponds to the *first* regression only. If, for example, we want to obtain the matrix corresponding to the second regression, we should compute it in the following way:

```
mat vcov2 = errvar(2,2)*XtXinv
```

If a multivariate estimation is carried out using iv, things are similar. The R2 variable is not generated, because the R^2 has no significance in this context. The role of the matrix XtXinv is played by XtPwXinv: once again this matrix is independent of the number of dependent variables. And the trap is the following: vcov contains only the estimated covariance matrix of the estimated parameters of the *first* regression.

Exercises:

Run separately the two regressions of the previous exercise and proceed with the computations that are required to obtain all the vectors and matrices computed by the multivariate procedure.

3. Other Operations

It is sometimes necessary to carry out matrix operations that are more sophisticated than the ones considered hitherto. In this section, we describe the functionality which is at our disposal in version 2 of ***Ects***.

* * * *

Reading this section is not essential. You can skip to the next chapter.

* * * *

Firstly, a simple operation. In an algebraic expression, if `A` is a square $n \times n$ matrix, `diag(A)` is an $n \times 1$ vector composed of the diagonal elements of the matrix `A`. If `A` is not a square matrix, then `diag(A)` will be a vector of which the number of rows is equal to the number of rows of `A`. Elements that don't exist will be as usual replaced by zeros.

EXERCISES:

Let $\boldsymbol{X}$ denote a matrix of regressors for a sample of reasonable size n. The $n \times n$ orthogonal projection matrix $\boldsymbol{P}_X$, can then be generated by the command

```
mat Px = X*X inv
```

by using the generalised inverse of $\boldsymbol{X}$. Run a regression using $\boldsymbol{X}$ as the regressor matrix, and check that the series `hat` is the same as the one given by `diag(Px)`.

We sometimes need to compute the determinant of a matrix. A determinant is of course a scalar variable. The expression `det(A)` can be used in all algebraic expressions where a scalar variable is acceptable, and the expression `det(A)` can also be subject to the `set` command. For the determinant of a matrix to be defined, the matrix has to be square. If we write `det(A)` for a non-square matrix `A`, the result is zero.

In order to avoid possible complications, it is preferable to use the determinants within a `mat` command, rather than within the `gen` and `set`. Even though there exists rule that establish the result of `det` under `gen` and `set`, they are not simple. (And they will not be explained here!)

In a previous exercise, we briefly discussed the notion of a stacked regression. Usually, we use such a regression to carry out the estimation of a system of equations. Such a system is referred to as a **SUR system** (from *seemingly unrelated regressions*). The theory of SUR systems is set out in sections 9.7 – 9.9 of DM. According to this theory, we need an $m \times m$ matrix $\boldsymbol{\psi}$, where m is the number of equations forming the system, such that

$$\boldsymbol{\psi}\boldsymbol{\psi}^\top = \boldsymbol{\Sigma}^{-1},$$

where $\boldsymbol{\Sigma}$ is the $m \times m$ covariance matrix of the random elements associated with the m equations. Let $\hat{\boldsymbol{U}}$ denote the $n \times m$ matrix of residuals from the OLS estimation of the equations taken separately (n =sample size). The estimate of $\boldsymbol{\Sigma}$ is then

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

which we can compute easily, as well as its inverse, $\hat{\boldsymbol{\Sigma}}^{-1}$. Usually, it is convenient to require that $\boldsymbol{\psi}$ should be a **triangular** matrix. The function `uptriang` enables us to compute $\boldsymbol{\psi}$ under the upper-triangular form, that is,

$$\boldsymbol{\psi} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ 0 & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & a_{mm} \end{bmatrix}.$$

Let `Siginv` denote the matrix $\boldsymbol{\Sigma}^{-1}$. The command that generates $\boldsymbol{\psi}$ is then

```
mat psi = uptriang(Siginv)
```

EXERCISES:

Consider again the data contained in the `sur.dat` file and proceed with an estimation of the (9) system using the SUR method.

* * * *

The answer to this question is provided in the file `sur.ect`. Of course, it is preferable to use this file only after having completed the exercise.

* * * *

The last operation we consider in this chapter is fairly complicated. In fact, it is the central operation in the computation of OLS estimates and in the computation of a generalised inverse. It is the **Singular Value Decomposition**, or the **SVD**. We do not often have to use this operation directly, but sometimes it turns out to be essential. Here is the shortened theory of this decomposition. (The algorithm which performs the SVD, and some elements of the theory, are given in the relevant section of Press, Teukolsky, Vetterling, and Flannery (1986).)

Let $\boldsymbol{X}$ denote an $n \times k$ matrix, where $k \leq n$. We can show the existence of matrices $\boldsymbol{U}$, $n \times k$, $\boldsymbol{W}$, $k \times k$, and $\boldsymbol{V}$, $k \times k$, such that

$$\boldsymbol{X} = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^\top, \tag{11}$$

where, moreover,

$$\boldsymbol{U}^\top\boldsymbol{U} = \mathbf{I}_k; \qquad \boldsymbol{V}^\top\boldsymbol{V} = \boldsymbol{V}\boldsymbol{V}^\top = \mathbf{I}_k,$$

and $\boldsymbol{W}$ is a diagonal matrix, with non-negative diagonal elements. Indeed, these elements are the square roots of the eigenvalues of the matrix $\boldsymbol{X}^\top\boldsymbol{X}$, which is symmetric and positive semi-definite. The generalised inverse of $\boldsymbol{X}$ is computed as follows :

$$\boldsymbol{X}^+ = \boldsymbol{V}\boldsymbol{W}^+\boldsymbol{U}^\top,$$

from which we see that the usefulness of the SVD is that the (generalised) inverse of $\boldsymbol{W}$ is quite simple to compute: the diagonal elements must just be inverted one by one. If some diagonal elements of $\boldsymbol{W}$ are zero, we set the corresponding elements of $\boldsymbol{W}$ equal to zero.

* * * *

Given the limitations of numerical calculation on computers, a "zero" value is a value lower than a critical value close to zero. For the current version of ***Ects***, this critical value is close to 10^{-8}.

* * * *

Let `X` denote an $n \times k$ matrix. To carry out the decomposition (11), the relevant command is:

```
svdcmp X
```

The results of the command are not printed in the output file. However, three matrices are generated or updated if they already exist: `SVDU`, `SVDW`, and `SVDV`. The dimensions of these matrices are, respectively, $n \times k$, $k \times 1$, and $k \times k$. But watch out! The diagonal matrix $\boldsymbol{W}$ is stored in the *vector* form `SVDW`, of which the components are the diagonal elements of $\boldsymbol{W}$. If the matrix $\boldsymbol{X}$ has more columns than rows, that is if $n < k$, it will then be transposed before the computation of $\boldsymbol{U}$, $\boldsymbol{W}$, and $\boldsymbol{V}$, and the results will satisfy

$$\boldsymbol{X}^\top = \boldsymbol{U}\boldsymbol{W}\boldsymbol{V}^\top.$$

EXERCISES:

Take the variables $\boldsymbol{y}$, $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, and $\boldsymbol{x}_3$ from the `ols.dat` file and generate a 100×4 matrix $\boldsymbol{X}$, composed of the constant and the three explanatory variables. Compute the parameter estimates for regression (1), as well as the estimated covariance matrix of these estimates, without the use of the `ols` command. Also compute the `hat` vector, which contains the diagonal elements of the projection matrix $\boldsymbol{P}_{\boldsymbol{X}}$.

Chapter 4
Nonlinear Estimation

1. Nonlinear Regression

Ects is capable of performing several kinds of nonlinear estimation. In this section we will see how to perform nonlinear estimation of the simplest kind, **nonlinear least squares**, or **NLS**. The nonlinear regression model is written as

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

(See chapters 2 and 3 of DM.) The dependent variable $\boldsymbol{y}$ and the random vector $\boldsymbol{u}$ are each represented by an $n \times 1$ vector, just as for OLS. However, the **regression function**, $\boldsymbol{x}(\boldsymbol{\beta})$, is generally not a linear function of the $\boldsymbol{\beta}$ parameters anymore. In the context of the Gauss-Newton artificial regression, the GNR, we use a matrix of the same dimensions as the matrix of regressors within the framework of the linear regression. This matrix, written as $\boldsymbol{X}(\boldsymbol{\beta})$, is defined by the following relationship:

$$\boldsymbol{X}_{ti}(\boldsymbol{\beta}) \equiv \frac{\partial x_t}{\partial \beta_i}(\boldsymbol{\beta}). \tag{12}$$

Indeed, in the case of the linear regression model, we have $x_t(\boldsymbol{\beta}) = \boldsymbol{X}_t\boldsymbol{\beta}$, and, for any $\boldsymbol{\beta}$, the matrix of regressors $\boldsymbol{X}$ is equal to $\boldsymbol{X}(\boldsymbol{\beta})$.

The algorithm used by ***Ects*** to minimize the sum of squared residuals of a nonlinear regression is described in Press *et al* (1986), under the name of **method of Marquardt**, or of Levenberg-Marquardt. For the algorithm to perform correctly, we need the elements of the matrix $\boldsymbol{X}(\boldsymbol{\beta})$. To avoid unnecessary complications, the `nls` command requires that the user provide them at the same time as the regression function itself. In order to understand how to specify a nonlinear model, look at the file `nls.ect`. Here are the contents of this short file:

```
sample 1 50
read nls.dat y x1 x2
ols y c x1 x2
set alpha = coef(1)
```

```
set beta = coef(2)
nls y = alpha + beta*x1 + (1/beta)*x2
deriv alpha = 1
deriv beta = x1 - x2/(beta*beta)
end
gen e = y - alpha - beta*x1 - (1/beta)*x2
gen ralpha = 1
gen rbeta = x1 - x2/(beta*beta)
ols e ralpha rbeta
quit
```

The file `nls.dat` must contain at least 50 observations on at least three variables. (This is the case for the file provided by ***Ects***.) The model that we wish to estimate by nonlinear least squares is:

$$\boldsymbol{y} = \alpha + \beta \boldsymbol{x}_1 + \frac{1}{\beta}\boldsymbol{x}_2 + \boldsymbol{u}. \tag{13}$$

The regression function is therefore $\alpha + \beta \boldsymbol{x}_1 + (1/\beta)\boldsymbol{x}_2$, which depends on two parameters, α and β. Regression (13) can be regarded as a *linear* regression subject to a nonlinear constraint: If in the linear regression

$$\boldsymbol{y} = \alpha + \gamma_1 \boldsymbol{x}_1 + \gamma_2 \boldsymbol{x}_2 + \boldsymbol{u} \tag{14}$$

we impose $\gamma_1 \gamma_2 = 1$, we end up with (13). It is therefore appropriate to estimate (14): that is the purpose of the command

```
ols y c x1 x2
```

Now for the nonlinear estimation. Before we are able to specify the model (13) in ***Ects***, it is *absolutely necessary* to give the algorithm which minimizes la somme des carrés des résidus a starting point.[6] In other words, the parameters α and β of the regression function must be defined by `set`, and we must assign appropriate values to them. Here, a completely intuitive starting point was chosen: we assigned the estimates obtained by the linear regression to α and β.

* * * *

Recall: `coef` is one of the series updated by ***Ects*** after each linear regression: see sections 1.2, 2.3, and 3.2.

* * * *

Now it is time to use `nls`. The command consists of several lines:

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

[6] No longer, although it is always advisable to do so. If no starting values are given, ***Ects*** assigns values of zero.

The first line is used to specify the dependent variable and the regression function. Unlike the `ols` command, we must separate these two elements by the equals sign =. The rest of the line follows the same syntax as used by `gen`. Indeed the digestive system of ***Ects*** has only one algorithm for the analysis of algebraic expressions, although this algorithm can be adapted to the needs of the `set`, `mat`, and `gen` commands. The reason we use the version adapted to `gen` is simple: The expression that follows the first part of the command line, that is, the expression that follows `nls y =` , is interpreted as a *series*, because, for each observation, there is a corresponding regression function.

Next, for each of the parameters in the model, a line is needed that starts with `deriv`, followed by the name of one of the parameters of the model, then an equality sign =, and, finally, an algebraic expression equal to the partial derivative of the regression function with respect to the parameter in question. All of these partial derivatives, like the regression function, will be evaluated according to `gen`, like series. The order of the parameters is not essential; however, it is useful to set up the order of the parameters in the printout. After listing the algebraic expressions of the derivatives, one more line is needed, which is just the word `end`. Why do we add this final line? Because, apart from the fact that it removes certain difficulties that the programmer would otherwise face, this convention allows us to incorporate in the regression function scalar variables whose values remain unchanged. In other words, the minimisation of la somme des carrés des résidus will be carried out with respect to only those parameters which appear in the list of derivatives.

The last group of commands implements the linear GNR[7] associated with the model estimated by NLS. If everything was done correctly, the parameter estimates will have a value of zero, within the margin of error. It should be noted that after an `nls` estimation, the parameter estimates, here α and β, are the values obtained by estimation; the old values, those of the starting point, are lost.

EXERCISES:

The `nlsols.ect` file reuses the commands of `nls.ect`, but with the saved starting point parameters. At the end of the nonlinear procedure, a new regression is specified by `nls`. It is completely equivalent to the linear regression from the beginning of the program. We see that it is possible to carry out linear regressions by `nls`. Increase the value of the `TOL` variable and note the consequences with regard to the GNR and of the linear regression carried out by `nls`.

Study the `arnls.ect` file. We use the same data that was used for `ar.ect` (see the exercises in section 2.1). This time, estimation of a model with AR(1) errors is carried out directly by a nonlinear procedure. Try to understand the link between the different procedures of the two files.

[7] Gauss-Newton Regression

Take the model (1) again and the data from `ols.dat`. Impose and then test the nonlinear restriction that $2\beta_1\beta_3 + \beta_2 = 0$.

The variables created or updated by `nls` are almost identical to those which are produced by `ols`. Once again, `coef` is a vector that contains the parameter estimates, in the order of the partial derivatives given by the rows of `deriv`. It should be noted that the `nls` command can accept only one dependent variable, unlike `ols` and `iv`. The result is that the variables `ssr`, `sse`, `sst`, `R2`, `errvar` are scalars, while `fit`, `res`, `stderr`, and `student` are series. The `vcov` matrix is unambiguously the estimated covariance matrix of the parameter estimates, that is, the matrix $\hat{\sigma}^2(\hat{\boldsymbol{X}}^\top\hat{\boldsymbol{X}})^{-1}$, where $\hat{\sigma}^2 =$ `errvar`, and $(\hat{\boldsymbol{X}}^\top\hat{\boldsymbol{X}})^{-1} =$ `XtXinv`.

* * * *

Recall: $\hat{\boldsymbol{X}} \equiv \boldsymbol{X}(\hat{\boldsymbol{\beta}})$.

* * * *

It is clear from all of these definitions that the variance estimates, standard errors, and t statistics, are *asymptotic*. The `nobs` and `nreg` scalars are defined like those of `ols`, but, in the nonlinear context, `nreg` corresponds to the number of parameters rather than to the number of explanatory variables. Finally, there is a new scalar, `niter`, which is defined only for nonlinear estimations. Its value is the number of iterations that was necessary for the minimisation algorithm to converge.

If the number of parameters is considerable, it may be the case that we will need a great many iterations. Compared to linear estimation, the computing time for the minimisation algorithm for nonlinear estimation is already high. Consequently, ***Ects*** stops after having carried out the number of iterations specified by the internal variable `maxiter`. The default value of this variable is 20, but there is the possibility of changing this value by means of a simple `set`. For example, if we want to allow ***Ects*** to run up to 100 iterations, we do

```
set maxiter = 100
```

before starting the nonlinear estimation procedure.

After running `maxiter` times, ***Ects*** will ask the user if s/he wants to continue. ***Ects*** will ask the following question:

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

where n is the number of iterations carried out at the time the question is asked. If we answer `y`, the iterations will continue, either until the algorithm converges or until the `maxiter` iterations are achieved again.

EXERCISES:

Choose one of the nonlinear estimates that you previously generated, and set `maxiter` equal to a value lower than the iteration count announced by ***Ects***. You will see the course of the process described above.

2. Maximum Likelihood

The main differences between estimation by the **method of maximum likelihood** and estimation by NLS are hidden within ***Ects***. The commands that are given by the user to ask for the two kinds of estimates are very similar. However, the name of the command is (inevitably!) different: To start an estimation by maximum likelihood, we use the `ml` command. In spite of its name, `ml` can be used for other types of nonlinear estimations as well. Indeed, the command gives access to the algorithm used by ***Ects*** for the maximisation of functions of several variables. The algorithm is once again in the work of Press *et al* (1986), called the **Davidon-Fletcher-Powell**[8] algorithm, or simply the **DFP** algorithm.

To be able to use `ml`, it is necessary that an estimator be defined by the maximisation or minimisation of a **criterion function**, which is expressed as the sum of **contributions**. It is demonstrated in chapter 8 of DM that the ML estimator satisfies this condition. For the general theory of ***M*-estimators**, see chapter 17 of DM.

As usual, the best way to see how `ml` works is to look carefully at a particular example. The `ml.ect` file contains such an example. Once again, we are dealing with a linear regression, so the model is not difficult to formulate:

$$\boldsymbol{y} = \alpha + \beta_1 \boldsymbol{x}_1 + \beta_2 \boldsymbol{x}_2 + \boldsymbol{u}.$$

Just as with `nls`, it is necessary to assign starting values to all of the parameters with the `set` command.[9] Then the first line of the `ml` command consists of the formulation of the contribution from each observation to the **loglikelihood**. This line is rather long, but we will later see how to shorten it.

```
ml - log(sig) - (1/(2*sig*sig))*(y - alpha -
beta1*x1 - beta2*x2)^2
```

The rest of the command is completely analogous to what we do for `nls`. We must specify the partial derivatives of the contributions with respect to each parameter of the model. The end of the list of derivatives is once again announced by `end`. In the context of ML estimation, the parameter `sig`, or σ, the standard deviation of the random elements, plays the same role as all the other parameters.

It is known that the method of maximum likelihood is equivalent to least squares for regression models that have normally distributed random elements; we can create a second estimate to take account of this fact. The second `ml` command of the file:

[8] Mind the spelling of the first name!

[9] Again, this is no longer true.

```
ml -(y - alpha - beta1*x1 - beta2*x2)^2
```

implements this estimation; observe that σ doesn't appear, either in the contribution or in the derivatives.

Between the two sets of commands which define the two estimation procedures, you will find a new command, `beep`. Because a nonlinear estimation can take a long time, it's sometimes useful to the user to know when it has come to an end. The task of the `beep` command is to make an audible signal.

The second estimation will take less time than the first, precisely because of the elimination of the variable `sig`. If we take a closer look, we can see that the function that ***Ects*** had to maximise this time is simply, apart from the sign, the sum of the squared residuals of the regression. In theory, one should find the same parameter estimates by use of the `nls` command, and, because the regression is linear, of the `ols` command. The rest of the file carries out the verification of this fact.

To understand the results of an estimation done by `ml`, we can now analyze the (partial) contents of the output file. Here is what's produced by the first command:

```
Maximising a Sum of Contributions:

Number of iterations = 10

Parameter  Parameter estimate   Standard error    T statistic
alpha       22.3283060           22.8222496        -0.9783569
beta1        3.1952808            0.3247480         9.8392637
beta2        0.6045778            0.4740587         1.2753228
sig         56.1839983            5.5196279        10.1789467

Number of observations = 50
Number of estimated parameters = 4
Maximised value of criterion function = -226.4316049

Estimated covariance matrix (from numerical Hessian):
520.8550775   -5.4218378     3.7532409    -0.4943376
 -5.4218378    0.1054612    -0.1339667     0.0073355
  3.7532409   -0.1339667     0.2247316    -0.0078630
 -0.4943376    0.0073355    -0.0078630    30.4662924

Estimate from Outer Product of the Gradient:

850.0957659   -9.9960386     9.5327995    34.6640472
 -9.9960386    0.1674585    -0.2084992    -0.2685440
  9.5327995   -0.2084992     0.3067047     0.3688634
 34.6640472   -0.2685440     0.3688634    43.0321803
```

The first table strongly resembles what is given by `nls` or even `ols`. We have parameter estimates, standard errors, and t statistics, where, as is always the case in the context of nonlinear estimation, the last two have only an asymptotic justification.

Then, other than the scalars with an obvious meaning, we find

```
Maximised value of criterion function = -226.4316049
```

The number that is printed here is the criterion function (here the loglikelihood) evaluated at $\hat{\boldsymbol{\theta}}$, the parameter estimate which has just been obtained. In other words it is the maximised value of the criterion function.

* * * *

The function that we maximised is not exactly the loglikelihood of the model. To be completely correct, we must add the term $-(1/2)\log(2\pi)$ to each contribution. For the calculation of the LR statistic, this is not a problem because these two terms are identical in the two loglikelihood functions.

* * * *

Then come two different estimates of the covariance matrix of the parameter estimates. The second is easier to understand. Let $\boldsymbol{G}(\boldsymbol{\theta})$ be the **CG matrix** associated to a model (see section 8.2 of DM for details), that is, let

$$G_{ti}(\boldsymbol{\theta}) = \frac{\partial \ell_t}{\partial \theta_i}(\boldsymbol{\theta}),$$

where ℓ_t is the contribution to the loglikelihood from observation t, and where $\boldsymbol{\theta}$ denotes the parameters of the model. The **OPG estimator** of the covariance matrix of $\hat{\boldsymbol{\theta}}$ is defined as follows:

$$\mathrm{Var}_{\mathrm{OPG}}(\hat{\boldsymbol{\theta}}) = (\hat{\boldsymbol{G}}^\top \hat{\boldsymbol{G}})^{-1},$$

where $\hat{\boldsymbol{G}} \equiv \boldsymbol{G}(\hat{\boldsymbol{\theta}})$. This estimator converges to the expected value of the **outer product of the gradient** but its rate of convergence is not very impressive. Indeed, in this case, we can see that all the elements of the OPG estimator are bigger than those of the OLS estimator.

The other covariance estimate is stated in the following way:

```
Estimated covariance matrix (from numerical Hessian)
```

Unfortunately, this isn't completely true. The empirical Hessian is defined by the relation

$$H_{ij}(\boldsymbol{\theta}) = \frac{\partial^2 \ell}{\partial \theta_i \partial \theta_j}(\boldsymbol{\theta}).$$

The calculation of the right hand side of this equation requires that the second derivatives of the loglikelihood are known. ***Ects*** is very weak in differential calculus: it is even necessary to provide it with the first derivative.[10] The "Hessian" estimator given by ***Ects*** is thus only a numerical approximation to the true empirical Hessian. For details about the approximation, see Press *et al* (1986). We leave to the user the task of correctly calculating the Hessian.

[10] Again, this is not true for current versions of ***Ects***. See Volume 2 of the documentation for details.

* * * *

It would be still better to calculate the true **information matrix**. To this end, we can use the expected value of the OPG, or the negative of the Hessian. See section 8.7 of DM.

* * * *

The `ml` command creates or updates variables just like the other commands that carry out estimation. As usual, `coef` is the vector of parameter estimates, and as with `nls`, the variables that represent the parameters of the model are changed by `ml` – the values that were used as starting points are lost. `stderr` and `student` have their usual interpretations, as well as `nobs`, `nreg`, and `niter`. The two estimated covariance matrices are found in the variables `invhess` and `invOPG`. The CG matrix can also be recovered in `CG`. Finally, the series `lt` contains the vector of contributions to the criterion function, evaluated at $\hat{\boldsymbol{\theta}}$, and the scalar variable `lhat` is the sum of these contributions, identical to the `Maximised value of the criterion function`.

* * * *

The variable `CG` is also updated by the command `nls`. Its components are the elements of the matrix $\boldsymbol{X}(\boldsymbol{\beta})$ defined in (12).

* * * *

The length of the expressions of the contributions and their derivatives can be a problem. We saw in the above example that even a simple regression model generates rather heavy expressions. An important application can quickly become unmanageable. To avoid such difficulties, there is a command called `def`, which is very similar to what is often called in programming languages a **macro**. To understand how this works, let's reformulate our regression model. The following commands have the effect as those which were used for the first ML estimation.

```
def u = y - alpha - beta1*x1 - beta2*x2
def sig2 = sig*sig
ml - log(sig) - (1/(2*sig2))*u^2
deriv alpha = u/sig2
deriv beta1 = x1*u/sig2
deriv beta2 = x2*u/sig2
deriv sig = -(1/sig)*(1 - u^2/sig2)
end
```

We see that it is possible to replace a long expression by only one symbol. However, isn't it already possible to do so by using a `gen` command? Not in the same way. If, instead of the `def` command we use the `gen` command, the criterion function would depend on the parameters of the model only through `log(sig)` in the first term. The derivatives with respect to `alpha`, `beta1`, and `beta2` would not depend anymore on the parameters, and the derivative with respect to `sig` would depend only on the `sig` of the denominator. The

maximisation algorithm may well change the values of the parameters, but the criterion function will change only because of changes in `sig`.

What is the difference if we use `def`? This command doesn't create any new variable; it is simply used to put in the memory of the computer the *text* that follows the equality sign, and to assign the name (here `u` or `sig2`) to this bit of text. However, each time the computer has to evaluate an expression which contains the name, it seeks the corresponding text, and the expression is evaluated accordingly. This results in the criterion function, as well as the partial derivatives, being evaluated correctly each time: the current parameter values are used as they're updated by the maximisation algorithm.

Even if the expressions are not very long, it is often desirable to define them by `def` for clarity of the program.

* * * *

Of course, a "smart-alec" user can assign the same name by a `gen` and by a `def`. In this case, the results of the two commands remain in memory, one part a series or a matrix, and the other part a string of text. But it's the variable, series or matrix, which will be used in subsequent expressions.

* * * *

What would happen if our computer had insufficient memory? Unlikely as this is with present-day computers, the answer depends on the operating system we're using. With a bad operating system, it's possible that the computer will crash. In better circumstances, an error message appears, and the programme stops. If there is a risk of such a thing happening after a succession of estimations and precious calculations, we must proceed with care. If we launch the `del` command, followed by a list of variables (scalars, series, and matrices) or of macros (defined by `def`), the location in memory where these variables are located will be cleared, and the names which belonged to them will be erased from the table of symbols.

3. Generalised Method of Moments

The **generalised method of moments** is usually thought of as a sophisticated method. The details of the method, which are rather technical, are explored in chapter 17 of DM. However, there is one type of estimate, whose principles are relatively simple, which can be carried out (in programming) only by this method.[11]

[11] See Volume 2 for a better approach possible with current versions.

In chapter 7 of DM, we consider nonlinear regression models estimated by instrumental variables. The initial model is written, as usual, as

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

and we denote by $\boldsymbol{W}$ the matrix of instruments used. The estimator of this model is defined by the minimisation of the following criterion function

$$\big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big)^{\top} \boldsymbol{P}_W \big(\boldsymbol{y} - \boldsymbol{x}(\boldsymbol{\beta})\big).$$

* * * *

It is easy to see that in the linear case, the result of this minimisation is the usual IV estimator.

* * * *

Nevertheless, this criterion function is not expressed as a sum of contributions, unlike the loglikelihood; rather, it is a quadratic form. The generalised method of moments uses precisely the minimisation of a quadratic form.

To illustrate the functioning of the `gmm` command, let's take a closer look at the file `ivnls.ect`. We first undertake a nonlinear estimation by `nls` of the following model:

$$\boldsymbol{y} = \alpha + \beta \boldsymbol{x}_1 + \frac{1}{\beta} \boldsymbol{x}_2 + \boldsymbol{u}. \tag{15}$$

Then, the linear model of which (15) is the restricted version, that is,

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

is the object of an instrumental variable estimation (*i.e.* by `iv`), where the instruments are the constant, $\boldsymbol{x}_1$, and a new variable, $\boldsymbol{w}$.

The idea of the rest of the file is, initially, to redo the IV estimation using `gmm` instead of `iv`, and then to impose the restriction $\gamma = 1/\beta$, which results in a nonlinear model that we can estimate only by using `gmm`. Before passing to the `gmm` commands, there are some manipulations to carry out:

```
gen iota = 1
gen W = colcat(iota, x1, w)
def resid = y - b0*iota - b1*x1 - b2*x2
set b0 = alpha
set b1 = beta
set b2 = 1/beta
mat WtWinv = (W'*W)inv
```

For the starting point of the first GMM estimation, we use the parameter estimates of the nonlinear regression without instrumental variables. Moreover, we create a `resid` macro, which allows us to write the criterion function in a

simple way. Finally, we create the matrix $(\boldsymbol{W}^\top\boldsymbol{W})^{-1}$, where $\boldsymbol{W}$ is the matrix of all instruments.

Then, the estimation itself is:

```
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
```

We see that the syntax is once again identical to the `nls` and `ml` commands. After defining the criterion function, we must provide the partial derivatives of this function with respect to the all of the parameters to estimate, and then `end`.

The crucial difference relative to an `ml` command is that the algebraic expressions are read in the same way as are the expressions in a `mat` command, rather than in a `gen` command. Another difference of no crucial importance, but capable of provoking considerable worries if we forget about it, is that the criterion function is *minimised*. Apart from these differences, the algorithm used by `gmm` is the same as the one used by `ml`.

We see that the parameter estimates given by this first GMM estimation are identical to those provided by the previous `iv` command. However, other than those estimates, `gmm` doesn't have much else to say. The reason is that, in the general case, there is no precise link between the criterion function and the covariance matrix of the parameter estimates. It is therefore impossible to compute the standard errors, the t statistics, *etc.* However, the Hessian approximation to the criterion function is printed out, for it can be useful in computations that can result in an estimate of the true covariance matrix. Another often useful piece of information is the minimised value of the criterion function: This value is also printed out in the output file.

Here are the results of the first `gmm` commands in `ivnls.ect`:

```
Minimising a Criterion Function:

Number of iterations = 5

b0 = -16.6437115
b1 = 0.5864273
b2 = 2.7326305

Number of estimated parameters = 3
Minimised value of criterion function = 0.0000000

Inverse of Numerical Hessian of Criterion function:

 0.1002287   -0.0012027      0.0010858
-0.0012027    0.0000213     -0.0000266
 0.0010858   -0.0000266      0.0000408
```

EXERCISES:

The minimised value of the criterion function, which can be read in the table in the following line:

```
Minimised value of criterion function = 0.0000000
```

is zero. Why?

We then proceed to the nonlinear model estimation by instrumental variables. There is little that remains to be done; in fact, we simply must redefine the residuals and the derivative of the criterion function with respect to β:

```
def resid = y - b0 - b1*x1 - (1/b1)*x2
gmm resid'*W*WtWinv*W'*resid
deriv b0 = -2*iota'*W*WtWinv*W'*resid
deriv b1 = -2*(x1 - x2/(b1^2))'*W*WtWinv*W'*resid
end
```

EXERCISES:

Compute a correct estimate of the covariance matrix of the parameters `b0` and `b1` of the above model.

Like all other commands that carry out an estimation, `gmm` generates or updates some variables. There are fewer variables than usual because obtaining a direct estimation of the covariance matrix is impossible. `coef` contains the parameter estimates, and `invhess` contains the Hessian approximation. The scalar variables `nreg` and `niter` are defined by `gmm` as well as by `ml`. There is only one more variable created by `gmm`. It contains the minimised value of the criterion function, under the name `crit`.

Chapter 5

Interactive mode; File management

1. The Command and Output files

All the programs considered until now were communicated to ***Ects*** by use of a command file. This way of working is often very practical. People can exchange such files; an ASCII file is easy to modify if it contains errors,... But what if we wish to make a small calculation without having to create a command file? Or, we may want to stop the operation of a program halfway through to control, and if required to modify, the values of the variables. This is why the **interactive mode** exists.

It was previously seen (in section 1.1) that, if we launch ***Ects*** without the name of a command file, we find ourselves in the interactive mode. A sign appears, as follows

```
>
```

and the cursor positions itself just after it. If you make a mistake, you need only type `quit`, and you leave ***Ects***. But you can type any other ***Ects*** command, and it will be carried out.

EXERCISES:

Launch ***Ects*** in interactive mode and type the four commands of the file `ols.ect` to see its result.

The interactive mode normally does not have an output file. Indeed, the results of all the commands, which in the non-interactive mode would have been printed into the output file, are displayed on the screen. For certain uses this is ideal. If you left your calculator at home, for example, and you must do some heavy numerical calculations by hand, then ***Ects*** can be very useful. But for other operations and in particular, the econometric estimations for which the results are often bulky, it is preferable to create an output file that you can either view, or print afterwards, even if you want to enter the commands directly from the keyboard.

The `out` command is used to create an output file. In interactive mode or elsewhere, a command in this form

```
out result.out
```

creates a file whose name is `result.out` (if a file of this name already exists, it will be overwritten), and `result.out` will be used as output file.

When ***Ects*** is launched, the choice between the interactive mode and the non-interactive mode, which we also sometimes call the "batch" mode, is made according to whether the word `ects` is followed on the command line by the name of a command file.

* * * *

Recall: if the extension of the command file is `.ect`, then you only need to specify the name of the file, without the extension – see Section 1.1.

* * * *

If the interactive mode is chosen, there will not be an output file when the `out` command is not used, and the results will be displayed on the screen. If the name of a command file is provided, the output file bears the same name as the command file with the `.out` extension.

If the name of the output file chosen automatically by ***Ects*** is not appropriate, you can specify it as well as the name of the command file on the command line. This command line then takes the following form:

`ects` ⟨*command file*⟩ ⟨*output file*⟩

This has the same effect as if the first command of the ⟨*command file*⟩ were

`out` ⟨*output file*⟩

We can now formulate the general rule: if the command line contains only the word `ects`, we will be in interactive mode without an output file; if the name of a file is provided, this file is used as the command file; if the names of two files are provided, the first name is that of the command file, the second is that of the output file. If one of the names does not correspond to a file accessible by software, an error message will be posted and ***Ects*** will stop.

2. Work Contexts

To understand fully the other ***Ects*** commands which make it possible to manage input/output, it is useful to know that inside ***Ects*** there is at every instant a **work context**, which is comprised of four elements. These elements are firstly, the command file, in other words the input file, secondly, the output file, thirdly, the message file, and fourthly, a binary variable whose value is 1 if ***Ects*** is in interactive mode, and 0 if not.

By default, the **message file** is the screen. Whatever the output file may be, it is in the message file that pending commands and error messages will be

printed. This is why, each time that we run an ***Ects*** program, you can see the sequence of commands unfolding on the screen. If the message file isn't the screen, the only other possibility is the "null" file. The `null` file is a file which is used as the recycling bin. If we write to this file, nothing occurs; if we read this file, what we read is precisely nothing. The existence of such a file is often very useful. For example, you may not want to see the commands unfolding on the screen. If a program is very long, and if it is comprised of many commands, the computing time will be decreased when the echoing of commands is suppressed.

The relevant command is the `noecho` command. You need only insert this command in a command file in order to stop echoing to the screen. Only true error messages will be displayed. The effect of `noecho` is cancelled by the opposite command, `echo`. In interactive mode, the "echo" is removed automatically – if it is necessary to type commands manually, it is useless to have them appear again immediately afterwards. Therefore, the `echo` and `noecho` commands have no visible effect in interactive mode. However, if you use `noecho` in interactive mode and then return to the batch mode, the effect of the command will persist.

During the execution by ***Ects*** of the command file, two files are written, the output file and the message file. It is possible to halt the writing to the output file by the `silent` command. This command has the same effect as the command

```
out null
```

Indeed, all that would be normally intended for the output file is thrown in the recycling bin. To cancel a `silent` command one can use the `restore` command, whose effect is similar to that of the command

`out` ⟨*out file*⟩

where ⟨*out file*⟩ is the name of an output file other than `null`.

The differences between `silent` followed by `restore` on one hand, and `out null` followed by another `out` command on the other hand, are unimportant. But it should be noticed that, throughout the duration of a `silent` command, the name of the previous output file is saved so that it can be restored after a possible `restore` command, which is not the case if we use `out`. In particular, you should avoid doing an `out null` followed by a `restore`: the restored file would be simply `null`!

We mentioned above the possibility of interrupting the unfolding of a command file in order to carry out some controls or modifications. This is done by switching to interactive mode. To interrupt a command file, you need to insert in the command file the `interact` command at the point where you want it to stop. The existing context is saved, and you switch into interactive mode. Even if the output file is the `null` file, interactive mode is launched

normally. If you wanted to divert the results obtained in the interactive mode towards the recycling bin, it would be necessary to launch a new `silent`.

EXERCISES:

Insert the `interact` command in the `ols.ect` file after the `ols` command. Once in interactive mode, view the variables created by `ols`. To do this, you can use either `print` or `show`; the effects are identical, for obvious reasons.

If you carried out the previous exercise and if you aren't in interactive mode anymore, you will have seen that, after having typed `quit` to leave the interactive mode, a new `quit` was displayed before the end of the program. This second `quit` is the one contained in `ols.ect`.

Why is it necessary to do `quit` twice before leaving ***Ects***? Because the effect of `quit` is not, or is not always, to leave ***Ects***. The successive contexts created during unfolding of the sessions are stacked. We noticed above that if we switch into interactive mode by using `interact`, the existing context is saved. It would be better to say that the context is stacked, so that it will be found when the context which was stacked just on top of it is subsequently removed. Basically, the `quit` command removes a context.

When we are finished with the interactive mode, we type `quit`. The context which has been created to constitute the interactive mode is removed, and we find ourselves in the context which after the removal is on top of the stack. This means that if, for example, `interact` is typed when we are already in interactive mode, a new interactive context will be established, the previous context being pushed down on the context stack. Then, `quit` is typed, but we do not leave interactive mode, because when the context below is found, it is also an interactive context. We leave the second interactive context to get back to the first.

What if there aren't any more contexts? What if the stack is empty? Then the program quits, and this is why, if we create only one context, `quit` has the effect of finishing the execution of ***Ects***. In section 1.2, we noticed that we don't have to use `quit`. Indeed, if ***Ects*** arrives at the end of a command file, a `quit` will be generated automatically: The current context will then be removed from the stack, and if there isn't anything below, the program ends.

Because, besides ending the program, `quit` is used to leave interactive mode, there is a `batch` command, similar to the `interact` command, which is used to end the interactive mode. However, this command is completely equivalent to `quit`.

EXERCISES:

Launch embedded interactive contexts to see the effects that we have just described. While quitting these contexts, use the `batch` command to ensure that it functions exactly like `quit`, until the ending of the program.

How do you embed contexts if they are not interactive contexts, and why? To answer the second question first, it is so that a command file can call another. If, for example, the reading of your data is done in several stages, with possible transformations of the data read, it can be convenient to put the relevant commands in a file other than that which contains the commands that use the data for estimation purposes. In such a case, one can put in the file which carries out the estimation a command which calls the file containing the reading commands before moving on to the commands which launch the estimation.

The command in question is the `run` command. Its syntax is very simple:

`run` ⟨*command file*⟩

What happens is also very simple. The current context, interactive or not, is saved on the context stack and a new non-interactive context is created to execute the commands contained in the ⟨*command file*⟩. Once the execution of these commands is finished, we find the previous context on top of the stack. A called file can call yet another file, and so on. The only limitation is the memory of the computer, which must contain the context stack.

An example of the use of the `run` command is provided by the file `mrs.ect`. If you look at the contents of this file, it will be largely incomprehensible. But, at the beginning of the file, you will see that the file `pitch.ect` is called. This second file contains a whole series of definitions of the variables `C`, `D`, `G`, *etc*, which appear in `mrs.ect`. The two files use the `noecho` and `silent` commands to hide their operations.

Exercises:

Before reading further, run the `mrs.ect` file.[12]

The exercise that you have just carried out shows the capacities of the `beep` command. It was seen how this command is used to activate the beep of the computer. What we have just heard shows that the audio signal can take several forms. Let us look at the first commands of `mrs.ect`:

```
set t = 1.5
set b = 400
beep .5*G*t .25*b
beep .5*G*t .5*b
beep 0 .25*b
beep .5*G*t .25*b
beep C*t .9*b
```

[12] This may be a disappointing experience. Ten years ago, a program had direct access to the computer's speaker. These days, operating systems seldom allow this, and, worse, they all have their own ways of giving a program access to the speaker. Had this been a vital feature of ***Ects***, I would have done what was necessary to get round this, but it was just too much trouble.

```
beep 0 .1*b
beep C*t .9*b
beep 0 .1*b
```

The command `beep` is apparently able to take two arguments. The first is used to establish the pitch, the second the duration, of the sound emitted by the computer. So that the commands are at least partly connected to musical notation, the definitions contained in the file `pitch.ect` assign to variables `A`, `B`, `Bf`, `Cs`, *etc*, the appropriate frequencies.

The notations `s` and `f` correspond to **sharp** and **flat**. We can thus easily access all the notes of the scale. The variable `t`, used in `mrs.ect`, makes it possible to transpose the notes, by multiplying them by suitable factors. In the same way, the variable `b` makes it possible to choose a rate/rhythm or, more precisely, a tempo.

* * * *

For further information, talk to your piano teacher! The above material has been retained as a record of the history of the software, but no current versions of ***Ects*** emit pitched sounds, and there are no plans to change this.

* * * *

The true mission of ***Ects*** is certainly not to play music. Nowadays, we must be satisfied with the `pause` command. The audio signal emitted by this command is devoid of all sensuality. If in an ***Ects*** program we enter the `pause` command, without argument, the program stops running, a beep is emitted, and the following message is displayed:

```
Press a key to continue
```

This command thus makes it possible to examine the contents of the screen; something which is not always possible if the computer runs rather quickly.

If the `pause` command is followed by an argument, this argument will be evaluated similarly to arguments that are positive integers (see sections 2. and 2.3). Let n be the value of the argument. After a preliminary beep, the machine remains inactive during n seconds, after which there is a new beep, and the execution of the commands following `pause` continues.

3. Control of the Contents of the Output File

Up to this point, the only way we had to influence the contents of the output file was through the use of the `silent` and `restore` commands. You can also use the `text` and `put` commands.

Before looking at the effect of these commands in detail, notice that a traditional method is available that answers certain needs. If a line in an ***Ects***

command file starts with `rem`, the program jumps immediately to the following line, regardless of the rest of the command. Indeed, `rem` makes it possible to insert *remarks*, or *comments*, into the program. But the whole line will be like any other line, displayed on the screen and printed in the output file, unless it is prevented by a `silent` or a `noecho` command.

* * * *

In interactive mode, there is no echo, and there is normally no output file. However, in interactive mode, comments are not very useful.

* * * *

You can use comments introduced by `rem` to punctuate and document an output file.

The operations of the commands `text` and `put` can be apprehended more easily if we look at an example. Such an example is contained in the file `logit.ect`. The first part of this file implements an estimation of a **logit model**. The procedure does not use the `ml` or `gmm` commands, which could have been useful, but rather uses an **artificial regression** called the **Binary Response Model Regression**, or **BRMR**. This artificial regression is explained in chapter 15 of DM, section 15.4. For the moment, we skip the details of the method, in order to see how the results can be printed in the output file as if we had used a standard method such as `nls`.

Before starting the estimation process, we suppress the echo and printing to the output file by

```
silent
noecho
```

Afterwards come the following commands:

```
mat parms = rowcat(a, b1, b2, b3)
sample 1 4
gen T = parms/stderr
mat block = colcat(parms, stderr, T)
mat line1 = block(1,1,1,3)
mat line2 = block(2,2,1,3)
mat line3 = block(3,3,1,3)
mat line4 = block(4,4,1,3)
text
Estimation of Logit Model by BRM Artificial Regression

Number of iterations =
end
put i
text
Parameter       Estimate        Std error       T statistic

end
text a
```

```
end
put line1
text b1
end
put line2
text b2
end
put line3
text b3
end
put line4
text
Estimated covariance matrix:

end
put XtXinv
text

end
```

The variables `a`, `b1`, `b2`, and `b3` are the parameters of the logit model. After estimation, they are gathered in by `rowcat` into a row referred to as `parms`. Then, a 4×3 matrix, named `block`, is created by `colcat` to contain the estimated parameters, standard errors, and t statistics. Finally, the four successive lines of `block` are extracted and saved under the names `line`i, $i = 1, \ldots, 4$.

The commands which follow these operations are used to print the following tables in the output file:

```
Estimation of Logit Model by BRM Artificial Regression

Number of iterations =        5.0000000

Parameter      Estimate        Std error       T statistic
a            -2.5354420       1.4272198      -1.7764901
b1            0.0862519       0.0193141       4.4657555
b2           -0.1313120       0.0234716      -5.5945115
b3            0.0407852       0.0110060       3.7057062

Estimated covariance matrix:

 2.9930578      -0.0332194     0.0184295     0.0089981
-0.0332194       0.0005481    -0.0004965     0.0000522
 0.0184295      -0.0004965     0.0008095    -0.0001893
 0.0089981       0.0000522    -0.0001893     0.0001780
```

What has happened? Only two commands were used for the construction of these tables: `text` and `put`. The effect of `text` is to print to the output file all that follows the word `text`, until a line beginning with `end` is encountered. The very first `text`, which precedes the line

```
Estimation of Logit Model by BRM Artificial Regression
```

is followed by a blank line. This will result in a line also being skipped in the output file before the text is printed. Furthermore, we see lines like

```
text a
```

In such a case, `a` is printed immediately, without skipping a line.

After the first line of text, the one quoted in the preceding paragraph, there is another new line in the command file, reproduced in the output file. Then, we have

```
Number of iterations =
end
```

As we can see in the output file, the very last skipped line, the one which precedes `end`, is not reproduced. The reason is that we often wish to print, on the same line of the output file, something else than a text which we can provide literally. Indeed, it is here that the value of the variable `i` must be printed. But skipping a line is necessary in the command file, so that the `end` is read like a command rather than the continuation of the text.

* * * *

> This means that you cannot begin a line of `text` with the word "end". Watch out! You can, however, begin the line with one or more white spaces, or a horizontal tab, followed by the word "end". In order for this word to announce the end of a `text`, it must be placed at the very beginning of the line.

* * * *

The `put` command is used to put the *values* of variables in the output file. We saw in section 1.3 that this command writes to the output file, and that it does not precede the scalars, vectors, and matrices which it prints by their names. We can now see why this command is useful. In order to use the command correctly, we should learn how to use its other two properties. Firstly, the printing will always be followed by a line skip. This explains why the estimated parameters, standard errors, and t statistics were gathered in rows before being printed. Without passing through this intermediate stage, only an arrangement by columns would have been possible. Second, the `put` command will write to the output file even if preceded by a `silent` command. This property is absolutely necessary. If we do not use `silent` before using `put` and `text`, then the command lines will be printed. For example, if we did

```
text a
end
put line1
```

without using `silent` first, we would get

```
text a
a               put line1
-2.5354420      1.4272198       -1.7764901
```

in the output file!

There are some aspects in the excerpt of the command file which are invisible but have significant consequences for the printing to the output file. In fact, lines like

```
text a
```

are followed by two horizontal tabulation characters (character 9 in ASCII.) This is used to align the printed tables. ***Ects*** itself deals with the insertion of these characters between the elements of the vectors and matrices.

EXERCISES:

Try to regenerate the tables printed in the output files by the `ols` command by using the `text` and `put` commands. Do not forget to work `silent`ly. In section 4.3, we looked at the estimation of the model (15) by use of the `gmm` command, which does not provide much information. However, it is possible to find an estimate of the covariance matrix of the estimated parameters: See sections 7.6 and 7.7 of DM. Build a table which presents all the usual information, as if the estimate had been carried out by `iv`.

Chapter 6
Monte Carlo Experiments

1. The Programming of Loops

The development of computers has made numerical experiments an integral part of scientific research. ***Ects*** is fully involved in the use of these experiments, which are often referred to as **Monte Carlo experiments**. An overview of these experiments can be found in chapter 21 of DM.

There are two essential ingredients in the recipe for a Monte Carlo Experiment: The generation of pseudo-random numbers, which we shall consider later on, and a large number of repetitions of a set of commands, a set that constitutes what we refer to as a **simulation**.

The repetition of a set of commands is carried out in programming by **loops**, that is, blocks of commands which are executed several times, until a certain condition is no longer met. To implement a loop in ***Ects***, we use the `while` command. Here is a scheme representing a loop:

```
while ⟨expression⟩
        ⟨block of commands⟩
end
```

After having read the command `while`, ***Ects*** then evaluates the ⟨*expression*⟩ that follows the command. If the value of the command is different from zero, the ⟨*block of commands*⟩ will be executed. Otherwise, ***Ects*** jumps to the end of the loop, signalled by the command `end`.

Let's take a closer look at the type of expressions that we can use after a `while` command, for example, if you wish to run the loop ten times. In this case a quite simple way to proceed is to make use of a scalar variable to keep track of the **iterations** of the loop. Before `while`, we enter

```
set i = 0
```

for example, and the first command of the block that forms the loop will be

```
set i = i+1
```

* * * *

The variable `i` can be useful for several purposes. If the ten iterations are used to fill in the ten elements of some vector `x`, we then just have to enter the command

`set x(i) =` ⟨*computed value*⟩

in the block.

* * * *

We reach the end of the execution of the loop when `i` = 10. That is, the loop is repeated as long as `i` < 10. The appropriate command is therefore:

```
while i < 10
```

The expression `i < 10` is an example of a **Boolean expression**. Such an expression takes on only two possible values, 0 and 1. In ***Ects***, all expressions that are interpreted as **relationships** either of equality or of inequality, are considered Boolean expressions. Thus, the following are Boolean expressions:

```
a < b
a = b
a > b
```

As usual, the variables `a` and `b` can be scalars, vectors or matrices. The Boolean expressions can be subject to the commands `gen`, `set`, and `mat`. If, for example, we enter

```
gen x = a > b
```

the component i of the vector `x` (the number of its components is determined by the current state of `smplstart` and `smplend`) equals 1 if $\mathtt{a}_i > \mathtt{b}_i$, and 0 otherwise.

The **precedence** of the relations `=`, `>`, and `<` is lower than that of all other arithmetic operations. This means that if we try to evaluate

```
gen z = x1*(x2 + x3) = x1*x2 + x2*x3
```

all the components of the vector `z` will be equal to 1. All the operations of addition and multiplication will be carried out before the comparison of the two numbers $\mathtt{x1} * (\mathtt{x2} + \mathtt{x3})$ and $\mathtt{x1} * \mathtt{x2} + \mathtt{x2} * \mathtt{x3}$. The two sides being equal, component by component, every component of `z` receives the value TRUE, or 1.

EXERCISES:

Generate a variable like the `z` of the above example, but enter `<` or `>` instead of `=`. How can we explain the results?

The evaluation of an expression that follows a command `while` is done as if we were under the influence of a `mat`.

* * * *

This choice was made to avoid long computations that would occur if the sample size is too large. Indeed, the relationship `i < 10` would be evaluated `smplend` times if the computation were done as they are done under `gen`.

* * * *

Then, to the element $(1,1)$ of the evaluated matrix, ***Ects*** adds 10^{-10}, to avoid the potential difficulties linked to rounding errors. Finally, if the biggest integer that is smaller than the value thereby obtained is zero, the condition is considered as being false, and the loop stops; otherwise, it is considered as true, and the loop continues.

An excellent example of a loop that has nothing to do with Monte Carlo experiments is provided by the estimation procedure contained in the file `logit.ect`. We noted in section 5.3 that this estimation makes use of an artificial regression. The estimates are obtained by the iteration of this regression, until the estimated parameters do not change from one iteration to the next. Here is the loop in question:

```
set tol = 1
set i = 0
while tol > 0.000001
        gen den = 1/sqrt(F*(1-F))
        gen r = (y - F)*den
        gen r0 = f*den
        gen r1 = r0*x1
        gen r2 = r0*x2
        gen r3 = r0*x3
        ols r r0 r1 r2 r3
        set a = a + coef(1)
        set b1 = b1 + coef(2)
        set b2 = b2 + coef(3)
        set b3 = b3 + coef(4)
        mat tol = coef'*coef
        set i = i+1
end
```

Before launching the loop, we initialize the scalars `i` and `tol`.[12] The sole purpose of `i` is to keep track of the number of iterations. The role of `tol` is to determine the moment when the algorithm converges. Indeed, we observe that the iterations continue as long as `tol` is greater than 0.000001. To interpret this variable, we need to know that the estimated parameters of the regression carried out by `ols` in the middle of the loop are the *corrections* that are added

[12] `tol` in lowercase letters to avoid possible confusion with the variable `TOL` which determines the precision level of the estimations.

to the existing values of the parameters. Thus, `tol` measures the square of the displacement, in the parameter space, caused by the last iteration. When this displacement is sufficiently small, the algorithm has converged.

2. Conditional Execution of a Command Block

For several reasons, it is often desirable to execute a block of commands only if a specific condition is met. In the example of the artificial regression that we just carried out, it is not excluded *a priori* that the algorithm does not converge, even after a considerable number of iterations. In this case, the value of `tol` will always be greater than 0.000001 when the counter `i` reaches a pre-specified value. If, but only if, this is the case, we would like the screen to display a suitable message. The command we use is `if`.

Let's consider the consequences of inserting the following commands at the end of the loop below.

```
if i = 20
   message
   20 iterations without convergence.  Should I continue
(1=Yes/0=No)?
   end
   input a
   if a = 0
       tol = 0
   end
end
```

The command `message` is identical to the command `text`, except the text is not printed in the output file: rather, it is printed on the screen, even if we had previously entered a `noecho` command. The command `input` permits us to stop the program in order to type a number, the value of which will be assigned to the scalar variable given as an argument to the command.

* * * *

The effect of the command

```
input a
```

is quite similar to the one of the commands

```
set t1 = smplstart
set t2 = smplend
sample 1 1
read con a
sample t1 t2
```

It appeared useful to have only one command to carry out such a small task!

* * * *

We can now describe what would happen if the algorithm has not yet converged after 20 iterations. ***Ects*** reads the command

```
if i = 20
```

and notices that the condition is satisfied. Then, the block of commands between `if` and `end` are executed. First, the `message`, ended by the first `end`, is displayed, and the machine waits for an answer from the user, who now has to type in a number. This number is assigned to the variable `a`. The following command is once again an `if` command. The variable `a` is examined, and if it equals zero, the block between the new `if` and the second `end` is executed. This block contains a single command, which is used only to set the variable `tol` to zero. And if `tol` is zero, the external loop, the one that is ended by the last `end`, stops, for the condition `tol` > 0.000001 is no longer satisfied. If the user's entry is different from 0, the loop continues until the algorithm converges.

EXERCISES:

Write an ***Ects*** program that invites the user to enter a number, and that then displays the entered number on the screen. Insert this program in a loop, which goes on indefinitely, or until the entered number equals zero.

We saw in the above example that there are several commands that await an `end` to signal that a block of commands, or of partial derivatives, or of text, is done. Here is the full list of these commands:

```
nls
ml
gmm
text
message
while
if
else
```

The first five commands of this list do not allow for the possibility of another command of this list intervening between the line of the command itself and its own `end`. But the last three commands can be enclosed one within another: As we have seen in the example, a block starting with an `if` can perfectly well be found inside a block starting with a `while`. For this to be possible, ***Ects*** uses a stack, like the stack that we have encountered in section 5.2. Thus, every `end` is correctly associated to the command of which it is the ending.

We also need to consider the command `else`. This command is used within an `if`. If the condition of the `if` is met, the commands of the block are executed. If among these commands ***Ects*** encounters an `else`, it jumps to the `end` of the `if` statement. However, if the condition of an `if` is not met, ***Ects*** starts looking, either for the `end` of that `if`, or for an `else`. If it is an `end` that is

found, ***Ects*** executes the block of commands between the `else` that has been encountered and the `end` of the `if` statement.
In other words, the structure

```
if ⟨condition⟩
     ⟨block1⟩
else
     ⟨block2⟩
end
```

executes ⟨*block*$_1$⟩ if the ⟨*condition*⟩ is met, and ⟨*block*$_2$⟩ otherwise. The two ⟨*blocks*⟩ can also contain other blocks, associated either to a `while` or to an `if`, which in turn can also contain blocks, and so on with no limit other than the memory of your computer.

A last remark on the use of the command `end`. We have seen in section 5.3 that, to signal the end of a `text` or of a `message`, the command `end` must be located in the beginning of the line, without a blank space. This is applicable only to the two commands `text` and `message`, which are in charge of accurately transferring the contents from the command file to the output file or to the screen. For the other commands that use `end`, we can equally well enter blank spaces at the beginning of the line so as to indicate the structure of the program, as was done in the examples of this section. You are even encouraged to do so.

3. (Pseudo-)Random Numbers

The bracket in the heading of the section states clearly that a computer is a deterministic machine. If this was not the case, programming would be considerably different and definitely less useful. Nonetheless, we need randomness if we are to implement simulations.

* * * *

And also for games, but, as I have already mentioned, games are stupid.

* * * *

The best we can do if we dispose of a deterministic machine is to use it to compute **pseudo-random numbers**, that is, numbers that share with truly random numbers all their essential properties.

The random number generator[13] that ***Ects*** uses is of the type referred to as **congruential**. Even if this type of generator is not necessarily the best one, it is the most frequently used. We can find a brief overview in section 21.2 of DM, and a more detailed one in Press Chapter 7 *et al.* (1986). The

[13] It is no longer necessary to distinguish between random and pseudo-random numbers. It is understood that the former cannot be generated by a machine.

generator of ***Ects*** is a double generator, that is, two independent congruences are used. This enables us to generate very long series of random numbers before returning to the starting point. For certain Monte Carlo experiments this aspect is important.

The function which is used to generate random numbers is `random`. According to programming jargon this function is **overloaded**. By this, we mean that the same term, `random`, is used to define several slightly different functions. The user indeed has two options. First, using the function with no argument:

```
gen u = random()
```

The elements of the vector `u` are random numbers, independent of each other, following the standard normal distribution.

* * * *

The Box-Muller algorithm is used for this purpose

* * * *

If, instead of `gen`, we were to use `set`, a single random number would be generated. If we were to use `mat`, the result would be identical to that of `gen`.

The other option is to employ two arguments. The only difference relative to the option of using the function without arguments is that the generated numbers follow the uniform distribution $\mathrm{U}(a, b)$, where the scalars a and b are respectively the first and second arguments provided to `random`. As usual, if the arguments are vectors or matrices, the element $(1, 1)$ is selected.

4. Monte Carlo

Now all the ingredients are in place. We can try a small experiment. What type of experiment? ... A Monte Carlo one of course! The experiment we will try is programmed in the file `nearunit.ect`. It is not essential to understand all the theoretical aspects of the experiment, even though an outline of the theory is given below.

First, the command file:

```
set NOBS = 100
set NREPS = 1000

silent
noecho

sample 1 NREPS
gen zero = 0
gen tau = colcat(zero,zero,zero,zero,zero,zero)
mat z = tau

sample 1 NOBS
set rho = 0.9
gen t = time(-1)
```

```
set j = 0
while rho < 1.01
     set j = j+1
     gen rhot = rho^t
     set i = 0
     while i < NREPS
         set i = i+1
         gen eps = random()
         gen y = conv(rhot, eps)
         gen ylag = lag(1,y)
         gen dely = y - ylag
         sample 2 NOBS
         ols dely ylag
         sample 1 NOBS
         set tau(i,j) = student
         set z(i,j) = NOBS*coef
     end
     message End of the experiment
end
     show rho
     set rho = rho + 0.02
end
message End of the experiment
end
sample 1 NREPS
echo
restore
ols tau c
ols z c
quit
```

The first two commands define two variables, `NOBS`, the size of the sample, and `NREPS`, the number of repetitions, or simulations, to carry out. It is always desirable to define such variables rather than to use precise numbers, in order to be able easily to reprogram the number of simulations carried out by the experiment by merely changing a few definitions at the beginning of the file. Also, a Monte Carlo experiment may be time consuming, especially if the process is being recorded on the screen or in an output file. Therefore, we enter `silent` and `noecho`. We then generate two `NREPS` $\times$ 6 matrices, `z` and `tau`, to contain the results of the experiment.

Then comes the initialization of the variables that will be used during the experiment. `rho` is a parameter, more precisely an autocorrelation parameter; `j` is a counter, which keeps track of the number of iterations of the outside loop; and `t` is a lagged time variable, the first component of which is equal to 0.

The variable controlling the external loop is `rho`. From its initial value of 0.9 it is incremented by 0.02 up to a final value of 1.0. After the last iteration, the value of `rho` will be 1.02, larger than 1.01, and the loop stops. We could have used the counter `j` in a similar way to control the loop. Within this loop is embedded another loop, controlled by the variable `i`, which is used to do the `NREPS` simulations for every one of the six possible values of `rho`, from which follows the necessity of having on hand `NREPS` $\times$ 6 matrices ready to contain all the results.

For each simulation, a random vector `eps` is generated, which is then subject to the command

```
gen y = conv(rhot, eps)
```

The vector `rhot` is defined only once for each value of `rho`: The component t of the vector equals ρ^{t-1}. The name of the new function `conv` comes from the technical term **convolution**. Let $\boldsymbol{x}$ and $\boldsymbol{w}$ denote two $n \times 1$ vectors. Then the convolution $\boldsymbol{z}$ of $\boldsymbol{x}$ and $\boldsymbol{w}$ is defined below by the formula:

$$z_t = \sum_{s=1}^{t} x_s w_{t-s+1}. \tag{16}$$

It is easy to see that the definition is symmetric with respect to $\boldsymbol{x}$ and $\boldsymbol{w}$. According to this definition, the variable `y` defined using the function `conv` is such that

$$y_t = \sum_{s=1}^{t} \rho^{s-1} \varepsilon_{t-s+1} = \sum_{s=1}^{t} \rho^{t-s} \varepsilon_s.$$

The interpretation of this definition comes from the fact that we wish the vector with typical element y_t to satisfy the recursive relation

$$y_t = \rho y_{t-1} + \varepsilon_t.$$

This equation is a first order autoregression, with autoregressive parameter ρ.

The series `y` obtained is then lagged and differenced. The sample size is adjusted so as to eliminate the first observation, and the first differences `dely` are regressed on the lagged variables `ylag`. The estimated parameter, multiplied by the sample size, and the t statistic, are then stored in the big matrices `z` and `tau`.

These are the two statistics used to detect the presence of a **unit root**. The theory of **Dickey-Fuller tests**, which use these statistics, can be found in chapter 20 of DM. The notations z and τ are often encountered in this theory: Refer to section 20.2 of DM.

After all the iterations of the external loop, `NREPS` simulations have been carried out. In addition to the generation of the variables, each simulation

requires an OLS estimate. In the current case, every big iteration includes a thousand regressions. On a 486/50MHz,[14] the computation time for these thousand regressions is approximately 50 seconds. After this, a message will appear on the screen, displaying the current state of `rho`. After the six iterations of the big loop, a concluding message is displayed. Then follows the analysis of the results. The two commands

```
ols tau c
ols z c
```

can be used to run 12 regressions, because the two variables `tau` and `z` have 6 columns each. The 12 columns are regressed on the constant, which is equivalent to computing the mean of each column as well as its standard deviation.

Then what? The theory of unit root tests is fairly complex, and few results are totally correct. It is known that the probability distributions followed by the statistics z and τ, even asymptotically, are not the standard distributions that we normally find in econometrics. The experiment we just carried out gave us estimates of the first moments of these probability distributions as functions of the autocorrelation parameter of the variable $\boldsymbol{y}$. Furthermore, the estimated standard deviations provided us with a measure of the accuracy of these estimates. Let's consider as an illustration, the case for which $\rho = 1$, which corresponds to the last big iteration, where the null hypothesis is true: There indeed is a unit root. We could have hoped for the expected values of the statistics to be zero in this case. Consider the results of the experiments on the statistic z, for $\rho = 1$. We obtain

```
Variable      Parameter estimate      Standard error   T statistic
c             -1.7138185              0.0954023       -17.9641271
```

The hypothesis that the expected value of z is zero is unambiguously rejected. The point estimate of this expected value is -1.71.[15] and its standard error is 0.095. A reasonable confidence interval would therefore be $[-1.90, -1.52]$. It is easy to obtain a narrower confidence interval, either by simply increasing the number of simulations, or by use of a technique of variance reduction; refer to sections 21.5 and 21.6 of DM.

* * * *

The complete experiment lasted less than 5 minutes on a 486/50. If we are willing to let the machine run a whole night, we can obtain a much

[14] The 80486, to give it its full name, was the chip that preceded the first Pentiums produced by Intel. At the time, it was a super fast chip; now it may be used in nursery school, but not elsewhere.

[15] It is useless to keep the number as printed out: the estimation error is too great.

more precise result. (These days, 5 minutes would be enough for much more accurate results.)

* * * *

EXERCISES:

Limit yourself to the case where $\rho = 1$, and obtain a more precise estimate for the expected value of z using a control variate.

Chapter 7
Everything else

1. Other Functions, Other Rules

It is impossible to consider *all* the functions that we can use in ***Ects*** in the econometric contexts considered in this manual. However, the functions that we have not yet seen can be quite useful in other contexts as well. The purpose of this chapter is to describe these functions, and to specify some general rules followed by ***Ects*** in the evaluation of functions.

The first function we consider is `round`, which is used to round real numbers. The result of `round(x)` for a *scalar* `x` is the integer closest to `x`. The result of `round(y)` for a *column vector* `y` is a column vector that has the same dimension as `y`. The function is applied element by element. The result of `round(A)` for any *matrix* `A` is a column vector that has the same number of rows as `A`, and which results from the application of the function to the *first column* of `A`, *if* the calculation is done under `gen`, but if the calculation is done under `mat`, the result is a matrix of the same dimensions as `A`, resulting from the application of the function element by element.

The rules mentioned above generally apply to all ***Ects*** functions, save for a few exceptions.

* * * *

Recall: `set` is equivalent to `gen` preceded by

```
sample 1 1
```

and followed by the restoration of the old values of `smplstart` and `smplend`.

* * * *

To avoid future difficulties, here is a list of exceptions which are considered later on.

```
colcat
rowcat
random
uptriang
conv
```

```
diag
det
sum
time
seasonal
lag
sort
```

Here are two functions that obey the rules, but which, unlike `round`, take two arguments. `max(a,b)` gives the maximum (algebraic, and taking into account the sign) of the `a` and `b` variables. Similarly, `min(a,b)` gives the algebraic minimum. We can now state some rules that apply to functions that, like `max` and `min`, take more than one argument.[16] Under `gen`, only the first columns of *each and every argument* are taken into account. Under `mat`, any function to which more than one argument is given, gives a syntax error. Now here is a rule that even applies to the exceptional functions: if the number of arguments is smaller than what the function expects, there is a syntax error. If the number of arguments is greater than what the function expects, the arguments that follow are ignored.

There exists a group of (unexceptional) functions that allow us to calculate the critical values associated with the most common probability distributions in econometrics. These are the standard normal, the χ^2, t, and the F distributions. The syntax

```
normcrit(x)
```

calculates the real number z which satisfies the following equation:

$$\Phi(z) = x,$$

where Φ is the standard normal cumulative distribution function, given in equation (4). Therefore, to obtain the critical value of a two-tailed test with a level of 5%, we use `normcrit(0.975)`

EXERCISES:

Why?

It is clear that the argument x of `normcrit` can be interpreted as a probability, and therefore is an element of the $[0, 1]$ interval. Otherwise, the value given by `normcrit` is an approximation to either $-\infty$ or ∞.

The function Φ itself, available under ***Ects*** through the function `Phi`, enables us to compute the **marginal significance level** of a test statistic. It is often referred to as the ***P* value** of the test statistic. This concept is defined in section 3.4 of DM. For example, if we have obtained a statistic of which the numerical value is 2.2, and if, under the null hypothesis, this statistic

[16] Always except for the exceptions in the list.

is a drawing from the N(0, 1) distribution, the P value associated with it is computed as `1 - Phi(2.2)`. It represents the probability mass in the tail of the N(0, 1) distribution beyond the value 2.2.

EXERCISES:

Does the P value computed in such a way correspond to a one-tailed or two-tailed test?

The functions corresponding to `Phi` and `normcrit` for a t distribution are `tstudent` and `studcrit`. The syntax is as follows: `tstudent(x,n)` is the probability mass to the left of the real number `x` for a t statistic with `n` degrees of freedom. `studcrit(x,n)` is the real number z that satisfies the following equation:

$$T_n(z) = x,$$

where T_n denotes the probability distribution function of a t statistic with n degrees of freedom. If x lies outside the $[0, 1]$ interval, this function will give a value close to infinity, just like `normcrit`.

* * * *

Watch out for the name of the function `tstudent`. The initial `t` is necessary to distinguish the *function* from the *variable* `student` which contains the t statistics of a regression.

* * * *

The χ^2 distribution is given by the functions `chisq` and `chicrit`. Just as in the case of the t distribution, two arguments are necessary, the first one being a real number and the second being the degrees of freedom.

* * * *

It is not necessary for the number of degrees of freedom to be an integer. In fact, the PDF associated with the χ^2 distribution exists for any positive real value of the degrees of freedom. Since the invention of fractals we know that that a dimension is not necessarily an integer. However, if the number of degrees of freedom is negative, the function `chisq` gives a value of 0.

* * * *

For the F distribution, the functions are `fisher` and `fishcrit`. This time, three arguments are required, the last two being the degrees of freedom. The first deals with the numerator, the second with the denominator.

Three other functions are available. They are used by ***Ects*** to calculate the cumulative distribution functions of the statistics we have just considered, and, because life is unpredictable, they are put at the disposal of the general public. The `gln` function corresponds to the logarithm of the **gamma function**. The definition of the gamma function itself is

$$\Gamma(z) = \int_0^\infty t^{z-1} e^{-t}\, dt.$$

If z is an integer, the function is linked to the factorial by the formula

$$z! = \Gamma(z+1).$$

It is because the factorial is a function of which the values quickly become huge that we prefer to use its log. The syntax is quite simple: `gln(x)` equals $\log \Gamma(x)$.

Another function that is required to calculate the CDF is the **incomplete gamma function**. It is defined as follows:

$$P(a,x) = \frac{\gamma(a,x)}{\Gamma(a)} = \frac{1}{\Gamma(a)} \int_0^x e^{-t} t^{a-1}\, dt \qquad (a > 0).$$

Both of the functions $P(a,x)$ and $\gamma(a,x)$ are referred to as incomplete gamma functions. It is the first one, $P(a,x)$, that is available through the expression `gamma(a,x)`. Both of the arguments have to be positive; or else a zero value is obtained.

Finally, the **incomplete beta function** is defined by the formula

$$I_x(a,b) = \frac{1}{B(a,b)} \int_0^x t^{a-1}(1-t)^{b-1}\, dt,$$

where the **beta function** is defined in terms of the gamma function by the following relationship:

$$B(a,b) = \frac{\Gamma(a)\Gamma(b)}{\Gamma(a+b)}.$$

The $I_x(a,b)$ function is evaluated by the ***Ects*** expression `betafn(a,b,x)`. Watch out: the argument x must be an element of the interval $[0,1]$, or else the value of the function will be zero.

For more detailed information concerning the functions described above and on other useful applied mathematics functions, see chapter 6 and the work of Press*et al* (1986).

2. Arithmetic Operations: Useful Rules

Considering all the possibilities offered by ***Ects*** when it comes to matrix and vector calculations, but also to the manipulation of economic series, there are necessarily more or less subtle rules that govern the interpretation of the expressions read by the software. We have seen on several occasions that these expressions have different meanings in the `gen` and `mat` commands. It is, however, not necessary to consider `set` separately, because its operation is it always equivalent to the following commands:

```
set t1 = smplstart
set t2 = smplend
sample 1 1
gen ...
sample t1 t2
```

where the `gen` replaces the `set`, with a tiny difference: the `set` command is the only one that can be used to assign scalar values to the *elements* of vectors and matrices. Thus, `set A(3,4) =` ⟨*expression*⟩ is correct, but if we were to use `gen` or `mat` instead of the `set`, a syntax error would occur.

Regarding the ***Ects*** *functions*, the relevant rules have been set out in the previous section. We will now consider the rules that apply to the elementary operations of addition, subtraction, multiplication, and division, respectively noted `+`, `-`, `*`, and `/`. The exponential operation, denoted by `^`, was covered in section 3.1.

The elementary operations are all **binary operations**. In other words, they have two arguments. Both of these arguments are matrices; the first one precedes and the second follows the `+`, `-`, `*`, or `/` symbol of the operation. At this stage, it is no longer useful to distinguish between matrices that are scalars, vectors, or matrices with more than one row and more than one column.

The operations are carried out according to the **precedence** rules that we laid out in section 2.1. The inverse and transpose matrix operations are carried out before the arithmetic operations, and similarly for the exponential operations. Then the multiplications and divisions are executed followed by the additions and subtractions. If there are any Boolean operations, symbolised by `=`, `>`, or `<`, they are carried out at the end. At each stage, two matrices are involved, one from each side of the symbol of the operation to be carried out.

Even before the operations are started, the matrices that are the object of these operations must be constructed by ***Ects***. The construction is effected either from an explicit numerical value, or from a symbol that ***Ects*** can recognize. Let us first consider numerical values. Suppose ***Ects*** reads the number 4. What will be the constructed matrix that represents this number? In a `mat` command, the answer is simple: it will be a 1×1 matrix, of which the single element equals 4. In a `gen` command, a column vector will be generated with `smplend` components, each equal to 4, regardless of the value of `smplstart`.

There are two types of symbols that can intervene in these expressions, in addition to the names of the ***Ects*** functions: previously defined matrices, and macros. If we're dealing with a matrix under `mat`, the existing matrix is reproduced as is for the purposes of arithmetic operations. Under `gen`, if the existing matrix is a 1×1 matrix, a column vector with `smplend` components is generated, and each component is assigned the value of the single matrix element. But if the existing matrix is not just a scalar, a matrix is generated with the same number of columns, and `smplend` rows. At the beginning, all the elements of the new matrix are 0. Then all the elements of the old matrix that do not overflow the dimensions of the new matrix are transferred to it.

If ***Ects*** encounters a symbol that represents a macro, the corresponding text is found, and the matrix is calculated according to the rules of the command being used. This computation is carried out separately, as if the text of the macro was in parentheses.

At this stage we can reasonably ask why the value of `smplstart` is not taken into consideration. The answer is that it is easier to account for it at the last minute, that is, at the moment when the whole expression that is subject to the command is evaluated. After this evaluation, the resulting matrix is used either to generate or to update the variable that is the left-hand side of the command. Thus, if we enter

```
gen z = x + y
```

the result of evaluating the expression `x + y` is either a vector or a matrix with `smplend` rows. Let $\boldsymbol{Z}$ denote this `smplend` $\times k$ matrix.

If the variable `z` does not exist, a `smplend` $\times k$ matrix is generated with its typical element equal to zero. But if there already exists in the memory of the computer a variable `z`, two options are considered. If the number of rows of `z` is less than `smplend`, or if the number of columns is less than k, the existing variable will be deleted and a zero `smplend` $\times k$ matrix will be generated. Otherwise, `z` keeps its old dimensions and its old elements. Finally, the block of the new matrix `z` running from the row with subscript `smplstart` to the row with subscript `smplend`, and from the first column to the column k, is replaced by the corresponding block of $\boldsymbol{Z}$.

Under `mat`, things are much simpler. Neither `smplstart` nor `smplend` is taken into account, and the matrix which is the result of the evaluated expression is not modified at all.

Now we have seen how ***Ects*** constructs the matrices that will then be subject to arithmetic operations. Let `A` denote an $n \times m$ matrix, and `B` denote a $p \times q$ matrix. If ***Ects*** then has to evaluate `A + B` in a `mat` command, the result is an $n \times m$ matrix. The same holds for `A - B`. If some of the elements of `B` are missing, they are replaced by zeros; if there are too many, they are ignored. In a `gen` command, the result is a `smplend` $\times m$ matrix. Once again, all the missing elements are replaced by zeros, and all the excess elements are dropped. But here is a great difference relative to `mat`. Under `gen`, *only the first column* of `B` is taken into consideration. If we denote this first column by $\boldsymbol{b}_1$, and the columns of `A`, $i = 1, \ldots, m$, by $\boldsymbol{a}_i$, then `A + B` is a matrix of which the successive columns are $\boldsymbol{a}_i + \boldsymbol{b}_1$, $i = 1, \ldots, m$.

Keeping the same notations, let's first take a closer look at `A*B` under `mat`. In matrix calculations, there are two types of multiplication operations, regular matrix multiplication, and multiplication by a scalar, where one of the two "matrices" is simply a scalar, and the "product" is a matrix of which the elements are the ordinary products of the scalar with the elements of the non-scalar matrix. To deal with this double usage of the multiplication operation,

Ects first examines the two matrices `A` and `B` to determine if one of them is a scalar. If `A` is a scalar, the matrix `A*B` is a $p \times q$ with the dimensions of `B`. If `B` is a scalar, the matrix `A*B` is $n \times m$, like the matrix `A`. Similarly, if, in the expression `A/B`, `B` is a scalar, the result has the dimensions of `A`. However, if `A` is a scalar, and `B` is not, a syntax error will occur.

If neither `A` nor `B` is a scalar, the evaluation of `A*B` is an $n \times q$ matrix, computed following the rules of matrix multiplication. As in the case of addition, every missing element is replaced by zero, and all the extra elements are ignored.

The expression `A*B` in a `gen` command is interpreted quite differently. It is no longer considered as a matrix multiplication, but rather as a multiplication, element by element, of two series. In fact, the operation is perfectly analogous to an addition operation. The result has `smplend` rows and m columns, the successive columns being the result of the multiplication, element by element, of the corresponding column of `A` with the *first* column of `B`. The columns of `B` following the first one are once again ignored. Under `gen`, division obeys the same rules as multiplication.

Finally, the Boolean operations are to be considered. The rules are the same for all these operations; we therefore need only treat the case of `<`. The evaluation of the expression `A < B` under `mat` always results in a scalar. The *only* elements taken into consideration are the $(1, 1)$ elements of the two matrices. Thus, if $a_{11} < b_{11}$, the value assigned to `A < B` is 1, and is 0 otherwise. Under `gen`, the result is a `smplend` $\times m$ matrix, of which the elements are computed using the successive columns of `A` and the first column of `B`, exactly as for the other arithmetic operations.

3. Exceptional Functions

In the first section of this chapter, we established the list of ***Ects*** functions of which the behaviour relative to their arguments does not follow the general rules stated above. We now consider these exceptional functions, except `random`, which we have already covered in section 6.3.

The function `colcat` generally takes several arguments. (Refer to section 3.1.) Let us consider the expression

```
colcat(A1,...,Ak)
```

where the matrix `Ai`, for $i = 1, \ldots, k$, has dimensions $n_i \times m_i$. The result is a matrix of which the number of rows equals $\max_i n_i$ under `mat` and $\min(\texttt{smplend}, \max_i n_i)$ under `gen`; and the number of columns is equal to $\sum_{i=1}^{k} m_i$ for both commands. Any missing element is replaced by zero, and in the case of `gen` the rows following `smplend` are lost. The syntax of `rowcat` is similar. The evaluation of

```
rowcat(A1,...,Ak)
```

is a matrix of which the number of rows is equal to $\sum_{i=1}^{k} n_i$ and the number of columns is equal to $\max_i m_i$. The effect of the function does not depend on the choice of the `mat` or `gen` command.

The usage of the function `sum` under `mat` results in a syntax error. Under `gen`, it can take many arguments. The result of

```
sum(A1,...,Ak)
```

is a matrix of which the dimensions are identical to the matrix resulting from

```
colcat(A1,...,Ak)
```

Exceptionally, the value of `smplstart` is taken into consideration by `sum`. The rows of the result between the first and that with subscript `smplstart` -1 are zero. The following rows result from the addition of the previous rows starting at the subscript `smplstart`. Like `sum`, the function `time` will only start acting from row `smplstart`. This function only accepts one argument. In the result, every element of the row t is incremented by t.

The functioning of `seasonal` was covered in section 2.2. This function, which takes two arguments, does not distinguish between `gen` and `mat`. The function `lag` was covered in section 2.1. This function will result in a syntax error if it is used under `mat`. The result of the evaluation of `lag(k,A)`, where `A` is an $n \times m$ matrix, is a matrix with dimensions `smplend` $\times m$, of which the rows from the first one until the one with subscript `smplstart` -1 are zero, as well as those that would correspond to a non-existing row of `A`.

The functions `det`, `diag`, and `uptriang` were set out in 3.3. Their functioning is similar under `gen` and `mat`; they each require a single argument.

The convolution operation was defined in section 6.4, where the function `conv` was introduced. When the expression `conv(A,B)` is evaluated, for an $n \times m$ matrix `A`, the result is a `smplend` $\times m$ matrix under `gen`, and an $n \times m$ matrix under `mat`. The only column of `B` to be used is the first column. In the definition (16) of the convolution, the first observation is in fact the observation `smplstart` under `gen`.

The last function we will look at is `sort`, which has not been introduced so far. This function is used to **sort** the elements of a vector or a matrix. It only takes a single argument, the matrix to be sorted. The sorting is carried out by rows, in ascending order of the elements of the first column. Thus, if `A` is an $n \times m$ matrix, the result of `sort(A)` is an $n \times m$ matrix under `mat`, and a `smplend` $\times m$ matrix under `gen`. In the second case, the first `smplstart` -1 rows are zero, and the sorting is limited to the rows starting at `smplstart`.

EXERCISES:

How do we sort the elements of a vector in descending order?

4. Unix and the Source Code

Ects can be run not only on PCs, but also on all other workstations and other computers on which Unix is installed. (Linux is a version of Unix for this purpose.) As far as I know, it can also be run under other operating systems: Once again, the necessary and sufficient condition is that a program written in C++ can be correctly compiled.

The traditional method for distributing software intended for Unix use is to place at the disposal of the potential users the **source code** of the software, along with a **Makefile**. The source code is a program, which may be quite long, that uses a programming language. The C language and the Unix operating system are twins: they were developed in tandem, and, since then, they have been used in close partnership. Only a program written in C, as was the case for the first version of ***Ects***, can be guaranteed to compile correctly on all Unix based machines. But C++ is a rising star whose gravitational attraction was too strong for me. Most modern workstations can access a C++ compiler for the mere reason that such a compiler is freely available at the Massachusetts Free Software Foundation.

The ***Ects*** source code is available on request. When the next version (version 4) finally becomes available, the source code will be more readily available on the web. For a long time, compiler development lagged behind the development of the C++ language, and it was next door to impossible to maintain genuinely portable code. This problem has now been overcome, and the main reason for which I do not make the code too readily available is that it is a mess! It is full of conditional statements intended to handle the quirks of various compilers and their successive versions. Explaining it takes days. It seemed better to adopt the radical solution of recoding version 4 from scratch, with readable, portable code.

Problems with the use of ***Ects*** should be communicated to me by email, at the address

`russell.davidson@mcgill.ca`

All linguistic mistakes in this manual should be blamed on Amjad, Martin, Ram and Serge, whose email addresses will remain confidential.[17] I wish you a pleasurable experience with the software. Updates, both of the executable files and the documentation, are made available as they appear on the websites listed at the beginning of Chapter 1, repeated here for convenience:

`http://russell.vcharite.univ-mrs.fr/ects3`

`http://russell-davidson.arts.mcgill.ca/ects3`

[17] Or so they think! The Internet lets you discover all kinds of things that some people would rather keep secret.

References

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

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

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

Press, W. H., B. P. Flannery, S. A. Teukolsky, et W. T. Vetterling (1986). *Numerical Recipes*, Cambridge University Press, Cambridge.[18]

[18] A later edition of this book appeared in 1992, in several versions, for several different programming languages. It was the algorithms of the first edition that were the inspiration of those used in the current version of ***Ects***, but interested readers are strongly recommended to make use of the second edition.

General Index

2sls.dat, 4, 20
2sls.ect, 4, 20

abs, 16
ar.dat, 4
ar.ect, 4, 17, 37
archdemo.ect, 4
argen.ect, 4
arnls.ect, 4, 37

batch, 50
beep, 40, 51–52
betafn, 71

c, 6, 23
CG, 42
chicrit, 70
chisq, 70
coef, 8, 21, 30, 31, 38, 42, 46
colcat, 23–25, 29, 54, 74
conv, 65, 75
cos, 16
crit, 46

def, 42–43
del, 43
deriv, 37, 38
det, 32, 75
diag, 32, 75

echo, 49
else, 61–62
end, 37, 39, 45, 54, 55, 57–62
errvar, 8, 21, 31, 38
exp, 16

fishcrit, 70
fisher, 70
fit, 8, 21, 30, 38

gamma, 71
gen, 13–15, 17, 22, 24, 28, 29, 32, 37, 42, 43, 45, 58, 59, 63, 68, 69, 71–75
gen.ect, 4, 13–14
gln, 70
gmm, 44–46, 53, 56
gmm.ect, 4
gv.dat, 4
gv.ect, 4

hat, 8, 31, 32, 34

if, 60–62
input, 60–61
integral.ect, 4
interact, 49–51
inv, 24
invhess, 42, 46
invOPG, 42
iv, 19–21, 25, 30, 31, 38, 44, 45, 56
ivnls.dat, 4
ivnls.ect, 4, 44, 45

lag, 16–18, 75
lhat, 42
log, 16
logit.ect, 4, 53, 59
lt, 42

MacKinnon, James G, iii
mat, 22–30, 32, 37, 45, 58, 63, 68, 69, 71–75
max, 69
maxiter, 38
message, 60–62
min, 69
ml, 39–42, 45, 46, 53
ml.ect, 4, 39
mrs.ect, 4, 51, 52

nearunit.ect, 4, 63
newlogit.ect, 4
ninst, 21
niter, 38, 42, 46
nliv.ect, 4
nls, 35–40, 42, 44, 45
nls.dat, 4, 36
nls.ect, 4, 35, 37
nlsols.ect, 4, 37
nobs, 7, 21, 38, 42
noecho, 49, 51, 53, 60, 64
normcrit, 69, 70
nreg, 8, 21, 38, 42, 46

ols, 6–9, 18, 20, 21, 24, 25, 30, 37, 38, 40, 50, 56, 59
ols.dat, 4, 18, 34, 38
ols.ect, 4–7, 23, 47, 50

out, 48–49

pause, 52
Phi, 16, 69, 70
phi, 16
pitch.ect, 4, 51, 52
print, 11–12, 22, 50
proclogit.ect, 4
put, 11–12, 52–56

quit, 6, 47, 50–51

R2, 8, 31, 38
random, 63–64
read, 6, 9–10
rem, 53
res, 8, 21, 30, 38
restore, 49, 52
round, 68, 69
rowcat, 24, 29, 54, 74
run, 51

sample, 5–9, 11, 17, 22
season.ect, 4, 18
seasonal, 18, 75
set, 13–15, 17, 18, 22, 27–29, 32, 36–39, 58, 63, 68, 71, 72
show, 11–12, 22, 50
sign, 16
silent, 49–53, 55, 56, 64
sin, 16
smplend, 22, 24, 27, 30, 58, 59, 68, 72–75
smplstart, 22, 27, 30, 58, 68, 72, 73, 75
sort, 75
sqrt, 16
sse, 8, 20, 30, 38
ssr, 8, 14, 20, 30, 38
sst, 8, 20, 30, 38
stderr, 8, 21, 31, 38, 42
studcrit, 70
student, 8, 21, 31, 38, 42, 70
sum, 16, 75
sur.dat, 4, 29, 33
sur.ect, 4, 33
svdcmp, 34
SVDU, 34
SVDV, 34
SVDW, 34

tan, 16
testplot.ect, 4
text, 52–56, 60, 62
time, 16, 75
TOL, 18, 19, 37, 59
tstudent, 70

uptriang, 32, 75

var.ect, 4
vcov, 8, 21, 31, 38

while, 57–62
write, 11–12

XtPwXinv, 21, 31
XtXinv, 8, 21, 31, 38

Ects Index

Command Files
- 2sls.ect, 4, 20
- ar.ect, 4, 17, 37
- archdemo.ect, 4
- argen.ect, 4
- arnls.ect, 4, 37
- gen.ect, 4, 13–14
- gmm.ect, 4
- gv.ect, 4
- integral.ect, 4
- ivnls.ect, 4, 44, 45
- logit.ect, 4, 53, 59
- ml.ect, 4, 39
- mrs.ect, 4, 51, 52
- nearunit.ect, 4, 63
- newlogit.ect, 4
- nliv.ect, 4
- nls.ect, 4, 35, 37
- nlsols.ect, 4, 37
- ols.ect, 4–7, 23, 47, 50
- pitch.ect, 4, 51, 52
- proclogit.ect, 4
- season.ect, 4, 18
- sur.ect, 4, 33
- testplot.ect, 4
- var.ect, 4

Data Files
- 2sls.dat, 4, 20
- ar.dat, 4
- gv.dat, 4
- ivnls.dat, 4
- nls.dat, 4, 36
- ols.dat, 4, 18, 34, 38
- sur.dat, 4, 29, 33

Ects Commands
- batch, 50
- beep, 40, 51–52
- def, 42–43
- del, 43
- deriv, 37, 38
- echo, 49
- else, 61–62
- end, 37, 39, 45, 54, 55, 57–62
- gen, 13–15, 17, 22, 24, 28, 29, 32, 37, 42, 43, 45, 58, 59, 63, 68, 69, 71–75
- gmm, 44–46, 53, 56
- if, 60–62
- input, 60–61
- interact, 49–51
- iv, 19–21, 25, 30, 31, 38, 44, 45, 56
- mat, 22–30, 32, 37, 45, 58, 63, 68, 69, 71–75
- message, 60–62
- ml, 39–42, 45, 46, 53
- nls, 35–40, 42, 44, 45
- noecho, 49, 51, 53, 60, 64
- ols, 6–9, 18, 20, 21, 24, 25, 30, 37, 38, 40, 50, 56, 59
- out, 48–49
- pause, 52
- print, 11–12, 22, 50
- put, 11–12, 52–56
- quit, 6, 47, 50–51
- read, 6, 9–10
- rem, 53
- restore, 49, 52
- run, 51
- sample, 5–9, 11, 17, 22
- set, 13–15, 17, 18, 22, 27–29, 32, 36–39, 58, 63, 68, 71, 72
- show, 11–12, 22, 50
- silent, 49–53, 55, 56, 64
- svdcmp, 34
- text, 52–56, 60, 62
- while, 57–62
- write, 11–12

Ects Functions
- abs, 16
- betafn, 71
- chicrit, 70
- chisq, 70
- colcat, 23–25, 29, 54, 74
- conv, 65, 75
- cos, 16
- det, 32, 75
- diag, 32, 75
- exp, 16
- fishcrit, 70
- fisher, 70
- gamma, 71
- gln, 70
- inv, 24

lag, 16–18, 75
log, 16
max, 69
min, 69
normcrit, 69, 70
Phi, 16, 69, 70
phi, 16
random, 63–64
round, 68, 69
rowcat, 24, 29, 54, 74
seasonal, 18, 75
sign, 16
sin, 16
sort, 75
sqrt, 16
studcrit, 70
sum, 16, 75
tan, 16
time, 16, 75
tstudent, 70
uptriang, 32, 75

Ects Variables

c, 6, 23
CG, 42
coef, 8, 21, 30, 31, 38, 42, 46
crit, 46
errvar, 8, 21, 31, 38
fit, 8, 21, 30, 38
hat, 8, 31, 32, 34
invhess, 42, 46
invOPG, 42
lhat, 42
lt, 42
maxiter, 38
ml, 39
ninst, 21
niter, 38, 42, 46
nobs, 7, 21, 38, 42
nreg, 8, 21, 38, 42, 46
R2, 8, 31, 38
res, 8, 21, 30, 38
smplend, 22, 24, 27, 30, 58, 59, 68, 72–75
smplstart, 22, 27, 30, 58, 68, 72, 73, 75
sse, 8, 20, 30, 38
ssr, 8, 14, 20, 30, 38
sst, 8, 20, 30, 38
stderr, 8, 21, 31, 38, 42
student, 8, 21, 31, 38, 42, 70
SVDU, 34
SVDV, 34
SVDW, 34
TOL, 18, 19, 37, 59
vcov, 8, 21, 31, 38
XtPwXinv, 21, 31
XtXinv, 8, 21, 31, 38

Exceptional Functions

colcat, 68
conv, 68
det, 69
diag, 68
lag, 69
random, 68
rowcat, 68
seasonal, 69
sort, 69
sum, 69
time, 69
uptriang, 68