Gretl User's Guide

Gnu Regression, Econometrics and Time-series

Allin Cottrell
Department of Economics
Wake Forest university

Riccardo "Jack" Lucchetti
Dipartimento di Economia
Università Politecnica delle Marche

February, 2006

Contents

Chapter 1

Introduction

1.1 Features at a glance

Gretl is an econometrics package, including a shared library, a command-line client program and a graphical user interface.

User-friendly Gretl offers an intuitive user interface; it is very easy to get up and running with econometric analysis. Thanks to its association with the econometrics textbooks by Ramu Ramanathan, Jeffrey Wooldridge, and James Stock and Mark Watson the package offers many practice data files and command scripts. These are well annotated and accessible.

Flexible You can choose your preferred point on the spectrum from interactive point-and-click to batch processing, and can easily combine these approaches.

Cross-platform Gretl's "home" platform is Linux but it is also available for MS Windows and Mac OS X, and should work on any unix-like system that has the appropriate basic libraries (see Appendix B).

Open source The full source code for gretl is available to anyone who wants to critique it, patch it, or extend it.

Reasonably sophisticated Gretl offers a full range of least-squares based estimators, including two-stage least squares and nonlinear least squares. It also offers several specific maximum-likelihood estimators (e.g. logit, probit, tobit) and, as of version 1.5.0, general likelihood-maximization functionality. The program supports estimation of systems of simultaneous equations, GARCH, ARMA, vector autoregressions and vector error correction models.

Accurate Gretl has been thoroughly tested on the NIST reference datasets. See Appendix C.

Internet ready Gretl can access and fetch databases from a server at Wake Forest University. The MS Windows version comes with an updater program which will detect when a new version is available and offer the option of auto-updating.

International Gretl will produce its output in English, French, Italian, Spanish, Polish or German, depending on your computer's native language setting.

1.2 Acknowledgements

The gretl code base originally derived from the program ESL ("Econometrics Software Library"), written by Professor Ramu Ramanathan of the University of California, San Diego. We are much in debt to Professor Ramanathan for making this code available under the GNU General Public Licence and for helping to steer gretl's development.

We are also grateful to the authors of several econometrics textbooks for permission to package for gretl various datasets associated with their texts. This list currently includes William Greene, author of *Econometric Analysis*; Jeffrey Wooldridge (*Introductory Econometrics: A Modern Approach*); James Stock and Mark Watson (*Introduction to Econometrics*); Damodar Gujarati (*Basic Econometrics*); and Russell Davidson and James MacKinnon (*Econometric Theory and Methods*).

GARCH estimation in gretl is based on code deposited in the archive of the *Journal of Applied Econometrics* by Professors Fiorentini, Calzolari and Panattoni, and the code to generate *p*-values for Dickey–Fuller tests is due to James MacKinnon. In each case we are grateful to the authors for permission to use their work.

With regard to the internationalization of gretl, thanks go to Ignacio Díaz-Emparanza, Michel Robitaille, Cristian Rigamonti, Tadeusz and Pawel Kufel, and Markus Hahn, who prepared the Spanish, French, Italian, Polish and German translations respectively.

Gretl has benefitted greatly from the work of numerous developers of free, open-source software: for specifics please see Appendix B. Our thanks are due to Richard Stallman of the Free Software Foundation, for his support of free software in general and for agreeing to "adopt" gretl as a GNU program in particular.

Many users of gretl have submitted useful suggestions and bug reports. In this connection particular thanks are due to Ignacio Díaz-Emparanza, Tadeusz Kufel, Pawel Kufel, Alan Isaac, Cri Rigamonti and Dirk Eddelbuettel, who maintains the gretl package for Debian GNU/Linux.

1.3 Installing the programs

Linux

On the Linux[1] platform you have the choice of compiling the gretl code yourself or making use of a pre-built package. Ready-to-run packages are available in rpm format (suitable for Red Hat Linux and related systems) and also deb format (Debian GNU/Linux). If you prefer to compile your own (or are using a unix system for which pre-built packages are not available) here is what to do.

1. Download the latest gretl source package from gretl.sourceforge.net.
2. Unzip and untar the package. On a system with the GNU utilities available, the command would be `tar xvfz gretl-N.tar.gz` (replace `N` with the specific version number of the file you downloaded at step 1).
3. Change directory to the gretl source directory created at step 2 (e.g. `gretl-1.1.5`).
4. The basic routine is then

   ```
   ./configure
   make
   make check
   make install
   ```

   However, you should probably read the `INSTALL` file first, and/or do

   ```
   ./configure --help
   ```

   first to see what options are available. One option you way wish to tweak is `--prefix`. By default the installation goes under `/usr/local` but you can change this. For example

   ```
   ./configure --prefix=/usr
   ```

   will put everything under the `/usr` tree. In the event that a required library is not found on your system, so that the configure process fails, please see Appendix B.

Gretl offers support for the gnome desktop. To take advantage of this you should compile the program yourself (as described above). If you want to suppress the gnome-specific features you can pass the option `--without-gnome` to `configure`.

[1] In this manual we use "Linux" as shorthand to refer to the GNU/Linux operating system. What is said herein about Linux mostly applies to other unix-type systems too, though some local modifications may be needed.

MS Windows

The MS Windows version comes as a self-extracting executable. Installation is just a matter of downloading `gretl_install.exe` and running this program. You will be prompted for a location to install the package (the default is `c:\userdata\gretl`).

Updating

If your computer is connected to the Internet, then on start-up gretl can query its home website at Wake Forest University to see if any program updates are available; if so, a window will open up informing you of that fact. If you want to activate this feature, check the box marked "Tell me about gretl updates" under gretl's "File, Preferences, General" menu.

The MS Windows version of the program goes a step further: it tells you that you can update gretl automatically if you wish. To do this, follow the instructions in the popup window: close gretl then run the program titled "gretl updater" (you should find this along with the main gretl program item, under the Programs heading in the Windows Start menu). Once the updater has completed its work you may restart gretl.

Chapter 2

Getting started

2.1 Let's run a regression

This introduction is mostly angled towards the graphical client program; please see Chapter 18 below and the *Gretl Command Reference* for details on the command-line program, gretlcli.

You can supply the name of a data file to open as an argument to gretl, but for the moment let's not do that: just fire up the program.[1] You should see a main window (which will hold information on the data set but which is at first blank) and various menus, some of them disabled at first.

What can you do at this point? You can browse the supplied data files (or databases), open a data file, create a new data file, read the help items, or open a command script. For now let's browse the supplied data files. Under the File menu choose "Open data, sample file, Ramanathan. . . ". A second window should open, presenting a list of data files supplied with the package (see Figure 2.1). The numbering of the files corresponds to the chapter organization of Ramanathan (2002), which contains discussion of the analysis of these data. The data will be useful for practice purposes even without the text.

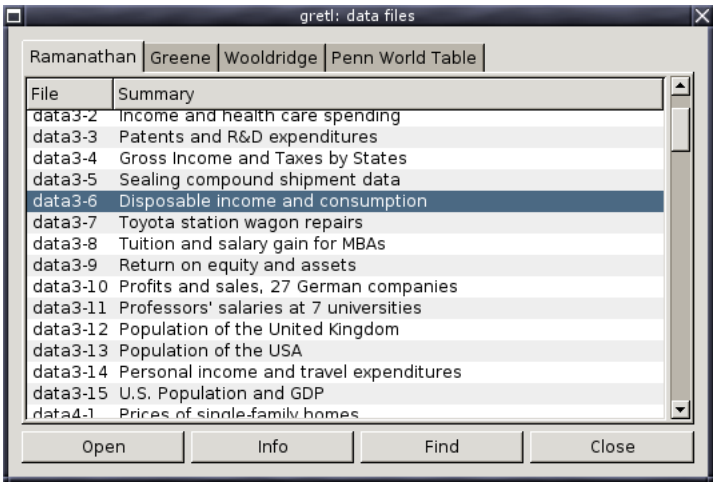


Figure 2.1: Practice data files window

If you select a row in this window and click on "Info" this opens a window showing information on the data set in question (for example, on the sources and definitions of the variables). If you find a file that is of interest, you may open it by clicking on "Open", or just double-clicking on the file name. For the moment let's open `data3-6`.

☞ In gretl windows containing lists, double-clicking on a line launches a default action for the associated list entry: e.g. displaying the values of a data series, opening a file.

This file contains data pertaining to a classic econometric "chestnut", the consumption function.

[1] For convenience I will refer to the graphical client program simply as gretl in this manual. Note, however, that the specific name of the program differs according to the computer platform. On Linux it is called `gretl_x11` while on MS Windows it is `gretlw32.exe`. On Linux systems a wrapper script named `gretl` is also installed — see also the *Gretl Command Reference*.

The data window should now display the name of the current data file, the overall data range and sample range, and the names of the variables along with brief descriptive tags — see Figure 2.2.

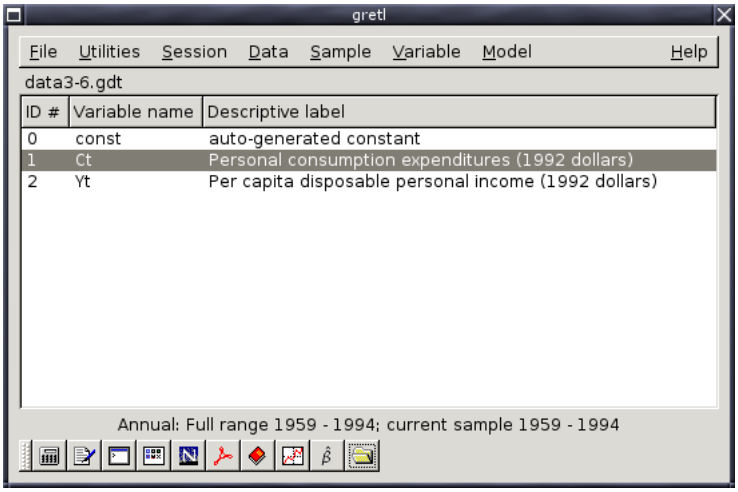


Figure 2.2: Main window, with a practice data file open

OK, what can we do now? Hopefully the various menu options should be fairly self explanatory. For now we'll dip into the Model menu; a brief tour of all the main window menus is given in Section 2.3 below.

gretl's Model menu offers numerous various econometric estimation routines. The simplest and most standard is Ordinary Least Squares (OLS). Selecting OLS pops up a dialog box calling for a *model specification* — see Figure 2.3.

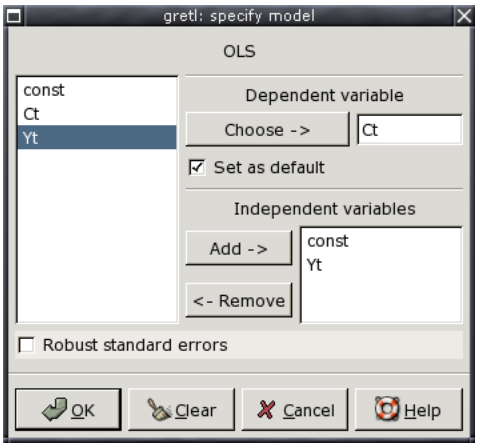


Figure 2.3: Model specification dialog

To select the dependent variable, highlight the variable you want in the list on the left and click the "Choose" button that points to the Dependent variable slot. If you check the "Set as default" box this variable will be pre-selected as dependent when you next open the model dialog box. Shortcut: double-clicking on a variable on the left selects it as dependent and also sets it as the default. To select independent variables, highlight them on the left and click the "Add" button (or click the right mouse button over the highlighted variable). To select several variable in the list box, drag the mouse over them; to select several non-contiguous variables, hold down the `Ctrl` key and click on the variables you want. To run a regression with consumption as the dependent variable and income as independent, click `Ct` into the Dependent slot and add `Yt` to the Independent variables

list.

2.2 Estimation output

Once you've specified a model, a window displaying the regression output will appear. The output is reasonably comprehensive and in a standard format (Figure 2.4).

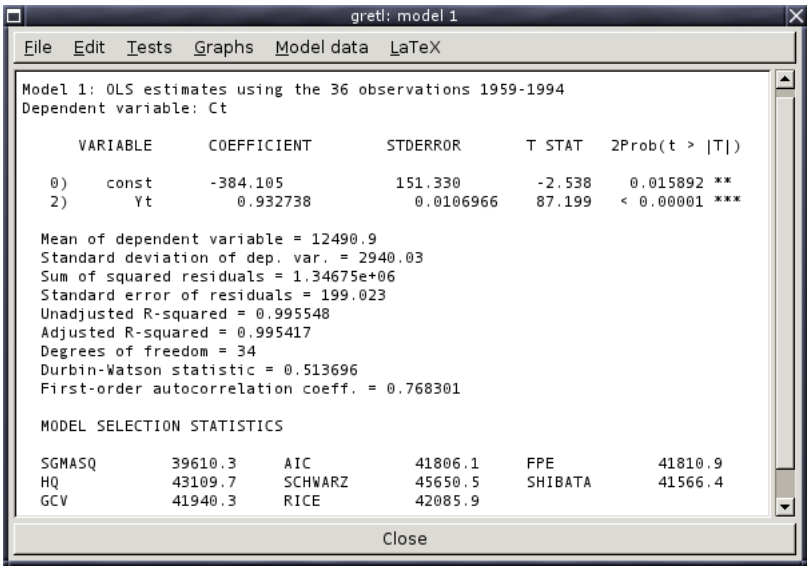


Figure 2.4: Model output window

The output window contains menus that allow you to inspect or graph the residuals and fitted values, and to run various diagnostic tests on the model.

For most models there is also an option to reprint the regression output in LaTeX format. You can print the results in a tabular format (similar to what's in the output window, but properly typeset) or as an equation, across the page. For each of these options you can choose to preview the typeset product, or save the output to file for incorporation in a LaTeX document. Previewing requires that you have a functioning TeX system on your computer. You can control the appearance of gretl's LaTeX output using a file named `gretlpre.tex`, which should be placed in your gretl user directory (see the *Gretl Command Reference*). If a file of this name is found, its contents will be used as the LaTeX "preamble". The default value of the preamble is as follows:

```
\documentclass[11pt]{article}
\usepackage[latin1]{inputenc}
\usepackage{amsmath}
\usepackage{dcolumn,longtable}
\begin{document}
\thispagestyle{empty}
```

Note that the `amsmath` and `dcolumn` packages are required.

To import gretl output into a word processor, you may copy and paste from an output window, using its Edit menu (or Copy button, in some contexts) to the target program. Many (not all) gretl windows offer the option of copying in RTF (Microsoft's "Rich Text Format") or as LaTeX. If you are pasting into a word processor, RTF may be a good option because the tabular formatting of the output is preserved.[2] Alternatively, you can save the output to a (plain text) file then import the

[2]Note that when you copy as RTF under MS Windows, Windows will only allow you to paste the material into applications that "understand" RTF. Thus you will be able to paste into MS Word, but not into notepad. Note also that there

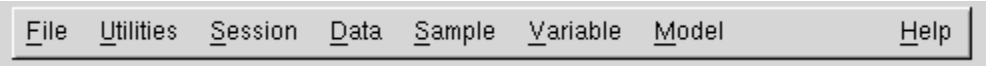
file into the target program. When you finish a gretl session you are given the option of saving all the output from the session to a single file.

Note that on the gnome desktop and under MS Windows, the File menu includes a command to send the output directly to a printer.

☞ When pasting or importing plain text gretl output into a word processor, select a monospaced or typewriter-style font (e.g. Courier) to preserve the output's tabular formatting. Select a small font (10-point Courier should do) to prevent the output lines from being broken in the wrong place.

2.3 The main window menus

Reading left to right along the main window's menu bar, we find the File, Utilities, Session, Data, Sample, Variable, Model and Help menus.



- File menu
  - Open data: Open a native gretl data file or import from other formats. See Chapter 4.
  - Append data: Add data to the current working data set, from a gretl data file, a comma-separated values file or a spreadsheet file.
  - Save data: Save the currently open native gretl data file.
  - Save data as: Write out the current data set in native format, with the option of using gzip data compression. See Chapter 4.
  - Export data: Write out the current data set in Comma Separated Values (CSV) format, or the formats of GNU R or GNU Octave. See Chapter 4 and also Appendix D.
  - Send to: Send the current data set as an e-mail attachment.
  - Clear data set: Clear the current data set out of memory. Generally you don't have to do this (since opening a new data file automatically clears the old one) but sometimes it's useful.
  - Browse databases: See Section 4.3.
  - Create data set: Initialize the built-in spreadsheet for entering data manually. See Section 4.4.
  - View command log: Open a window containing a record of the commands executed so far.
  - Open command file: Open a file of gretl commands, either one you have created yourself or one of the practice files supplied with the package. If you want to create a command file from scratch use the next item, New command file.
  - Preferences: Set the paths to various files gretl needs to access. Choose the font in which gretl displays text output. Select or unselect "expert mode". (If this mode is selected various warning messages are suppressed.) Activate or suppress gretl's messaging about the availability of program updates. Configure or turn on/off the main-window toolbar. See the *Gretl Command Reference* for further details.
  - Exit: Quit the program. If expert mode is not selected you'll be prompted to save any unsaved work.
- Utilities menu

appears to be a bug in some versions of Windows, whereby the paste will not work properly unless the "target" application (e.g. MS Word) is already running prior to copying the material in question.

  - Statistical tables: Look up critical values for commonly used distributions (normal or Gaussian, *t*, chi-square, *F* and Durbin–Watson).
  - p-value finder: Open a window which enables you to look up p-values from the Gaussian, *t*, chi-square, *F* or gamma distributions. See also the `pvalue` command in the *Gretl Command Reference*.
  - Test statistic calculator: Calculate test statistics and p-values for a range of common hypothesis tests (population mean, variance and proportion; difference of means, variances and proportions). The relevant sample statistics must be already available for entry into the dialog box. For some simple tests that take as input data series rather than pre-computed sample statistics, see "Difference of means" and "Difference of variances" under the Data menu.
  - Gretl console: Open a "console" window into which you can type commands as you would using the command-line program, gretlcli (as opposed to using point-and-click).
  - Start Gnu R: Start R (if it is installed on your system), and load a copy of the data set currently open in gretl. See Appendix D.
  - NIST test suite: Check the numerical accuracy of gretl against the reference results for linear regression made available by the (US) National Institute of Standards and Technology.

- Session menu
  - Icon view: Open a window showing the current gretl session as a set of icons. For details see Section 3.4.
  - Open: Open a previously saved session file.
  - Save: Save the current session to file.
  - Save as: Save the current session to file under a chosen name.

- Data menu
  - Display values: Pops up a window with a simple (not editable) printout of the values of the variables (either all of them or a selected subset).
  - Edit values: Pops up a spreadsheet window where you can make changes, add new variables, and extend the number of observations.
  - Sort variables: Rearrange the listing of variables in the main window, either by ID number or alphabetically by name.
  - Graph specified vars: Gives a choice between a time series plot, a regular X–Y scatter plot, an X–Y plot using impulses (vertical bars), an X–Y plot "with factor separation" (i.e. with the points colored differently depending to the value of a given dummy variable), boxplots, and a 3-D graph. Serves up a dialog box where you specify the variables to graph. See Chapter 8 for details.
  - Multiple scatterplots: Show a collection of (at most six) pairwise plots, with either a given variable on the *y* axis plotted against several different variables on the *x* axis, or several *y* variables plotted against a given *x*. May be useful for exploratory data analysis.
  - Read info, Edit info: "Read info" just displays the summary information for the current data file; "Edit info" allows you to make changes to it (if you have permission to do so).
  - Print description: Opens a window containing a full account of the current dataset, including the summary information and any specific information on each of the variables.
  - Summary statistics: Shows a fairly full set of descriptive statistics for all variables in the data set, or for selected variables.
  - Correlation matrix: Shows the pairwise correlation coefficients for all variables in the data set, or selected variables

  - Principal components: Active only if two or more variables are selected; produces a Principal Components Analysis of the selected variables.
  - Mahalonobis distances: Active only if two or more variables are selected; computes the Mahalonobis distance of each observation from the centroid of the selected set of variables.
  - Difference of means: Calculates the t statistic for the null hypothesis that the population means are equal for two selected variables and shows its p-value.
  - Difference of variances: Calculates the F statistic for the null hypothesis that the population variances are equal for two selected variables and shows its p-value.
  - Add variables: Gives a sub-menu of standard transformations of variables (logs, lags, squares, etc.) that you may wish to add to the data set. Also gives the option of adding random variables, and (for time-series data) adding seasonal dummy variables (e.g. quarterly dummy variables for quarterly data). Includes an item for seeding the program's pseudo-random number generator.
  - Add observations: Gives a dialog box in which you can choose a number of observations to add at the end of the current dataset; for use with forecasting.
  - Remove extra observations: Active only if extra observations have been added automatically in the process of forecasting; deletes these extra observations.
  - Refresh window: Sometimes gretl commands generate new variables. The "refresh" item ensures that the listing of variables visible in the main data window is in sync with the program's internal state.

- Sample menu
  - Set range: Select a different starting and/or ending point for the current sample, within the range of data available.
  - Restore full range: self-explanatory.
  - Dataset structure: invokes a series of dialog boxes which allow you to change the structural interpretation of the current dataset. For example, if data were read in as a cross section you can get the program to interpret them as time series or as a panel. See also Chapter 6.
  - Compact data: For time-series data of higher than annual frequency, gives you the option of compacting the data to a lower frequency, using one of four compaction methods (average, sum, start of period or end of period).
  - Expand data: For time-series data, gives you the option of expanding the data to a higher frequency.
  - Define, based on dummy: Given a dummy (indicator) variable with values 0 or 1, this drops from the current sample all observations for which the dummy variable has value 0.
  - Restrict, based on criterion: Similar to the item above, except that you don't need a predefined variable: you supply a Boolean expression (e.g. `sqft > 1400`) and the sample is restricted to observations satisfying that condition. See the entry for `genr` in the *Gretl Command Reference* for details on the Boolean operators that can be used.
  - Random sub-sample: Draw a random sample from the full dataset.
  - Drop all obs with missing values: Drop from the current sample all observations for which at least one variable has a missing value (see Section 4.5).
  - Count missing values: Give a report on observations where data values are missing. May be useful in examining a panel data set, where it's quite common to encounter missing values.
  - Set missing value code: Set a numerical value that will be interpreted as "missing" or "not available".

  - Add case markers: Prompts for the name of a text file containing "case markers" (short strings identifying the individual observations) and adds this information to the data set. See Chapter 4.
  - Remove case markers: Active only if the dataset has case markers indentifying the observations; removes these case markers.
  - Restructure panel: Allows the conversion of a panel data set in stacked cross-section form into stacked time series or vice versa. (Unlike the Dataset structure menu item above, this one actually changes the organization of the data.)
  - Transpose data: Turn each observation into a variable and vice versa (or in other words, each row of the data matrix becomes a column in the modified data matrix); can be useful with imported data that have been read in "sideways".

- Variable menu Most items under here operate on a single variable at a time. The "active" variable is set by highlighting it (clicking on its row) in the main data window. Most options will be self-explanatory. Note that you can rename a variable and can edit its descriptive label under "Edit attributes". You can also "Define a new variable" via a formula (e.g. involving some function of one or more existing variables). For the syntax of such formulae, look at the online help for "Generate variable syntax" or see the `genr` command in the *Gretl Command Reference*. One simple example:

  ```
  foo = x1 * x2
  ```

  will create a new variable `foo` as the product of the existing variables `x1` and `x2`. In these formulae, variables must be referenced by name, not number.

- Model menu For details on the various estimators offered under this menu please consult the *Gretl Command Reference* and/or the online help under "Help, Estimation". Also see Chapter 9 regarding the estimation of nonlinear models.

- Help menu Please use this as needed! It gives details on the syntax required in various dialog entries.

2.4 Keyboard shortcuts

When working in the main gretl window, some common operations may be performed using the keyboard, as shown in the table below.

`Return`	Opens a window displaying the values of the currently selected variables: it is the same as selecting "Data, Display Values".
`Delete`	Pressing this key has the effect of deleting the selected variables. A confirmation is required, to prevent accidental deletions.
`e`	Has the same effect as selecting "Edit attributes" from the "Variable" menu.
`F2`	Same as "e". Included for compatibility with other programs.
`g`	Has the same effect as selecting "Define new variable" from the "Variable" menu (which maps onto the `genr` command).
`h`	Opens a help window for gretl commands.
`F1`	Same as "h". Included for compatibility with other programs.
`t`	Graphs the selected variable; a line graph is used for time-series datasets, whereas a distribution plot is used for cross-sectional data.

2.5 The gretl toolbar

At the bottom left of the main window sits the toolbar.

The icons have the following functions, reading from left to right:

1. Launch a calculator program. A convenience function in case you want quick access to a calculator when you're working in gretl. The default program is `calc.exe` under MS Windows, or `xcalc` under the X window system. You can change the program under the "File, Preferences, General" menu, "Programs" tab.
2. Start a new script. Opens an editor window in which you can type a series of commands to be sent to the program as a batch.
3. Open the gretl console. A shortcut to the "Gretl console" menu item (Section 2.3 above).
4. Open the gretl session window.
5. Open the gretl website in your web browser. This will work only if you are connected to the Internet and have a properly configured browser.
6. Open the current version of this manual, in PDF format. As with the previous item, this requires an Internet connection; it also requires that your browser knows how to handle PDF files.
7. Open the help item for script commands syntax (i.e. a listing with details of all available commands).
8. Open the dialog box for defining a graph.
9. Open the dialog box for estimating a model using ordinary least squares.
10. Open a window listing the datasets associated with Ramanathan's *Introductory Econometrics* (and also the datasets from the various other econometrics texts that are supported by gretl, if they are installed).

If you don't care to have the toolbar displayed, you can turn it off under the "File, Preferences, General" menu. Go o the Toolbar tab and uncheck the "show gretl toolbar" box.

Chapter 3

Modes of working

3.1 Command scripts

As you execute commands in gretl, using the GUI and filling in dialog entries, those commands are recorded in the form of a "script" or batch file. Such scripts can be edited and re-run, using either gretl or the command-line client, gretlcli.

To view the current state of the script at any point in a gretl session, choose "View command log" under the File menu. This log file is called `session.inp` and it is overwritten whenever you start a new session. To preserve it, save the script under a different name. Script files will be found most easily, using the GUI file selector, if you name them with the extension "`.inp`".

To open a script you have written independently, use the "File, Open command file" menu item; to create a script from scratch use the "File, New command file" item or the "new script" toolbar button. In either case a script window will open (see Figure 3.1).

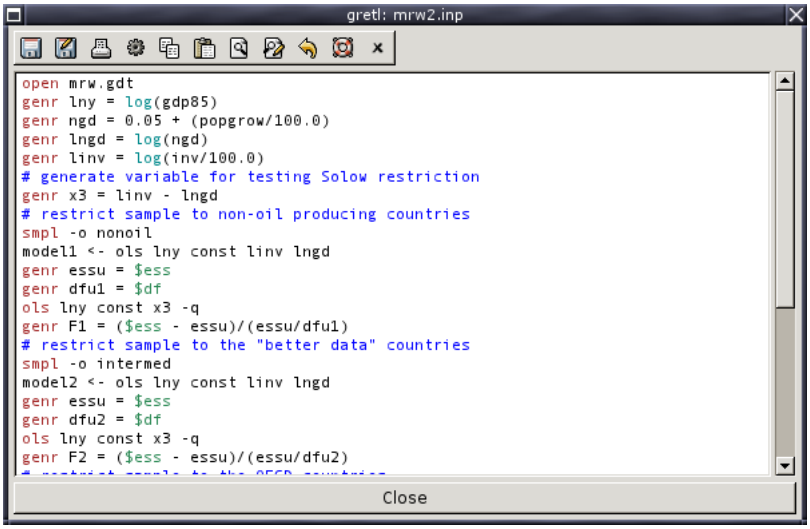


Figure 3.1: Script window, editing a command file

The toolbar at the top of the script window offers the following functions (left to right): (1) Save the file; (2) Save the file under a specified name; (3) Print the file (under Windows or the gnome desktop only); (4) Execute the commands in the file; (5) Copy selected text; (6) Paste the selected text; (7) Find and replace text; (8) Undo the last Paste or Replace action; (9) Help (if you place the cursor in a command word and press the question mark you will get help on that command); (10) Close the window.

When you click the Execute icon or choose the "File, Run" menu item all output is directed to a single window, where it can be edited, saved or copied to the clipboard. To learn more about the possibilities of scripting, take a look at the gretl Help item "Script commands syntax," or start up the command-line program gretlcli and consult its help, or consult the *Gretl Command Reference*.

In addition, the gretl package includes over 70 "practice" scripts. Most of these relate to Ra-

manathan (2002), but they may also be used as a free-standing introduction to scripting in gretl and to various points of econometric theory. You can explore the practice files under "File, Open command file, practice file" There you will find a listing of the files along with a brief description of the points they illustrate and the data they employ. Open any file and run it to see the output. Note that long commands in a script can be broken over two or more lines, using backslash as a continuation character.

You can, if you wish, use the GUI controls and the scripting approach in tandem, exploiting each method where it offers greater convenience. Here are two suggestions.

- Open a data file in the GUI. Explore the data — generate graphs, run regressions, perform tests. Then open the Command log, edit out any redundant commands, and save it under a specific name. Run the script to generate a single file containing a concise record of your work.
- Start by establishing a new script file. Type in any commands that may be required to set up transformations of the data (see the `genr` command in the *Gretl Command Reference*). Typically this sort of thing can be accomplished more efficiently via commands assembled with forethought rather than point-and-click. Then save and run the script: the GUI data window will be updated accordingly. Now you can carry out further exploration of the data via the GUI. To revisit the data at a later point, open and rerun the "preparatory" script first.

3.2 Saving script objects

When you estimate a model using point-and-click, the model results are displayed in a separate window, offering menus which let you perform tests, draw graphs, save data from the model, and so on. Ordinarily, when you estimate a model using a script you just get a non-interactive printout of the results. You can, however, arrange for models estimated in a script to be "captured", so that you can examine them interactively when the script is finished. Here is an example of the syntax for achieving this effect:

```
Model1 <- ols Ct 0 Yt
```

That is, you type a name for the model to be saved under, then a back-pointing "assignment arrow", then the model command. You may use names that have embedded spaces if you like, but such names must always be wrapped in double quotes:

```
"Model 1" <- ols Ct 0 Yt
```

Models saved in this way will appear as icons in the gretl session window (see Section 3.4) after the script is executed. In addition, you can arrange to have a named model displayed (in its own window) automatically as follows:

```
Model1.show
```

Again, if the name contains spaces it must be quoted:

```
"Model 1".show
```

The same facility can be used for graphs. For example the following will create a plot of `Ct` against `Yt`, save it under the name "CrossPlot" (it will appear under this name in the session icon window), and have it displayed:

```
CrossPlot <- gnuplot Ct Yt
CrossPlot.show
```

You can also save the output from selected commands as named pieces of text (again, these will appear in the session icon window, from where you can open them later). For example this command sends the output from an augmented Dickey–Fuller test to a "text object" named ADF1 and displays it in a window:

```
ADF1 <- adf 2 x1
ADF1.show
```

Objects saved in this way (whether models, graphs or pieces of text output) can be destroyed using the command `.free` appended to the name of the object, as in `ADF1.free`.

3.3 The gretl console

A further option is available for your computing convenience. Under gretl's Utilities menu you will find the item "Gretl console" (there is also an "open gretl console" button on the toolbar in the main window). This opens up a window in which you can type commands and execute them one by one (by pressing the Enter key) interactively. This is essentially the same as gretlcli's mode of operation, except that the GUI is updated based on commands executed from the console, enabling you to work back and forth as you wish.

In the console, you have "command history"; that is, you can use the up and down arrow keys to navigate the list of command you have entered to date. You can retrieve, edit and then re-enter a previous command.

In console mode, you can create, display and free objects (models, graphs or text) aa described above for script mode.

3.4 The Session concept

gretl offers the idea of a "session" as a way of keeping track of your work and revisiting it later. The basic idea is to provide an iconic space containing various objects pertaining to your current working session (see Figure 3.2). You can add objects (represented by icons) to this space as you go along. If you save the session, these added objects should be available again if you re-open the session later.

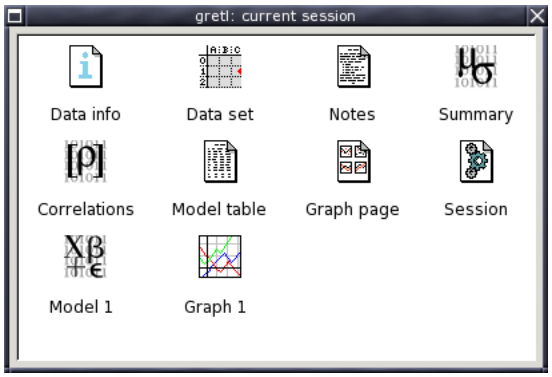


Figure 3.2: Icon view: one model and one graph have been added to the default icons

If you start gretl and open a data set, then select "Icon view" from the Session menu, you should see the basic default set of icons: these give you quick access to the command script ("Session"), information on the data set (if any), correlation matrix ("Correlations") and descriptive summary statistics ("Summary"). All of these are activated by double-clicking the relevant icon. The "Data set" icon is a little more complex: double-clicking opens up the data in the built-in spreadsheet, but you can also right-click on the icon for a menu of other actions.

To add a model to the session view, first estimate it using the Model menu. Then pull down the File menu in the model window and select "Save to session as icon..." or "Save as icon and close". Simply hitting the S key over the model window is a shortcut to the latter action.

To add a graph, first create it (under the Data menu, "Graph specified vars", or via one of gretl's other graph-generating commands). Click on the graph window to bring up the graph menu, and select "Save to session as icon".

Once a model or graph is added its icon should appear in the Icon View window. Double-clicking on the icon redisplays the object, while right-clicking brings up a menu which lets you display or delete the object. This popup menu also gives you the option of editing graphs.

The model table

In econometric research it is common to estimate several models with a common dependent variable — the models differing in respect of which independent variables are included, or perhaps in respect of the estimator used. In this situation it is convenient to present the regression results in the form of a table, where each column contains the results (coefficient estimates and standard errors) for a given model, and each row contains the estimates for a given variable across the models.

In the Icon View window gretl provides a means of constructing such a table (and copying it in plain text, LaTeX or Rich Text Format). Here is how to do it:[1]

1. Estimate a model which you wish to include in the table, and in the model display window, under the File menu, select "Save to session as icon" or "Save as icon and close".
2. Repeat step 1 for the other models to be included in the table (up to a total of six models).
3. When you are done estimating the models, open the icon view of your gretl session, by selecting "Icon view" under the Session menu in the main gretl window, or by clicking the "session icon view" icon on the gretl toolbar.
4. In session icon view, there is an icon labeled "Model table". Decide which model you wish to appear in the left-most column of the model table and add it to the table, either by dragging its icon onto the Model table icon, or by right-clicking on the model icon and selecting "Add to model table" from the pop-up menu.
5. Repeat step 4 for the other models you wish to include in the table. The second model selected will appear in the second column from the left, and so on.
6. When you are finished composing the model table, display it by double-clicking on its icon. Under the Edit menu in the window which appears, you have the option of copying the table to the clipboard in various formats.
7. If the ordering of the models in the table is not what you wanted, right-click on the model table icon and select "Clear table". Then go back to step 4 above and try again.

A simple instance of gretl's model table is shown in Figure 3.3.

The graph page

The "graph page" icon in the session window offers a means of putting together several graphs for printing on a single page. This facility will work only if you have the LaTeX typsetting system installed, and are able to generate and view PostScript output.[2]

[1] The model table can also be built non-interactively, in script mode. For details on how to do this, see the entry for `modeltab` in the *Gretl Command Reference*.

[2] Specifically, you must have dvips and ghostscript installed, along with a viewer such as gv, ggv or kghostview. The default viewer for systems other than MS Windows is gv.

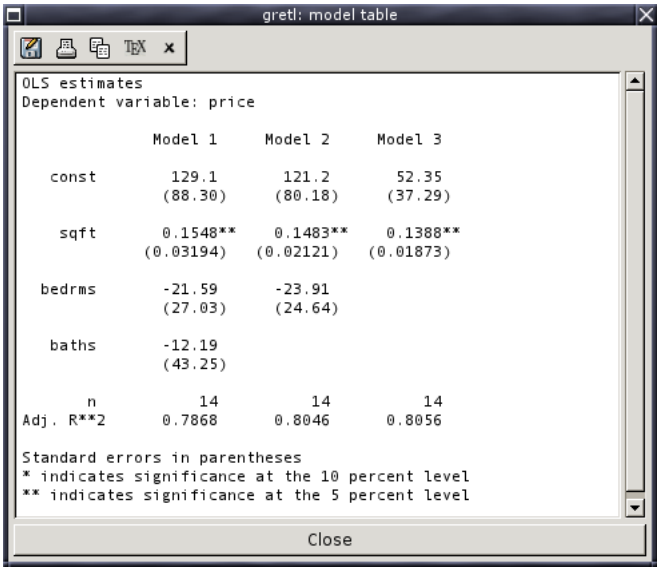


Figure 3.3: Example of model table

In the Icon View window, you can drag up to eight graphs onto the graph page icon. When you double-click on the icon (or right-click and select "Display"), a page containing the selected graphs (in EPS format) will be composed and opened in your postscript viewer. From there you should be able to print the page.

To clear the graph page, right-click on its icon and select "Clear".

On systems other than MS Windows, you may have to adjust the setting for the program used to view postscript. Find that under the "Programs" tab in the Preferences dialog box (under the "File" menu in the main window). On Windows, you may need to adjust your file associations so that the appropriate viewer is called for the "Open" action on files with the `.ps` extension.

Saving and re-opening sessions

If you create models or graphs that you think you may wish to re-examine later, then before quitting gretl select "Save as..." from the Session menu and give a name under which to save the session. To re-open the session later, either

- Start gretl then re-open the session file by going to the "Open" item under the Session menu, or
- From the command line, type `gretl -r` sessionfile, where sessionfile is the name under which the session was saved.

Chapter 4

Data files

4.1 Native format

gretl has its own format for data files. Most users will probably not want to read or write such files outside of gretl itself, but occasionally this may be useful and full details on the file formats are given in Appendix A.

4.2 Other data file formats

gretl will read various other data formats.

- Plain text (ASCII) files. These can be brought in using gretl's "File, Open Data, Import ASCII..." menu item, or the `import` script command. For details on what gretl expects of such files, see Section 4.4.
- Comma-Separated Values (CSV) files. These can be imported using gretl's "File, Open Data, Import CSV..." menu item, or the `import` script command. See also Section 4.4.
- Worksheets in the format of either MS Excel or Gnumeric. These are also brought in using gretl's "File, Open Data, Import" menu. The requirements for such files are given in Section 4.4.
- Stata data files (`.dta`).
- Eviews workfiles (`.wf1`).[1]

When you import data from the ASCII or CSV formats, gretl opens a "diagnostic" window, reporting on its progress in reading the data. If you encounter a problem with ill-formatted data, the messages in this window should give you a handle on fixing the problem.

For the convenience of anyone wanting to carry out more complex data analysis, gretl has a facility for writing out data in the native formats of GNU R and GNU Octave (see Appendix D). In the GUI client this option is found under the "File" menu; in the command-line client use the `store` command with the flag `-r` (R) or `-m` (Octave).

4.3 Binary databases

For working with large amounts of data gretl is supplied with a database-handling routine. A *database*, as opposed to a *data file*, is not read directly into the program's workspace. A database can contain series of mixed frequencies and sample ranges. You open the database and select series to import into the working dataset. You can then save those series in a native format data file if you wish. Databases can be accessed via gretl's menu item "File, Browse databases".

For details on the format of gretl databases, see Appendix A.

[1] This is somewhat experimental. See http://www.ecn.wfu.edu/eviews_format/.

Online access to databases

As of version 0.40, gretl is able to access databases via the internet. Several databases are available from Wake Forest University. Your computer must be connected to the internet for this option to work. Please see the item on "Online databases" under gretl's Help menu.

RATS 4 databases

Thanks to Thomas Doan of *Estima*, who provided me with the specification of the database format used by RATS 4 (Regression Analysis of Time Series), gretl can also handle such databases. Well, actually, a subset of same: I have only worked on time-series databases containing monthly and quarterly series. My university has the RATS G7 database containing data for the seven largest OECD economies and gretl will read that OK.

☞ Visit the gretl data page for details and updates on available data.

4.4 Creating a data file from scratch

There are five ways to do this:

1. Find, or create using a text editor, a plain text data file and open it with gretl's "Import ASCII" option.
2. Use your favorite spreadsheet to establish the data file, save it in Comma Separated Values format if necessary (this should not be necessary if the spreadsheet program is MS Excel or Gnumeric), then use one of gretl's "Import" options (CSV, Excel or Gnumeric, as the case may be).
3. Use gretl's built-in spreadsheet.
4. Select data series from a suitable database.
5. Use your favorite text editor or other software tools to a create data file in gretl format independently.

Here are a few comments and details on these methods.

Common points on imported data

Options (1) and (2) involve using gretl's "import" mechanism. For gretl to read such data successfully, certain general conditions must be satisfied:

- The first row must contain valid variable names. A valid variable name is of 15 characters maximum; starts with a letter; and contains nothing but letters, numbers and the underscore character, _. (Longer variable names will be truncated to 15 characters.) Qualifications to the above: First, in the case of an ASCII or CSV import, if the file contains no row with variable names the program will automatically add names, v1, v2 and so on. Second, by "the first row" is meant the first *relevant* row. In the case of ASCII and CSV imports, blank rows and rows beginning with a hash mark, #, are ignored. In the case of Excel and Gnumeric imports, you are presented with a dialog box where you can select an offset into the spreadsheet, so that gretl will ignore a specified number of rows and/or columns.
- Data values: these should constitute a rectangular block, with one variable per column (and one observation per row). The number of variables (data columns) must match the number of variable names given. See also Section 4.5. Numeric data are expected, but in the case of importing from ASCII/CSV, the program offers limited handling of character (string) data: if a given column contains character data only, consecutive numeric codes are substituted for

the strings, and once the import is complete a table is printed showing the correspondence between the strings and the codes.

- Dates (or observation labels): Optionally, the *first* column may contain strings such as dates, or labels for cross-sectional observations. Such strings have a maximum of 8 characters (as with variable names, longer strings will be truncated). A column of this sort should be headed with the string `obs` or `date`, or the first row entry may be left blank.

  For dates to be recognized as such, the date strings must adhere to one or other of a set of specific formats, as follows. For *annual* data: 4-digit years. For *quarterly* data: a 4-digit year, followed by a separator (either a period, a colon, or the letter `Q`), followed by a 1-digit quarter. Examples: `1997.1`, `2002:3`, `1947Q1`. For *monthly* data: a 4-digit year, followed by a period or a colon, followed by a two-digit month. Examples: `1997.01`, `2002:10`.

CSV files can use comma, space or tab as the column separator. When you use the "Import CSV" menu item you are prompted to specify the separator. In the case of "Import ASCII" the program attempts to auto-detect the separator that was used.

If you use a spreadsheet to prepare your data you are able to carry out various transformations of the "raw" data with ease (adding things up, taking percentages or whatever): note, however, that you can also do this sort of thing easily — perhaps more easily — within gretl, by using the tools under the "Data, Add variables" menu and/or "Variable, define new variable".

Appending imported data

You may wish to establish a gretl dataset piece by piece, by incremental importation of data from other sources. This is supported via the "File, Append data" menu items: gretl will check the new data for conformability with the existing dataset and, if everything seems OK, will merge the data. You can add new variables in this way, provided the data frequency matches that of the existing dataset. Or you can append new observations for data series that are already present; in this case the variable names must match up correctly. Note that by default (that is, if you choose "Open data" rather than "Append data"), opening a new data file closes the current one.

Using the built-in spreadsheet

Under gretl's "File, Create data set" menu you can choose the sort of dataset you want to establish (e.g. quarterly time series, cross-sectional). You will then be prompted for starting and ending dates (or observation numbers) and the name of the first variable to add to the dataset. After supplying this information you will be faced with a simple spreadsheet into which you can type data values. In the spreadsheet window, clicking the right mouse button will invoke a popup menu which enables you to add a new variable (column), to add an observation (append a row at the foot of the sheet), or to insert an observation at the selected point (move the data down and insert a blank row.)

Once you have entered data into the spreadsheet you import these into gretl's workspace using the spreadsheet's "Apply changes" button.

Please note that gretl's spreadsheet is quite basic and has no support for functions or formulas. Data transformations are done via the "Data" or "Variable" menus in the main gretl window.

Selecting from a database

Another alternative is to establish your dataset by selecting variables from a database. gretl comes with a database of US macroeconomic time series and, as mentioned above, the program will reads RATS 4 databases.

Begin with gretl's "File, Browse databases" menu item. This has three forks: "gretl native", "RATS 4" and "on database server". You should be able to find the file `fedstl.bin` in the file selector that opens if you choose the "gretl native" option — this file, which contains a large collection of US macroeconomic time series, is supplied with the distribution.

You won't find anything under "RATS 4" unless you have purchased RATS data.[2] If you do possess RATS data you should go into gretl's "File, Preferences, General" dialog, select the Databases tab, and fill in the correct path to your RATS files.

If your computer is connected to the internet you should find several databases (at Wake Forest University) under "on database server". You can browse these remotely; you also have the option of installing them onto your own computer. The initial remote databases window has an item showing, for each file, whether it is already installed locally (and if so, if the local version is up to date with the version at Wake Forest).

Assuming you have managed to open a database you can import selected series into gretl's workspace by using the "Import" menu item in the database window, or via the popup menu that appears if you click the right mouse button, or by dragging the series into the program's main window.

Creating a gretl data file independently

It is possible to create a data file in one or other of gretl's own formats using a text editor or software tools such as awk, sed or perl. This may be a good choice if you have large amounts of data already in machine readable form. You will, of course, need to study the gretl data formats (XML format or "traditional" format) as described in Appendix A.

Further note

gretl has no problem compacting data series of relatively high frequency (e.g. monthly) to a lower frequency (e.g. quarterly): you are given a choice of method (average, sum, start of period, or end of period). But it has no way of converting lower frequency data to higher. Therefore if you want to import series of various different frequencies from a database into gretl *you must start by importing a series of the lowest frequency you intend to use.* This will initialize your gretl dataset to the low frequency, and higher frequency data can be imported subsequently (they will be compacted automatically). If you start with a high frequency series you will not be able to import any series of lower frequency.

4.5 Missing data values

These are represented internally as DBL_MAX, the largest floating-point number that can be represented on the system (which is likely to be at least 10 to the power 300, and so should not be confused with legitimate data values). In a native-format data file they should be represented as NA. When importing CSV data gretl accepts several common representations of missing values including −999, the string NA (in upper or lower case), a single dot, or simply a blank cell. Blank cells should, of course, be properly delimited, e.g. `120.6,,5.38`, in which the middle value is presumed missing.

As for handling of missing values in the course of statistical analysis, gretl does the following:

- In calculating descriptive statistics (mean, standard deviation, etc.) under the `summary` command, missing values are simply skipped and the sample size adjusted appropriately.
- In running regressions gretl first adjusts the beginning and end of the sample range, truncating the sample if need be. Missing values at the beginning of the sample are common in time series work due to the inclusion of lags, first differences and so on; missing values at the end of the range are not uncommon due to differential updating of series and possibly the inclusion of leads.

If gretl detects any missing values "inside" the (possibly truncated) sample range for a regression, the result depends on the character of the dataset and the estimator chosen. In many cases, the

[2] See www.estima.com

program will automatically skip the missing observations when calculating the regression results. In this situation a message is printed stating how many observations were dropped. On the other hand, the skipping of missing observations is not supported for all procedures: exceptions include all autoregressive estimators, system estimators such as SUR, and nonlinear least squares. In the case of panel data, the skipping of missing observations is supported only if their omission leaves a balanced panel. If missing observations are found in cases where they are not supported, gretl gives an error message and refuses to produce estimates.

In case missing values in the middle of a dataset present a problem, the `misszero` function (use with care!) is provided under the `genr` command. By doing `genr foo = misszero(bar)` you can produce a series `foo` which is identical to `bar` except that any missing values become zeros. Then you can use carefully constructed dummy variables to, in effect, drop the missing observations from the regression while retaining the surrounding sample range.[3]

4.6 Data file collections

If you're using gretl in a teaching context you may be interested in adding a collection of data files and/or scripts that relate specifically to your course, in such a way that students can browse and access them easily.

This is quite easy as of gretl version 1.2.1. There are three ways to access such collections of files:

- For data files: select the menu item "File, Open data, sample file", or click on the folder icon on the gretl toolbar.
- For script files: select the menu item "File, Open command file, practice file".

When a user selects one of the items:

- The data or script files included in the gretl distribution are automatically shown (this includes files relating to Ramanathan's *Introductory Econometrics* and Greene's *Econometric Analysis*).
- The program looks for certain known collections of data files available as optional extras, for instance the datafiles from various econometrics textbooks (Wooldridge, Gujarati, Stock and Watson) and the Penn World Table (PWT 5.6). (See the data page at the gretl website for information on these collections.) If the additional files are found, they are added to the selection windows.
- The program then searches for valid file collections (not necessarily known in advance) in these places: the "system" data directory, the system script directory, the user directory, and all first-level subdirectories of these. (For reference, typical values for these directories are shown in Table 4.1.)

	Linux	*MS Windows*
system data dir	`/usr/share/gretl/data`	`c:\userdata\gretl\data`
system script dir	`/usr/share/gretl/scripts`	`c:\userdata\gretl\scripts`
user dir	`/home/me/gretl`	`c:\userdata\gretl\user`

Table 4.1: Typical locations for file collections

Any valid collections will be added to the selection windows. So what constitutes a valid file collection? This comprises either a set of data files in gretl XML format (with the `.gdt` suffix) or a set of

[3] `genr` also offers the inverse function to `misszero`, namely `zeromiss`, which replaces zeros in a given series with the missing observation code.

script files containing gretl commands (with `.inp` suffix), in each case accompanied by a "master file" or catalog. The gretl distribution contains several example catalog files, for instance the file `descriptions` in the `misc` sub-directory of the gretl data directory and `ps_descriptions` in the `misc` sub-directory of the scripts directory.

If you are adding your own collection, data catalogs should be named `descriptions` and script catalogs should be be named `ps_descriptions`. In each case the catalog should be placed (along with the associated data or script files) in its own specific sub-directory (e.g. `/usr/share/gretl/data/mydata` or `c:\userdata\gretl\data\mydata`).

The syntax of the (plain text) description files is straightforward. Here, for example, are the first few lines of gretl's "misc" data catalog:

```
# Gretl: various illustrative datafiles
"arma","artificial data for ARMA script example"
"ects_nls","Nonlinear least squares example"
"hamilton","Prices and exchange rate, U.S. and Italy"
```

The first line, which must start with a hash mark, contains a short name, here "Gretl", which will appear as the label for this collection's tab in the data browser window, followed by a colon, followed by an optional short description of the collection.

Subsequent lines contain two elements, separated by a comma and wrapped in double quotation marks. The first is a datafile name (leave off the `.gdt` suffix here) and the second is a short description of the content of that datafile. There should be one such line for each datafile in the collection.

A script catalog file looks very similar, except that there are three fields in the file lines: a filename (without its `.inp` suffix), a brief description of the econometric point illustrated in the script, and a brief indication of the nature of the data used. Again, here are the first few lines of the supplied "misc" script catalog:

```
# Gretl: various sample scripts
"arma","ARMA modeling","artificial data"
"ects_nls","Nonlinear least squares (Davidson)","artificial data"
"leverage","Influential observations","artificial data"
"longley","Multicollinearity","US employment"
```

If you want to make your own data collection available to users, these are the steps:

1. Assemble the data, in whatever format is convenient.
2. Convert the data to gretl format and save as `gdt` files. It is probably easiest to convert the data by importing them into the program from plain text, CSV, or a spreadsheet format (MS Excel or Gnumeric) then saving them. You may wish to add descriptions of the individual variables (the "Variable, Edit attributes" menu item), and add information on the source of the data (the "Data, Edit info" menu item).
3. Write a descriptions file for the collection using a text editor.
4. Put the datafiles plus the descriptions file in a subdirectory of the gretl data directory (or user directory).
5. If the collection is to be distributed to other people, package the data files and catalog in some suitable manner, e.g. as a zipfile.

If you assemble such a collection, and the data are not proprietary, I would encourage you to submit the collection for packaging as a gretl optional extra.

Chapter 5

Special functions in genr

5.1 Introduction

The genr command provides a flexible means of defining new variables. It is documented in the *Gretl Command Reference*. This chapter offers a more expansive discussion of some of the special functions available via genr and some of the finer points of the command.

5.2 Time-series filters

One sort of specialized function in genr is the time-series filter. Two such filters are currently available, the Hodrick–Prescott filter and the Baxter–King bandpass filter. These are accessed using hpfilt() and bkfilt() respectively: in each case the function takes one argument, the name of the variable to be processed.

The Hodrick–Prescott filter

A time series y_t may be decomposed into a trend or growth component g_t and a cyclical component c_t.

$$y_t = g_t + c_t, \quad t = 1, 2, \ldots, T$$

The Hodrick–Prescott filter effects such a decomposition by minimizing the following:

$$\sum_{t=1}^{T} (y_t - g_t)^2 + \lambda \sum_{t=2}^{T-1} \left((g_{t+1} - g_t) - (g_t - g_{t-1}) \right)^2 .$$

The first term above is the sum of squared cyclical components $c_t = y_t - g_t$. The second term is a multiple λ of the sum of squares of the trend component's second differences. This second term penalizes variations in the growth rate of the trend component: the larger the value of λ, the higher is the penalty and hence the smoother the trend series.

Note that the hpfilt function in gretl produces the cyclical component, c_t, of the original series. If you want the smoothed trend you can subtract the cycle from the original:

```
genr ct = hpfilt(yt)
genr gt = yt - ct
```

Hodrick and Prescott (1997) suggest that a value of $\lambda = 1600$ is reasonable for quarterly data. The default value in gretl is 100 times the square of the data frequency (which, of course, yields 1600 for quarterly data). The value can be adjusted using the set command, with a parameter of hp_lambda. For example, set hp_lambda 1200.

The Baxter and King filter

Consider the spectral representation of a time series y_t:

$$y_t = \int_{-\pi}^{\pi} e^{i\omega} \mathrm{d}Z(\omega)$$

To extract the component of y_t that lies between the frequencies $\underline{\omega}$ and $\overline{\omega}$ one could apply a bandpass filter:

$$c_t^* = \int_{-\pi}^{\pi} F^*(\omega) e^{i\omega} \mathrm{d}Z(\omega)$$

where $F^*(\omega) = 1$ for $\underline{\omega} < |\omega| < \overline{\omega}$ and 0 elsewhere. This would imply, in the time domain, applying to the series a filter with an infinite number of coefficients, which is undesirable. The Baxter and King bandpass filter applies to y_t a finite polynomial in the lag operator $A(L)$:

$$c_t = A(L) y_t$$

where $A(L)$ is defined as

$$A(L) = \sum_{i=-k}^{k} a_i L^i$$

The coefficients a_i are chosen such that $F(\omega) = A(e^{i\omega})A(e^{-i\omega})$ is the best approximation to $F^*(\omega)$ for a given k. Clearly, the higher k the better the approximation is, but since $2k$ observations have to be discarded, a compromise is usually sought. Moreover, the filter has also other appealing theoretical properties, among which the property that $A(1) = 0$, so a series with a single unit root is made stationary by application of the filter.

In practice, the filter is normally used with monthly or quarterly data to extract the "business cycle" component, namely the component between 6 and 36 quarters. Usual choices for k are 8 or 12 (maybe higher for monthly series). The default values for the frequency bounds are 8 and 32, and the default value for the approximation order, k, is 8. You can adjust these values using the `set` command. The keyword for setting the frequency limits is `bkbp_limits` and the keyword for k is `bkbp_k`. Thus for example if you were using monthly data and wanted to adjust the frequency bounds to 18 and 96, and k to 24, you could do

```
set bkbp_limits 18 96
set bkbp_k 24
```

These values would then remain in force for calls to the `bkfilt` function until changed by a further use of `set`.

5.3 Resampling and bootstrapping

Another specialized function is the resampling, with replacement, of a series. Given an original data series `x`, the command

```
genr xr = resample(x)
```

creates a new series each of whose elements is drawn at random from the elements of `x`. If the original series has 100 observations, each element of `x` is selected with probability 1/100 at each drawing. Thus the effect is to "shuffle" the elements of `x`, with the twist that each element of `x` may appear more than once, or not at all, in `xr`.

The primary use of this function is in the construction of bootstrap confidence intervals or p-values. Here is a simple example. Suppose we estimate a simple regression of y on x via OLS and find that the slope coefficient has a reported t-ratio of 2.5 with 40 degrees of freedom. The two-tailed p-value for the null hypothesis that the slope parameter equals zero is then 0.0166, using the $t(40)$ distribution. Depending on the context, however, we may doubt whether the ratio of coefficient to standard error truly follows the $t(40)$ distribution. In that case we could derive a bootstrap p-value as shown in Example 5.1.

Under the null hypothesis that the slope with respect to x is zero, y is simply equal to its mean plus an error term. We simulate y by resampling the residuals from the initial OLS and re-estimate the model. We repeat this procedure a large number of times, and record the number of cases where

the absolute value of the t-ratio is greater than 2.5: the proportion of such cases is our bootstrap p-value. For a good discussion of simulation-based tests and bootstrapping, see Davidson and MacKinnon (2004, chapter 4).

Example 5.1: Calculation of bootstrap p-value

```
ols y 0 x
# save the residuals
genr ui = $uhat
scalar ybar = mean(y)
# number of replications for bootstrap
scalar replics = 10000
scalar tcount = 0
series ysim = 0
loop replics --quiet
  # generate simulated y by resampling
  ysim = ybar + resample(ui)
  ols ysim 0 x
  scalar tsim = abs($coeff(x) / $stderr(x))
  tcount += (tsim > 2.5)
endloop
printf "proportion of cases with |t| > 2.5 = %g\n", \
   tcount / replics
```

5.4 Handling missing values

Four special functions are available for the handling of missing values. The boolean function `missing()` takes the name of a variable as its single argument; it returns a series with value 1 for each observation at which the given variable has a missing value, and value 0 otherwise (that is, if the given variable has a valid value at that observation). The function `ok()` is complementary to `missing`; it is just a shorthand for `!missing` (where `!` is the boolean NOT operator). For example, one can count the missing values for variable `x` using

```
genr nmiss_x = sum(missing(x))
```

The function `zeromiss()`, which again takes a single series as its argument, returns a series where all zero values are set to the missing code. This should be used with caution — one does not want to confuse missing values and zeros — but it can be useful in some contexts. For example, one can determine the first valid observation for a variable `x` using

```
genr time
genr x0 = min(zeromiss(time * ok(x)))
```

The function `misszero()` does the opposite of `zeromiss`, that is, it converts all missing values to zero.

It may be worth commenting on the propagation of missing values within `genr` formulae. The general rule is that in arithmetical operations involving two variables, if either of the variables has a missing value at observation t then the resulting series will also have a missing value at t. The one exception to this rule is multiplication by zero: zero times a missing value produces zero (since this is mathematically valid regardless of the unknown value).

5.5 Retrieving internal variables

The `genr` command provides a means of retrieving various values calculated by the program in the course of estimating models or testing hypotheses. The variables that can be retrieved in this

way are listed in the *Gretl Command Reference*; here we say a bit more about the special variables `$test` and `$pvalue`.

These variables hold, respectively, the value of the last test statistic calculated using an explicit testing command and the p-value for that test statistic. If no such test has been performed at the time when these variables are referenced, they will produce the missing value code. The "explicit testing commands" that work in this way are as follows: `add` (joint test for the significance of variables added to a model); `adf` (Augmented Dickey–Fuller test, see below); `arch` (test for ARCH); `chow` (Chow test for a structural break); `coeffsum` (test for the sum of specified coefficients); `cusum` (the Harvey–Collier *t*-statistic); `kpss` (KPSS stationarity test, no p-value available); `lmtest` (see below); `meantest` (test for difference of means); `omit` (joint test for the significance of variables omitted from a model); `reset` (Ramsey's RESET); `restrict` (general linear restriction); `runs` (runs test for randomness); `testuhat` (test for normality of residual); and `vartest` (test for difference of variances). In most cases both a `$test` and a `$pvalue` are stored; the exception is the KPSS test, for which a p-value is not currently available.

An important point to notice about this mechanism is that the internal variables `$test` and `$pvalue` are over-written each time one of the tests listed above is performed. If you want to reference these values, you must do so at the correct point in the sequence of gretl commands.

A related point is that some of the test commands generate, by default, more than one test statistic and p-value; in these cases only the last values are stored. To get proper control over the retrieval of values via `$test` and `$pvalue` you should formulate the test command in such a way that the result is unambiguous. This comment applies in particular to the `adf` and `lmtest` commands.

- By default, the `adf` command generates three variants of the Dickey–Fuller test: one based on a regression including a constant, one using a constant and linear trend, and one using a constant and a quadratic trend. When you wish to reference `$test` or `$pvalue` in connection with this command, you can control the variant that is recorded by using one of the flags `--nc`, `--c`, `--ct` or `--ctt` with `adf`.
- By default, the `lmtest` command (which must follow an OLS regression) performs several diagnostic tests on the regression in question. To control what is recorded in `$test` and `$pvalue` you should limit the test using one of the flags `--logs`, `--autocorr`, `--squares` or `--white`.

As an aid in working with values retrieved using `$test` and `$pvalue`, the nature of the test to which these values relate is written into the descriptive label for the generated variable. You can read the label for the variable using the `label` command (with just one argument, the name of the variable), to check that you have retrieved the right value. The following interactive session illustrates this point.

```
? adf 4 x1 --c

Augmented Dickey-Fuller tests, order 4, for x1
sample size 59
unit-root null hypothesis: a = 1

  test with constant
  model: (1 - L)y = b0 + (a-1)*y(-1) + ... + e
  estimated value of (a - 1): -0.216889
  test statistic: t = -1.83491
  asymptotic p-value 0.3638

P-values based on MacKinnon (JAE, 1996)
? genr pv = $pvalue
Generated scalar pv (ID 13) = 0.363844
? label pv
  pv=Dickey-Fuller pvalue (scalar)
```

Chapter 6

Panel data

6.1 Panel structure

Panel data are inherently three dimensional — the dimensions being variable, cross-sectional unit, and time-period. For representation in a textual computer file (and also for gretl's internal calculations) these three dimensions must somehow be flattened into two. This "flattening" involves taking layers of the data that would naturally stack in a third dimension, and stacking them in the vertical dimension.

Gretl always expects data to be arranged "by observation", that is, such that each row represents an observation (and each variable occupies one and only one column). In this context the flattening of a panel data set can be done in either of two ways:

- Stacked cross-sections: the successive vertical blocks each comprise a cross-section for a given period.
- Stacked time-series: the successive vertical blocks each comprise a time series for a given cross-sectional unit.

You may use whichever arrangement is more convenient. Under gretl's Sample menu you will find an item "Restructure panel" which allows you to convert from stacked cross section form to stacked time series or vice versa.

When you import panel data into gretl from a spreadsheet or comma separated format, the panel nature of the data will not be recognized automatically (most likely the data will be treated as "undated"). A panel interpretation can be imposed on the data in either of two ways.

1. Use the GUI menu item "Sample, Dataset structure". In the first dialog box that appears, select "Panel". In the next dialog, make a selection between stacked time series or stacked cross sections depending on how your data are organized. In the next, supply the number of cross-sectional units in the dataset. Finally, check the specification that is shown to you, and confirm the change if it looks OK.
2. Use the script command `setobs`. For panel data this command takes the form `setobs` *freq* `1:1` structure, where *freq* is replaced by the "block size" of the data (that is, the number of periods in the case of stacked time series, or the number of cross-sectional units in the case of stacked cross-sections) and structure is either `--stacked-time-series` or `--stacked-cross-section`. Two examples are given below: the first is suitable for a panel in the form of stacked time series with observations from 20 periods; the second for stacked cross sections with 5 cross-sectional units.

```
setobs 20 1:1 --stacked-time-series
setobs 5 1:1 --stacked-cross-section
```

Panel data arranged by variable

Publicly available panel data sometimes come arranged "by variable." Suppose we have data on two variables, `x1` and `x2`, for each of 50 states in each of 5 years (giving a total of 250 observations

per variable). One textual representation of such a data set would start with a block for x1, with 50 rows corresponding to the states and 5 columns corresponding to the years. This would be followed, vertically, by a block with the same structure for variable x2. A fragment of such a data file is shown below, with quinquennial observations 1965–1985. Imagine the table continued for 48 more states, followed by another 50 rows for variable x2.

x1					
	1965	1970	1975	1980	1985
AR	100.0	110.5	118.7	131.2	160.4
AZ	100.0	104.3	113.8	120.9	140.6

If a datafile with this sort of structure is read into gretl, the program will interpret the columns as distinct variables, so the data will not be usable "as is." But there is a mechanism for correcting the situation, namely the stack function within the genr command.

Consider the first data column in the fragment above: the first 50 rows of this column constitute a cross-section for the variable x1 in the year 1965. If we could create a new variable by stacking the first 50 entries in the second column underneath the first 50 entries in the first, we would be on the way to making a data set "by observation" (in the first of the two forms mentioned above, stacked cross-sections). That is, we'd have a column comprising a cross-section for x1 in 1965, followed by a cross-section for the same variable in 1970.

The following gretl script illustrates how we can accomplish the stacking, for both x1 and x2. We assume that the original data file is called panel.txt, and that in this file the columns are headed with "variable names" p1, p2, . . . , p5. (The columns are not really variables, but in the first instance we "pretend" that they are.)

```
open panel.txt
genr x1 = stack(p1..p5) --length=50
genr x2 = stack(p1..p5) --offset=50 --length=50
setobs 50 1.01 --stacked-cross-section
store panel.gdt x1 x2
```

The second line illustrates the syntax of the stack function. The double dots within the parentheses indicate a range of variables to be stacked: here we want to stack all 5 columns (for all 5 years). The full data set contains 100 rows; in the stacking of variable x1 we wish to read only the first 50 rows from each column: we achieve this by adding --length=50. Note that if you want to stack a non-contiguous set of columns you can put a comma-separated list within the parentheses, as in

```
genr x = stack(p1,p3,p5)
```

On line 3 we do the stacking for variable x2. Again we want a length of 50 for the components of the stacked series, but this time we want gretl to start reading from the 50th row of the original data, and we specify --offset=50.

Line 4 imposes a panel interpretation on the data, as explained in section 6.1. Finally, we save the data in gretl format, with the panel interpretation, discarding the original "variables" p1 through p5.

The illustrative script above is appropriate when the number of variable to be processed is small. When then are many variables in the data set it will be more efficient to use a command loop to accomplish the stacking, as shown in the following script. The setup is presumed to be the same as in the previous section (50 units, 5 periods), but with 20 variables rather than 2.

```
open panel.txt
loop for i=1..20
  genr k = ($i - 1) * 50
```

```
  genr x$i = stack(p1..p5) --offset=k --length=50
endloop
setobs 50 1.01 --stacked-cross-section
store panel.gdt x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 \
  x11 x12 x13 x14 x15 x16 x17 x18 x19 x20
```

6.2 Dummy variables

In a panel study you may wish to construct dummy variables of one or both of the following sorts: (a) dummies as unique identifiers for the cross-sectional units, and (b) dummies as unique identifiers for the time periods. The former may be used to allow the intercept of the regression to differ across the units, the latter to allow the intercept to differ across periods.

Three special functions are available to create such dummies. These are found under the "Data, Add variables" menu in the GUI, or under the `genr` command in script mode or gretlcli.

1. "periodic dummies" (script command `genr dummy`). This command creates a set of dummy variables identifying the periods. The variable `dummy_1` will have value 1 in each row corresponding to a period 1 observation, 0 otherwise; `dummy_2` will have value 1 in each row corresponding to a period 2 observation, 0 otherwise; and so on.
2. "unit dummies" (script command `genr unitdum`). This command creates a set of dummy variables identifying the cross-sectional units. The variable `du_1` will have value 1 in each row corresponding to a unit 1 observation, 0 otherwise; `du_2` will have value 1 in each row corresponding to a unit 2 observation, 0 otherwise; and so on.
3. "panel dummies" (script command `genr paneldum`). This creates both period and unit dummy variables. The unit dummies are named `du_1`, `du_2` and so on, while the period dummies are named `dt_1`, `dt_2`, etc.

If a panel data set has the `YEAR` of the observation entered as one of the variables you can create a periodic dummy to pick out a particular year, e.g. `genr dum = (YEAR=1960)`. You can also create periodic dummy variables using the modulus operator, `%`. For instance, to create a dummy with value 1 for the first observation and every thirtieth observation thereafter, 0 otherwise, do

```
genr index
genr dum = ((index-1)%30) = 0
```

6.3 Lags and differences with panel data

If the time periods are evenly spaced you may want to use lagged values of variables in a panel regression; you may also with to construct first differences of variables of interest.

Once a dataset is properly identified as a panel, gretl will handle the generation of such variables correctly. For example the command `genr x1_1 = x1(-1)` will create a variable that contains the first lag of `x1` where available, and the missing value code where the lag is not available. When you run a regression using such variables, the program will automatically skip the missing observations.

6.4 Pooled estimation

There is a special purpose estimation command for use with panel data, the "Pooled OLS" option under the Model menu. This command is available only if the data set is recognized as a panel. To take advantage of it, you should specify a model without any dummy variables representing cross-sectional units. The routine presents estimates for straightforward pooled OLS, which treats cross-sectional and time-series variation at par. This model may or may not be appropriate. Under

the Tests menu in the model window, you will find an item "panel diagnostics", which tests pooled OLS against the principal alternatives, the fixed effects and random effects models.

The *fixed effects* model adds a dummy variable for all but one of the cross-sectional units, allowing the intercept of the regression to vary across the units. An F-test for the joint significance of these dummies is presented: if the p-value for this test is small, that counts against the null hypothesis (that the simple pooled model is adequate) and in favor of the fixed effects model.

The *random effects* model, on the other hand, decomposes the residual variance into two parts, one part specific to the cross-sectional unit or "group" and the other specific to the particular observation. (This estimator can be computed only if the panel is "wide" enough, that is, if the number of cross-sectional units in the data set exceeds the number of parameters to be estimated.) The Breusch–Pagan LM statistic tests the null hypothesis (again, that the pooled OLS estimator is adequate) against the random effects alternative.

It is quite possible that the pooled OLS model is rejected against both of the alternatives, fixed effects and random effects. How, then, do we assess the relative merits of the two alternative estimators? The Hausman test (also reported, provided the random effects model can be estimated) addresses this issue. If the unit- or group-specific error is uncorrelated with the independent variables, the random effects estimator is more efficient than the fixed effects estimator; otherwise the random effects estimator is inconsistent, in which case the fixed effects estimator is to be preferred. The null hypothesis for the Hausman test is that the group-specific error is not so correlated (and therefore the random effects model is preferable). Thus a low p-value for this tests counts against the random effects model and in favor of fixed effects.

For a rigorous discussion of this topic, see Greene (2000), chapter 14.

6.5 Illustration: the Penn World Table

The Penn World Table (homepage at pwt.econ.upenn.edu) is a rich macroeconomic panel dataset, spanning 152 countries over the years 1950–1992. The data are available in gretl format; please see the gretl data site (this is a free download, although it is not included in the main gretl package).

Example 6.1 opens `pwt56_60_89.gdt`, a subset of the PWT containing data on 120 countries, 1960–89, for 20 variables, with no missing observations (the full data set, which is also supplied in the pwt package for gretl, has many missing observations). Total growth of real GDP, 1960–89, is calculated for each country and regressed against the 1960 level of real GDP, to see if there is evidence for "convergence" (i.e. faster growth on the part of countries starting from a low base).

Example 6.1: Use of the Penn World Table

```
open pwt56_60_89.gdt
# for 1989 (the last obs), lag 29 gives 1960, the first obs
genr gdp60 = RGDPL(-29)
# find total growth of real GDP over 30 years
genr gdpgro = (RGDPL - gdp60)/gdp60
# restrict the sample to a 1989 cross-section
smpl --restrict YEAR=1989
# convergence: did countries with a lower base grow faster?
ols gdpgro const gdp60
# result: No! Try an inverse relationship?
genr gdp60inv = 1/gdp60
ols gdpgro const gdp60inv
# no again.  Try treating Africa as special?
genr afdum = (CCODE = 1)
genr afslope = afdum * gdp60
ols gdpgro const afdum gdp60 afslope
```

Chapter 7

Sub-sampling a dataset

7.1 Introduction

Some subtle issues can arise here. This chapter attempts to explain the issues.

A sub-sample may be defined in relation to a full data set in two different ways: we will refer to these as "setting" the sample and "restricting" the sample respectively.

7.2 Setting the sample

By "setting" the sample we mean defining a sub-sample simply by means of adjusting the starting and/or ending point of the current sample range. This is likely to be most relevant for time-series data. For example, one has quarterly data from 1960:1 to 2003:4, and one wants to run a regression using only data from the 1970s. A suitable command is then

```
smpl 1970:1 1979:4
```

Or one wishes to set aside a block of observations at the end of the data period for out-of-sample forecasting. In that case one might do

```
smpl ; 2000:4
```

where the semicolon is shorthand for "leave the starting observation unchanged". (The semicolon may also be used in place of the second parameter, to mean that the ending observation should be unchanged.) By "unchanged" here, we mean unchanged relative to the last `smpl` setting, or relative to the full dataset if no sub-sample has been defined up to this point. For example, after

```
smpl 1970:1 2003:4
smpl ; 2000:4
```

the sample range will be 1970:1 to 2000:4.

An incremental or relative form of setting the sample range is also supported. In this case a relative offset should be given, in the form of a signed integer (or a semicolon to indicate no change), for both the starting and ending point. For example

```
smpl +1 ;
```

will advance the starting observation by one while preserving the ending observation, and

```
smpl +2 -1
```

will both advance the starting observation by two and retard the ending observation by one.

An important feature of "setting" the sample as described above is that it necessarily results in the selection of a subset of observations that are contiguous in the full dataset. The structure of the dataset is therefore unaffected (for example, if it is a quarterly time series before setting the sample, it remains a quarterly time series afterwards).

7.3 Restricting the sample

By "restricting" the sample we mean selecting observations on the basis of some Boolean (logical) criterion, or by means of a random number generator. This is likely to be most relevant for cross-sectional or panel data.

Suppose we have data on a cross-section of individuals, recording their gender, income and other characteristics. We wish to select for analysis only the women. If we have a `gender` dummy variable with value 1 for men and 0 for women we could do

```
smpl gender=0 --restrict
```

to this effect. Or suppose we want to restrict the sample to respondents with incomes over $50,000. Then we could use

```
smpl income>50000 --restrict
```

A question arises here. If we issue the two commands above in sequence, what do we end up with in our sub-sample: all cases with income over 50000, or just women with income over 50000? By default, in a gretl script, the answer is the latter: women with income over 50000. The second restriction augments the first, or in other words the final restriction is the logical product of the new restriction and any restriction that is already in place. If you want a new restriction to replace any existing restrictions you can first recreate the full dataset using

```
smpl --full
```

Alternatively, you can add the `replace` option to the `smpl` command:

```
smpl income>50000 --restrict --replace
```

This option has the effect of automatically re-establishing the full dataset before applying the new restriction.

Unlike a simple "setting" of the sample, "restricting" the sample may result in selection of non-contiguous observations from the full data set. It may also change the structure of the data set.

This can be seen in the case of panel data. Say we have a panel of five firms (indexed by the variable `firm`) observed in each of several years (identified by the variable `year`). Then the restriction

```
smpl year=1995 --restrict
```

produces a dataset that is not a panel, but a cross-section for the year 1995. Similarly

```
smpl firm=3 --restrict
```

produces a time-series dataset for firm number 3.

For these reasons (possible non-contiguity in the observations, possible change in the structure of the data), gretl acts differently when you "restrict" the sample as opposed to simply "setting" it. In the case of setting, the program merely records the starting and ending observations and uses these as parameters to the various commands calling for the estimation of models, the computation of statistics, and so on. In the case of restriction, the program makes a reduced copy of the dataset and by default treats this reduced copy as a simple, undated cross-section.[1]

If you wish to re-impose a time-series or panel interpretation of the reduced dataset you can do so using the `setobs` command, or the GUI menu item "Sample, Dataset structure".

[1] With one exception: if you start with a balanced panel dataset and the restriction is such that it preserves a balanced panel — for example, it results in the deletion of all the observations for one cross-sectional unit — then the reduced dataset is still, by default, treated as a panel.

The fact that "restricting" the sample results in the creation of a reduced copy of the original dataset may raise an issue when the dataset is very large (say, several thousands of observations). With such a dataset in memory, the creation of a copy may lead to a situation where the computer runs low on memory for calculating regression results. You can work around this as follows:

1. Open the full data set, and impose the sample restriction.
2. Save a copy of the reduced data set to disk.
3. Close the full dataset and open the reduced one.
4. Proceed with your analysis.

7.4 Random sampling

With very large datasets (or perhaps to study the properties of an estimator) you may wish to draw a random sample from the full dataset. This can be done using, for example,

```
smpl 100 --random
```

to select 100 cases. If you want the sample to be reproducible, you should set the seed for the random number generator first, using `set`. This sort of sampling falls under the "restriction" category: a reduced copy of the dataset is made.

7.5 The Sample menu items

The discussion above has focused on the script command `smpl`. You can also use the items under the Sample menu in the GUI program to select a sub-sample.

The menu items work in the same way as the corresponding `smpl` variants. When you use the item "Sample, Restrict based on criterion", and the dataset is already sub-sampled, you are given the option of preserving or replacing the current restriction. Replacing the current restriction means, in effect, invoking the `replace` option described above (Section 7.3).

Chapter 8

Graphs and plots

8.1 Gnuplot graphs

A separate program, gnuplot, is called to generate graphs. Gnuplot is a very full-featured graphing program with myriad options. It is available from www.gnuplot.info (but note that a copy of gnuplot is bundled with the MS Windows version of gretl). gretl gives you direct access, via a graphical interface, to a subset of gnuplot's options and it tries to choose sensible values for you; it also allows you to take complete control over graph details if you wish.

With a graph displayed, you can click on the graph window for a pop-up menu with the following options.

- Save as PNG: Save the graph in Portable Network Graphics format.
- Save as postscript: Save in encapsulated postscript (EPS) format.
- Save as Windows metafile: Save in Enhanced Metafile (EMF) format.
- Save to session as icon: The graph will appear in iconic form when you select "Icon view" from the Session menu.
- Zoom: Lets you select an area within the graph for closer inspection (not available for all graphs).
- Print: On the Gnome desktop only, lets you print the graph directly.
- Copy to clipboard: MS Windows only, lets you paste the graph into Windows applications such as MS Word.[1]
- Edit: Opens a controller for the plot which lets you adjust various aspects of its appearance.
- Close: Closes the graph window.

Displaying data labels

In the case of a simple X-Y scatterplot (with or without a line of best fit displayed), some further options are available if the dataset includes "case markers" (that is, labels identifying each observation).[2] With a scatter plot displayed, when you move the mouse pointer over a data point its label is shown on the graph. By default these labels are transient: they do not appear in the printed or copied version of the graph. They can be removed by selecting "Clear data labels" from the graph pop-up menu. If you want the labels to be affixed permanently (so they will show up when the graph is printed or copied), you have two options.

- To affix the labels currently shown on the graph, select "Freeze data labels" from the graph pop-up menu.

[1] For best results when pasting graphs into MS Office applications, choose the application's "Edit, Paste Special..." menu item, and select the option "Picture (Enhanced Metafile)".

[2] For an example of such a dataset, see the Ramanathan file `data4-10`: this contains data on private school enrollment for the 50 states of the USA plus Washington, DC; the case markers are the two-letter codes for the states.

- To affix labels for all points in the graph, select “Edit” from the graph pop-up and check the box titled “Show all data labels”. This option is available only if there are less than 55 data points, and it is unlikely to produce good results if the points are tightly clustered since the labels will tend to overlap.

To remove labels that have been affixed in either of these ways, select “Edit” from the graph pop-up and uncheck “Show all data labels”.

Advanced options

If you know something about gnuplot and wish to get finer control over the appearance of a graph than is available via the graphical controller (“Edit” option), you have two further options.

- Once the graph is saved as a session icon, you can right-click on its icon for a further pop-up menu. One of the otions here is “Edit plot commands”, which opens an editing window with the actual gnuplot commands displayed. You can edit these commands and either save them for future processing or send them to gnuplot (with the “File/Send to gnuplot” menu item in the plot commands editing window).
- Another way to save the plot commands (or to save the displayed plot in formats other than EPS or PNG) is to use “Edit” item on a graph’s pop-up menu to invoke the graphical controller, then click on the “Output to file” tab in the controller. You are then presented with a drop-down menu of formats in which to save the graph.

To find out more about gnuplot see the online manual or www.gnuplot.info.

See also the entry for `gnuplot` in the *Gretl Command Reference* — and the `graph` and `plot` commands for “quick and dirty” ASCII graphs.

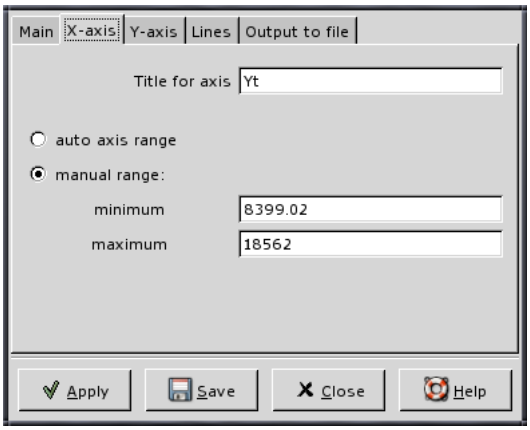


Figure 8.1: gretl’s gnuplot controller

8.2 Boxplots

Boxplots are not generated using gnuplot, but rather by gretl itself.

These plots (after Tukey and Chambers) display the distribution of a variable. The central box encloses the middle 50 percent of the data, i.e. it is bounded by the first and third quartiles. The “whiskers” extend to the minimum and maximum values. A line is drawn across the box at the median.

In the case of notched boxes, the notch shows the limits of an approximate 90 percent confidence interval. This is obtained by the bootstrap method, which can take a while if the data series is very long.

Clicking the mouse in the boxplots window brings up a menu which enables you to save the plots as encapsulated postscript (EPS) or as a full-page postscript file. Under the X window system you can also save the window as an XPM file; under MS Windows you can copy it to the clipboard as a bitmap. The menu also gives you the option of opening a summary window which displays five-number summaries (minimum, first quartile, median, third quartile, maximum), plus a confidence interval for the median if the "notched" option was chosen.

Some details of gretl's boxplots can be controlled via a (plain text) file named `.boxplotrc` which is looked for, in turn, in the current working directory, the user's home directory (corresponding to the environment variable HOME) and the gretl user directory (which is displayed and may be changed under the "File, Preferences, General" menu). Options that can be set in this way are the font to use when producing postscript output (must be a valid generic postscript font name; the default is Helvetica), the size of the font in points (also for postscript output; default is 12), the minimum and maximum for the y-axis range, the width and height of the plot in pixels (default, 560 x 448), whether numerical values should be printed for the quartiles and median (default, don't print them), and whether outliers (points lying beyond 1.5 times the interquartile range from the central box) should be indicated separately (default, no). Here is an example:

```
font = Times-Roman
fontsize = 16
max = 4.0
min = 0
width = 400
height = 448
numbers = %3.2f
outliers = true
```

On the second to last line, the value associated with `numbers` is a "printf" format string as in the C programming language; if specified, this controls the printing of the median and quartiles next to the boxplot, if no `numbers` entry is given these values are not printed. In the example, the values will be printed to a width of 3 digits, with 2 digits of precision following the decimal point.

Not all of the options need be specified, and the order doesn't matter. Lines not matching the pattern "key = value" are ignored, as are lines that begin with the hash mark, #.

After each variable specified in the boxplot command, a parenthesized boolean expression may be added, to limit the sample for the variable in question. A space must be inserted between the variable name or number and the expression. Suppose you have salary figures for men and women, and you have a dummy variable `GENDER` with value 1 for men and 0 for women. In that case you could draw comparative boxplots with the following line in the boxplots dialog:

```
salary (GENDER=1) salary (GENDER=0)
```

Chapter 9

Nonlinear least squares

9.1 Introduction and examples

As of version 1.0.9, gretl supports nonlinear least squares (NLS) using a variant of the Levenberg–Marquandt algorithm. The user must supply a specification of the regression function; prior to giving this specification the parameters to be estimated must be "declared" and given initial values. Optionally, the user may supply analytical derivatives of the regression function with respect to each of the parameters. The tolerance (criterion for terminating the iterative estimation procedure) can be set using the genr command.

The syntax for specifying the function to be estimated is the same as for the genr command. Here are two examples, with accompanying derivatives.

Example 9.1: Consumption function from Greene

```
nls C = alpha + beta * Y^gamma
deriv alpha = 1
deriv beta = Y^gamma
deriv gamma = beta * Y^gamma * log(Y)
end nls
```

Example 9.2: Nonlinear function from Russell Davidson

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

Note the command words nls (which introduces the regression function), deriv (which introduces the specification of a derivative), and end nls, which terminates the specification and calls for estimation. If the --vcv flag is appended to the last line the covariance matrix of the parameter estimates is printed.

9.2 Initializing the parameters

The parameters of the regression function must be given initial values prior to the nls command. This can be done using the genr command (or, in the GUI program, via the menu item "Define new variable").

In some cases, where the nonlinear function is a generalization of (or a restricted form of) a linear model, it may be convenient to run an ols and initialize the parameters from the OLS coefficient estimates. In relation to the first example above, one might do:

```
ols C 0 Y
genr alpha = $coeff(0)
genr beta = $coeff(Y)
genr gamma = 1
```

And in relation to the second example one might do:

```
ols y 0 x1 x2
genr alpha = $coeff(0)
genr beta = $coeff(x1)
```

9.3 NLS dialog window

It is probably most convenient to compose the commands for NLS estimation in the form of a gretl script but you can also do so interactively, by selecting the item "Nonlinear Least Squares" under the Model menu. This opens a dialog box where you can type the function specification (possibly prefaced by `genr` lines to set the initial parameter values) and the derivatives, if available. An example of this is shown in Figure 9.1. Note that in this context you do not have to supply the `nls` and `end nls` tags.

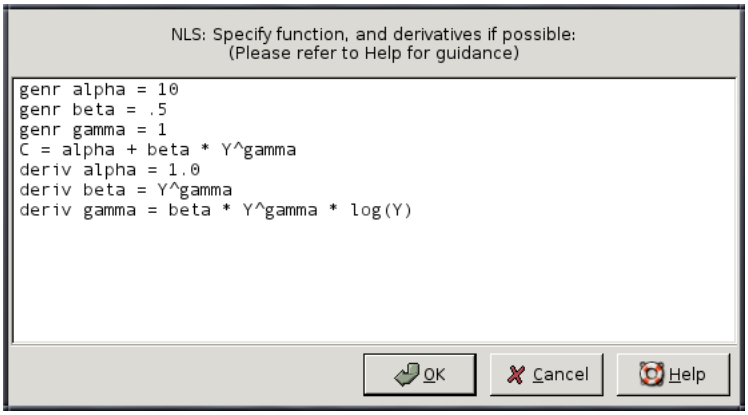


Figure 9.1: NLS dialog box

9.4 Analytical and numerical derivatives

If you are able to figure out the derivatives of the regression function with respect to the parameters, it is advisable to supply those derivatives as shown in the examples above. If that is not possible, gretl will compute approximate numerical derivatives. The properties of the NLS algorithm may not be so good in this case (see Section 9.7).

If analytical derivatives are supplied, they are checked for consistency with the given nonlinear function. If the derivatives are clearly incorrect estimation is aborted with an error message. If the derivatives are "suspicious" a warning message is issued but estimation proceeds. This warning may sometimes be triggered by incorrect derivatives, but it may also be triggered by a high degree of collinearity among the derivatives.

Note that you cannot mix analytical and numerical derivatives: you should supply expressions for all of the derivatives or none.

9.5 Controlling termination

The NLS estimation procedure is an iterative process. Iteration is terminated when the criterion for convergence is met or when the maximum number of iterations is reached, whichever comes first.

Let k denote the number of parameters being estimated. The maximum number of iterations is $100 \times (k + 1)$ when analytical derivatives are given, and $200 \times (k + 1)$ when numerical derivatives are used.

Let ϵ denote a small number. The iteration is deemed to have converged if one or both of the following conditions are satisfied:

- Both the actual and predicted relative reductions in the error sum of squares are at most ϵ.
- The relative error between two consecutive iterates is at most ϵ.

This default value of ϵ is the machine precision to the power 3/4,[1] but it can be adjusted using the `set` command with the parameter `nls_toler`. For example

```
set nls_toler .0001
```

will relax the value of ϵ to 0.0001.

9.6 Details on the code

The underlying engine for NLS estimation is based on the minpack suite of functions, available from netlib.org. Specifically, the following minpack functions are called:

`lmder`	Levenberg–Marquandt algorithm with analytical derivatives
`chkder`	Check the supplied analytical derivatives
`lmdif`	Levenberg–Marquandt algorithm with numerical derivatives
`fdjac2`	Compute final approximate Jacobian when using numerical derivatives
`dpmpar`	Determine the machine precision

On successful completion of the Levenberg–Marquandt iteration, a Gauss–Newton regression is used to calculate the covariance matrix for the parameter estimates. Since NLS results are asymptotic, there is room for debate over whether or not a correction for degrees of freedom should be applied when calculating the standard error of the regression (and the standard errors of the parameter estimates). For comparability with OLS, and in light of the reasoning given in Davidson and MacKinnon (1993), the estimates shown in gretl *do* use a degrees of freedom correction.

9.7 Numerical accuracy

Table 9.1 shows the results of running the gretl NLS procedure on the 27 Statistical Reference Datasets made available by the U.S. National Institute of Standards and Technology (NIST) for testing nonlinear regression software.[2] For each dataset, two sets of starting values for the parameters are given in the test files, so the full test comprises 54 runs. Two full tests were performed, one using all analytical derivatives and one using all numerical approximations. In each case the default tolerance was used.[3]

Out of the 54 runs, gretl failed to produce a solution in 4 cases when using analytical derivatives, and in 5 cases when using numeric approximation. Of the four failures in analytical derivatives mode, two were due to non-convergence of the Levenberg–Marquandt algorithm after the maximum number of iterations (on `MGH09` and `Bennett5`, both described by NIST as of "Higher difficulty") and two were due to generation of range errors (out-of-bounds floating point values) when computing the Jacobian (on `BoxBOD` and `MGH17`, described as of "Higher difficulty" and "Average difficulty" respectively). The additional failure in numerical approximation mode was on `MGH10` ("Higher difficulty", maximum number of iterations reached).

[1] On a 32-bit Intel Pentium machine a likely value for this parameter is 1.82×10^{-12}.

[2] For a discussion of gretl's accuracy in the estimation of linear models, see Appendix C.

[3] The data shown in the table were gathered from a pre-release build of gretl version 1.0.9, compiled with gcc 3.3, linked against glibc 2.3.2, and run under Linux on an i686 PC (IBM ThinkPad A21m).

The table gives information on several aspects of the tests: the number of outright failures, the average number of iterations taken to produce a solution and two sorts of measure of the accuracy of the estimates for both the parameters and the standard errors of the parameters.

For each of the 54 runs in each mode, if the run produced a solution the parameter estimates obtained by gretl were compared with the NIST certified values. We define the "minimum correct figures" for a given run as the number of significant figures to which the *least accurate* gretl estimate agreed with the certified value, for that run. The table shows both the average and the worst case value of this variable across all the runs that produced a solution. The same information is shown for the estimated standard errors.[4]

The second measure of accuracy shown is the percentage of cases, taking into account all parameters from all successful runs, in which the gretl estimate agreed with the certified value to at least the 6 significant figures which are printed by default in the gretl regression output.

Table 9.1: Nonlinear regression: the NIST tests

	Analytical derivatives	*Numerical derivatives*
Failures in 54 tests	4	5
Average iterations	32	127
Mean of min. correct figures, parameters	8.120	6.980
Worst of min. correct figures, parameters	4	3
Mean of min. correct figures, standard errors	8.000	5.673
Worst of min. correct figures, standard errors	5	2
Percent correct to at least 6 figures, parameters	96.5	91.9
Percent correct to at least 6 figures, standard errors	97.7	77.3

Using analytical derivatives, the worst case values for both parameters and standard errors were improved to 6 correct figures on the test machine when the tolerance was tightened to 1.0e−14. Using numerical derivatives, the same tightening of the tolerance raised the worst values to 5 correct figures for the parameters and 3 figures for standard errors, at a cost of one additional failure of convergence.

Note the overall superiority of analytical derivatives: on average solutions to the test problems were obtained with substantially fewer iterations and the results were more accurate (most notably for the estimated standard errors). Note also that the six-digit results printed by gretl are not 100 percent reliable for difficult nonlinear problems (in particular when using numerical derivatives). Having registered this caveat, the percentage of cases where the results were good to six digits or better seems high enough to justify their printing in this form.

[4] For the standard errors, I excluded one outlier from the statistics shown in the table, namely `Lanczos1`. This is an odd case, using generated data with an almost-exact fit: the standard errors are 9 or 10 orders of magnitude smaller than the coefficients. In this instance gretl could reproduce the certified standard errors to only 3 figures (analytical derivatives) and 2 figures (numerical derivatives).

Chapter 10

Maximum likelihood estimation

10.1 Generic ML estimation with gretl

Maximum likelihood estimation is a cornerstone of modern inferential procedures. Gretl provides a way to implement this method for a wide range of estimation problems, by use of the `mle` command. We give here a few examples.

To give a fiundation for the examples that follow, we start from a brief reminder on the basics of ML estimation. Given a sample of size T, it is possible to define the density function[1] for the whole sample, namely the joint distribution of all the observations $f(\mathbf{Y};\theta)$, where $\mathbf{Y} = \{y_1, \ldots, y_T\}$. Its shape is determined by a vector of unknown parameters θ, which we assume is contained in a set Θ, and which can be used to evaluate the probability of observing a sample with any given characteristic.

After observing the data, the values $\mathbf{Y}$ are given, and this function can be evaluated for any legitimate value of θ. In this case, we prefer to call it the *likelihood* function; the need for another name stems from the fact that this function works as a density when we use the y_t's as arguments and θ as parameters, whereas in this context θ is taken as the function's argument, and the data $\mathbf{Y}$ only have the role of determining its shape.

In standard cases, this function has a unique maximum. The location of the maximum is unaffected if we consider the logarithm of the likelihood (or log-likelihood for short): this function will be denoted as

$$\ell(\theta) = \log f(\mathbf{Y};\theta).$$

The log-likelihood functions that gretl can handle are those when $\ell(\theta)$ can be written as

$$\ell(\theta) = \sum_{t=1}^{T} \ell_t(\theta),$$

which is true in most cases of interest. The functions $\ell_t(\theta)$ are called the log-likelihood contributions.

Moreover, the location of the maximum is obviously determined by the data $\mathbf{Y}$. This means that the value

$$\hat{\theta}(\mathbf{Y}) = \operatorname*{Argmax}_{\theta\in\Theta} \ell(\theta) \tag{10.1}$$

is some function of the observed data (a statistic), which has the property, under mild conditions, of being a consistent, asymptotically normal and asymptotically efficient estimator of θ.

Sometimes it is possible to write down explicitly the function $\hat{\theta}(\mathbf{Y})$; in general, it need not be so. In these circumstances, the maximum can be found by means of numerical techniques. These often rely on the fact that the log-likelihood is a smooth function of θ, and therefore on the maximum its partial derivatives should all be 0. The gradient of the log-likelihood is called the *score* vector and is a function that enjoys many interesting statistical properties in its own right; it will be denoted here as $\mathbf{s}(\theta)$.

Gradient-based methods can be shortly illustrated as follows:

[1] We are imagining here that our data are a realisation of continuous random variables. For discrete random variable, everything continues to apply by referring to the probability function instead of the density. In both cases, the distribution may be conditional on some exogenous variables.

1. pick a point $\theta_0 \in \Theta$;
2. evaluate $\mathbf{s}(\theta_0)$;
3. if $\mathbf{s}(\theta_0)$ is "small", stop. Otherwise, compute a direction vector $d(\mathbf{s}(\theta_0))$;
4. evaluate $\theta_1 = \theta_0 + d(\mathbf{s}(\theta_0))$;
5. substitute θ_0 with θ_1 ;
6. restart from 2.

Many algorithms of this kind exist; they basically differ from one another in the way they compute the direction vector $d(\mathbf{s}(\theta_0))$, to ensure that $\ell(\theta_1) > \ell(\theta_0)$ (so that we eventually end up on the maximum).

The method gretl uses to maximize the log-likelihood is a gradient-based algorithm known as the **BFGS** (Broyden, Fletcher, Goldfarb and Shanno) method. This technique is used in most econometric and statistical packages, as is well-established and remarkably powerful. Clearly, in order to make this technique operational, it must be possible to compute the vector $\mathbf{s}(\theta)$ for any value of θ. In some cases, the function $\mathbf{s}(\theta)$ can be written explicitly in term of $\mathbf{Y}$. At times, this is not possible or too difficult; therefore, the function $\mathbf{s}(\theta)$ is evaluated numerically.

The choice of the starting value θ_0 is crucial in some contexts and inconsequential in others. In general, however, it is advisable to start the algorithm from "sensible" values whenever possible. If a consistent estimator is available, this is usually a safe and efficient choice: this ensures that in large samples the starting point will be likely close to $\hat{\theta}$ and convergence can be achieved in few iterations.

10.2 Gamma estimation

Suppose we have a sample of T independent and identically distributed observations from a Gamma distribution. The density function for each observation x_t is

$$f(x_t) = \frac{\alpha^p}{\Gamma(p)} x_t^{p-1} \exp(-\alpha x_t). \tag{10.2}$$

The log-likelihood for the entire sample can be written as the logarithm of the joint density of all the observations. Since these are independent and identical, the joint density is the product of the individual densities, and hence its log is

$$\ell(\alpha, p) = \sum_{t=1}^{T} \log \left[\frac{\alpha^p}{\Gamma(p)} x_t^{p-1} \exp(-\alpha x_t) \right] = \sum_{t=1}^{T} \ell_t, \tag{10.3}$$

where

$$\ell_t = p \cdot \log(\alpha x_t) - \gamma(p) - \log x_t - \alpha x_t$$

and $\gamma(\cdot)$ is the log of the gamma function. In order to estimate the parameters α and p via ML, we need to maximize (10.3) with respect to them. The corresponding gretl code snippet is

```
scalar alpha = 1
scalar p = 1

mle logl =  p*ln(alpha * x) - lngamma(p) - ln(x) - alpha * x
end mle
```

The two statements

```
alpha = 1
p = 1
```

are necessary to ensure that the variables p and alpha exist before the computation of logl is attempted. The values of these variables will be changed by the execution of the mle command; upon successful completion, they will be replaced by the ML estimates. The starting value is 1 for both; this is arbitrary and does not matter much in this example (more on this later).

The above code can be made more readable, and marginally more efficient, by defining a variable to hold $\alpha \cdot x_t$. This command can be embedded into the mle block as follows:

```
scalar alpha = 1
scalar p = 1

mle logl =  p*ln(ax) - lngamma(p) - ln(x) - ax
series ax = alpha*x
params alpha p
end mle
```

In this case, it is necessary to include the line params alpha p to set the symbols p and alpha apart from ax, which is a temporarily generated variable and not a parameter to be estimated.

In a simple example like this, the choice of the starting values is almost inconsequential; the algorithm is likely to converge no matter what the starting values are. However, consistent method-of-moments estimators of p and α can be simply recovered from the sample mean m and variance V: since it can be shown that

$$E(x_t) = p/\alpha \qquad V(x_t) = p/\alpha^2,$$

it follows that the following estimators

$$\begin{aligned} \tilde{\alpha} &= m/V \\ \tilde{p} &= m \cdot \tilde{\alpha} \end{aligned}$$

are consistent, and therefore suitable to be used as starting point for the algorithm. The gretl script code then becomes

```
scalar m = mean(x)
scalar alpha = var(x)/m
scalar p = m*alpha

mle logl =  p*ln(ax) - lngamma(p) - ln(x) - ax
series ax = alpha*x
params alpha p
end mle
```

10.3 Stochastic frontier cost function

When modeling a cost function, it is sometimes worthwhile to incorporate explicitly into the statistical model the notion that firms may be inefficient, so that the observed cost deviates from the theoretical figure not only because of unobserved heterogeneity between firms, but also because two firms could be operating at a different efficiency level, despite being identical under all other respects. In this case we may write

$$C_i = C_i^* + u_i + v_i,$$

where C_i is some variable cost indicator, C_i^* is its "theoretical" value, u_i is a zero-mean disturbance term and v_i is the inefficiency term, which is supposed to be nonnegative by its very nature.

A linear specification for C_i^* is often chosen. For example, the Cobb–Douglas cost function arises when C_i^* is a linear function of the logarithms of the input prices and the output quantities.

The *stochastic frontier* model is a linear model of the form $y_i = x_i\beta + \varepsilon_i$ in which the error term ε_i is the sum of u_i and v_i. A common postulate is that $u_i \sim N(0, \sigma_u^2)$ and $v_i \sim |N(0, \sigma_v^2)|$. If

independence between u_i and v_i is also assumed, then it is possible to show that the density function of ε_i has the form:

$$f(\varepsilon_i) = \sqrt{\frac{2}{\pi}}\Phi\left(\frac{\lambda\varepsilon_i}{\sigma}\right)\frac{1}{\sigma}\phi\left(\frac{\varepsilon_i}{\sigma}\right), \tag{10.4}$$

where $\Phi(\cdot)$ and $\phi(\cdot)$ are, respectively, the distribution and density function of the standard normal, $\sigma = \sqrt{\sigma_u^2 + \sigma_v^2}$ and $\lambda = \frac{\sigma_u}{\sigma_v}$.

As a consequence, the log-likelihood for one observation takes the form (apart form an irrelevant constant)

$$\ell_t = \log\Phi\left(\frac{\lambda\varepsilon_i}{\sigma}\right) - \left[\log(\sigma) + \frac{\varepsilon_i^2}{2\sigma^2}\right];$$

therefore, a Cobb–Douglas cost function with stochastic frontier is the model described by the following equations:

$$\begin{aligned}
\log C_i &= \log C_i^* + \varepsilon_i \\
\log C_i^* &= c + \sum_{j=1}^{m}\beta_j \log y_{ij} + \sum_{j=1}^{n}\alpha_j \log p_{ij} \\
\varepsilon_i &= u_i + v_i \\
u_i &\sim N(0, \sigma_u^2) \\
v_i &\sim \left|N(0, \sigma_v^2)\right|.
\end{aligned}$$

In most cases, one wants to ensure that the homogeneity of the cost function with respect to the prices holds by construction. Since this requirement is equivalent to $\sum_{j=1}^{n}\alpha_j = 1$, the above equation for C_i^* can be rewritten as

$$\log C_i - \log p_{in} = c + \sum_{j=1}^{m}\beta_j \log y_{ij} + \sum_{j=2}^{n}\alpha_j(\log p_{ij} - \log p_{in}) + \varepsilon_i. \tag{10.5}$$

The above equation could be estimated by OLS, but it would suffer from two drawbacks: first, the OLS estimator for the intercept c is inconsistent because the disturbance term has a non-zero expected value; second, the OLS estimators for the other parameters are consistent, but inefficient in view of the non-normality of ε_i. Both issues can be addressed by estimating (10.5) by maximum likelihood. Nevertheless, OLS estimation is a quick and convenient way to provide starting values for the MLE algorithm.

Example 10.1 shows how to implement the model described so far. The `banks91` file contains part of the data used in Lucchetti, Papi and Zazzaro (2001).

10.4 GARCH models

GARCH models are handled by gretl via a native function. However, it is instructive to see how they can be estimated through the `mle` command.

The following equations provide the simplest example of a GARCH(1,1) model:

$$\begin{aligned}
y_t &= \mu + \varepsilon_t \\
\varepsilon_t &= u_t \cdot \sigma_t \\
u_t &\sim N(0,1) \\
h_t &= \omega + \alpha\varepsilon_{t-1}^2 + \beta h_{t-1}.
\end{aligned}$$

Since the variance of y_t depends on past values, writing down the log-likelihood function is not simply a matter of summing the log densities for individual observations. As is common in time series models, y_t cannot be considered independent of the other observations in our sample, and

Example 10.1: Estimation of stochastic frontier cost function

```
open banks91

# Cobb-Douglas cost function

ols cost const y p1 p2 p3

# Cobb-Douglas cost function with homogeneity restrictions

genr rcost = cost - p3
genr rp1 = p1 - p3
genr rp2 = p2 - p3

ols rcost const y rp1 rp2

# Cobb-Douglas cost function with homogeneity restrictions
# and inefficiency

scalar b0 = $coeff(const)
scalar b1 = $coeff(y)
scalar b2 = $coeff(rp1)
scalar b3 = $coeff(rp2)

scalar su = 0.1
scalar sv = 0.1

mle logl = ln(cnorm(e*lambda/ss)) - (ln(ss) + 0.5*(e/ss)^2)
  scalar ss = sqrt(su^2 + sv^2)
  scalar lambda = su/sv
  series e = rcost - b0*const - b1*y - b2*rp1 - b3*rp2
  params b0 b1 b2 b3 su sv
end mle
```

consequently the density function for the whole sample (the joint density for all observations) is not just the product of the marginal densities.

Maximum likelihood estimation, in these cases, is achieved by considering *conditional* densities, so what we maximize is a conditional likelihood function. If we define the information set at time t as

$$F_t = \{y_t, y_{t-1}, \ldots\},$$

then the density of y_t conditional on F_{t-1} is normal:

$$y_t | F_{t-1} \sim N[\mu, h_t].$$

By means of the properties of conditional distributions, the joint density can be factorized as follows

$$f(y_t, y_{t-1}, \ldots) = \left[\prod_{t=1}^{T} f(y_t | F_{t-1})\right] \cdot f(y_0);$$

if we treat y_0 as fixed, then the term $f(y_0)$ does not depend on the unknown parameters, and therefore the conditional log-likelihood can then be written as the sum of the individual contributions as

$$\ell(\mu, \omega, \alpha, \beta) = \sum_{t=1}^{T} \ell_t, \tag{10.6}$$

where

$$\ell_t = \log\left[\frac{1}{\sqrt{h_t}}\phi\left(\frac{y_t - \mu}{\sqrt{h_t}}\right)\right] = -\frac{1}{2}\left[\log(h_t) + \frac{(y_t - \mu)^2}{h_t}\right].$$

The following script shows a simple application of this technique, which uses the data file `djclose`; it is one of the example dataset supplied with gretl and contains daily data from the Dow Jones stock index.

```
open djclose

series y = 100*ldiff(djclose)

scalar mu = 0.0
scalar omega = 1
scalar alpha = 0.4
scalar beta = 0.0

mle ll = -0.5*(log(h) + (e^2)/h)
  series e = y - mu
  series h = var(y)
  series h = omega + alpha*(e(-1))^2 + beta*h(-1)
  params mu omega alpha beta
end mle
```

10.5 Analytical derivatives

Computation of the score vector is essential for the working of the BFGS method. In all the previous examples, no explicit formula for the computation of the score was given, so the algorithm was fed numerically evaluated gradients. Numerical computation of the score for the i-th parameter is performed via a finite approximation of the derivative, namely

$$\frac{\partial \ell(\theta_1, \ldots, \theta_n)}{\partial \theta_i} \simeq \frac{\ell(\theta_1, \ldots, \theta_i + h, \ldots, \theta_n) - \ell(\theta_1, \ldots, \theta_i - h, \ldots, \theta_n)}{2h},$$

where h is a small number.

In many situations, this is rather efficient and accurate. However, one might want to avoid the approximation and specify an exact function for the derivatives. As an example, consider the following script:

```
nulldata 1000

genr x1 = normal()
genr x2 = normal()
genr x3 = normal()

genr ystar = x1 + x2 + x3 + normal()
genr y = (ystar > 0)

scalar b0 = 0
scalar b1 = 0
scalar b2 = 0
scalar b3 = 0

mle logl = y*ln(P) + (1-y)*ln(1-P)
  series ndx = b0 + b1*x1 + b2*x2 + b3*x3
  series P = cnorm(ndx)
  params b0 b1 b2 b3
end mle --verbose
```

Here, 1000 data points are artificially generated for an ordinary probit model[2]: y_t is a binary variable, which takes the value 1 if $y_t^* = \beta_1 x_{1t} + \beta_2 x_{2t} + \beta_3 x_{3t} + \varepsilon_t > 0$ and 0 otherwise. Therefore, $y_t = 1$ with probability $\Phi(\beta_1 x_{1t} + \beta_2 x_{2t} + \beta_3 x_{3t}) = \pi_t$. The probability function for one observation can be written as

$$P(y_t) = \pi_t^{y_t}(1 - \pi_t)^{1-y_t};$$

since the observations are independent and identically distributed, the log-likelihood is simply the sum of the individual contributions. Hence

$$\ell = \sum_{t=1}^{T} y_t \log(\pi_t) + (1 - y_t)\log(1 - \pi_t).$$

The `--verbose` switch at the end of the `end mle` statement produces a detailed account of the iterations done by the BFGS algorithm.

In this case, numerical differentiation works rather well; nevertheless, computation of the analytical score is straightforward, since the derivative $\frac{\partial \ell}{\partial \beta_i}$ can be written as

$$\frac{\partial \ell}{\partial \beta_i} = \frac{\partial \ell}{\partial \pi_t} \cdot \frac{\partial \pi_t}{\partial \beta_i}$$

via the chain rule, and it is easy to see that

$$\begin{aligned} \frac{\partial \ell}{\partial \pi_t} &= \frac{y_t}{\pi_t} - \frac{1 - y_t}{1 - \pi_t} \\ \frac{\partial \pi_t}{\partial \beta_i} &= \phi(\beta_1 x_{1t} + \beta_2 x_{2t} + \beta_3 x_{3t}) \cdot x_{it}. \end{aligned}$$

The `mle` block in the above script can therefore be modified as follows:

```
mle logl = y*ln(P) + (1-y)*ln(1-P)
  series ndx = b0 + b1*x1 + b2*x2 + b3*x3
  series P = cnorm(ndx)
```

[2] Again, gretl does provide a native `probit` command, but a probit model makes for a nice example here.

```
  series tmp = dnorm(ndx)*(y/P - (1-y)/(1-P))
  deriv b0 = tmp
  deriv b1 = tmp*x1
  deriv b2 = tmp*x2
  deriv b3 = tmp*x3
end mle --verbose
```

Note that the `params` statement has been replaced by a series of `deriv` statements; these have the double function of identifying the parameters over which to optimize and providing an analytical expression for their respective score elements.

Chapter 11

Model selection criteria

11.1 Introduction

In some contexts the econometrician chooses between alternative models based on a formal hypothesis test. For example, one might choose a more general model over a more restricted one if the restriction in question can be formulated as a testable null hypothesis, and the null is rejected on an appropriate test.

In other contexts one sometimes seeks a criterion for model selection that somehow measures the balance between goodness of fit or likelihood, on the one hand, and parsimony on the other. The balancing is necessary because the addition of extra variables to a model cannot reduce the degree of fit or likelihood, and is very likely to increase it somewhat even if the additional variables are not truly relevant to the data-generating process.

The best known such criterion, for linear models estimated via least squares, is the adjusted R^2,

$$\bar{R}^2 = 1 - \frac{\text{SSR}/(n-k)}{\text{TSS}/(n-1)}$$

where n is the number of observations in the sample, k denotes the number of parameters estimated, and SSR and TSS denote the sum of squared residuals and the total sum of squares for the dependent variable, respectively. Compared to the ordinary coefficient of determination or unadjusted R^2,

$$R^2 = 1 - \frac{\text{SSR}}{\text{TSS}}$$

the "adjusted" calculation penalizes the inclusion of additional parameters, other things equal.

11.2 Information criteria

A more general criterion in a similar spirit is Akaike's (1974) "Information Criterion" (AIC). The original formulation of this measure is

$$\text{AIC} = -2\ell(\hat{\theta}) + 2k \tag{11.1}$$

where $\ell(\hat{\theta})$ represents the maximum loglikelihood as a function of the vector of parameter estimates, $\hat{\theta}$, and k (as above) denotes the number of "independently adjusted parameters within the model." In this formulation, with AIC negatively related to the likelihood and positively related to the number of parameters, the researcher seeks the minimum AIC.

The AIC can be confusing, in that several variants of the calculation are "in circulation." For example, Davidson and MacKinnon (2004) present a simplified version,

$$\text{AIC} = \ell(\hat{\theta}) - k$$

which is just -2 times the original: in this case, obviously, one wants to maximize AIC.

In the case of models estimated by least squares, the loglikelihood can be written as

$$\ell(\hat{\theta}) = -\frac{n}{2}(1 + \log 2\pi - \log n) - \frac{n}{2}\log \text{SSR} \tag{11.2}$$

Substituting (11.2) into (11.1) we get

$$\text{AIC} = n(1 + \log 2\pi - \log n) + n \log \text{SSR} + 2k$$

which can also be written as

$$\text{AIC} = n \log \left(\frac{\text{SSR}}{n} \right) + 2k + n(1 + \log 2\pi) \tag{11.3}$$

Some authors simplify the formula for the case of models estimated via least squares. For instance, William Greene writes

$$\text{AIC} = \log \left(\frac{\text{SSR}}{n} \right) + \frac{2k}{n} \tag{11.4}$$

This variant can be derived from (11.3) by dividing through by n and subtracting the constant $1 + \log 2\pi$. That is, writing AIC_G for the version given by Greene, we have

$$\text{AIC}_G = \frac{1}{n}\text{AIC} - (1 + \log 2\pi)$$

Finally, Ramanathan gives a further variant:

$$\text{AIC}_R = \left(\frac{\text{SSR}}{n} \right) e^{2k/n}$$

which is the exponential of the one given by Greene.

Gretl began by using the Ramanathan variant, but since version 1.3.1 the program has used the original Akaike formula (11.1), and more specifically (11.3) for models estimated via least squares.

Although the Akaike criterion is designed to favor parsimony, arguably it does not go far enough in that direction. For instance, if we have two nested models with k and $k+1$ parameters respectively, and if the null hypothesis that parameter $k + 1$ equals 0 is true, in large samples the AIC will nonetheless tend to select the less parsimonious model about 16 percent of the time (see Davidson and MacKinnon, 2004, chapter 15).

An alternative to the AIC which avoids this problem is the Schwarz (1978) "Bayesian information criterion" (BIC). The BIC can be written (in line with Akaike's formulation of the AIC) as

$$\text{BIC} = -2\ell(\hat{\theta}) + k \log n$$

The multiplication of k by $\log n$ in the BIC means that the penalty for adding extra parameters grows with the sample size. This ensures that, asymptotically, one will not select a larger model over a correctly specified parsimonious model.

A further alternative to AIC, which again tends to select more parsimonious models than AIC, is the Hannan–Quinn criterion or HQC (Hannan and Quinn, 1979). Written consistently with the formulations above, this is

$$\text{HQC} = -2\ell(\hat{\theta}) + 2k \log \log n$$

The Hannan–Quinn calculation is based on the law of the iterated logarithm (note that the last term is the log of the log of the sample size). The authors argue that their procedure provides a "strongly consistent estimation procedure for the order of an autoregression", and that "compared to other strongly consistent procedures this procedure will underestimate the order to a lesser degree."

Gretl reports the AIC, BIC and HQC (calculated as explained above) for most sorts of models. The key point is interpreting these values is to know whether they are calculated such that smaller values are better, or such that larger values are better. In gretl, smaller values are better: one wants to minimize the chosen criterion.

Chapter 12

Loop constructs

12.1 Introduction

The command `loop` opens a special mode in which gretl accepts a block of commands to be repeated one or more times. This feature is designed for use with Monte Carlo simulations, bootstrapping of test statistics, and iterative estimation procedures. The general form of a loop is:

```
loop control-expression [ --progressive | --verbose | --quiet ]
   loop body
endloop
```

Five forms of control-expression are available, as explained below. In the loop body the following commands are accepted: `genr`, `ols`, `print`, `printf`, `pvalue`, `sim`, `smpl`, `store`, `summary`, `if`, `else` and `endif`.

By default, the `genr` command operates quietly in the context of a loop (without printing information on the variable generated). To force the printing of feedback from `genr` you may specify the `--verbose` option to `loop`. The `--quiet` option suppresses the usual printout of the number of iterations performed, which may be desirable when loops are nested.

The `--progressive` option to `loop` modifies the behavior of the commands `ols`, `print` and `store` in a manner that may be useful with Monte Carlo analyses (see Section 12.3).

The following sections explain the various forms of the loop control expression and provide some examples of use of loops.

☞ If you are carrying out a substantial Monte Carlo analysis with many thousands of repetitions, memory capacity and processing time may be an issue. To minimize the use of computer resources, run your script using the command-line program, gretlcli, with output redirected to a file.

12.2 Loop control variants

Count loop

The simplest form of loop control is a direct specification of the number of times the loop should be repeated. We refer to this as a "count loop". The number of repetitions may be a numerical constant, as in `loop 1000`, or may be read from a variable, as in `loop replics`.

In the case where the loop count is given by a variable, say `replics`, in concept `replics` is an integer scalar. If it is in fact a series, its first value is read. If the value is not integral, it is converted to an integer by truncation. Note that `replics` is evaluated only once, when the loop is initially compiled.

While loop

A second sort of control expression takes the form of the keyword `while` followed by an inequality: the left-hand term should be the name of a predefined variable; the right-hand side may be either a numerical constant or the name of another predefined variable. For example,

```
loop while essdiff > .00001
```

Execution of the commands within the loop will continue so long as the specified condition evaluates as true. If the right-hand term of the inequality is a variable, it is evaluated at the top of the loop at each iteration.

Index loop

A third form of loop control uses the special internal index variable `i`. In this case you specify starting and ending values for `i`, which is incremented by one each time round the loop. The syntax looks like this: `loop i=1..20`.

The index variable may be used within the loop body in one or both of two ways: you can access the value of `i` (see Example 12.4) or you can use its string representation, `$i` (see Example 12.5).

The starting and ending values for the index can be given in numerical form, or by reference to predefined variables. In the latter case the variables are evaluated once, when the loop is set up. In addition, with time series data you can give the starting and ending values in the form of dates, as in `loop i=1950:1..1999:4`.

For each loop

The fourth form of loop control also uses the internal variable `i`, but in this case the variable ranges over a specified list of strings. The loop is executed once for each string in the list. This can be useful for performing repetitive operations on a list of variables. Here is an example of the syntax:

```
loop foreach i peach pear plum
   print "$i"
endloop
```

This loop will execute three times, printing out "peach", "pear" and "plum" on the respective iterations.

If you wish to loop across a list of variables that are contiguous in the dataset, you can give the names of the first and last variables in the list, separated by "`..`", rather than having to type all the names. For example, say we have 50 variables `AK`, `AL`, ..., `WY`, containing income levels for the states of the US. To run a regression of income on time for each of the states we could do:

```
genr time
loop foreach i AL..WY
   ols $i const time
endloop
```

For loop

The final form of loop control uses a simplified version of the `for` statement in the C programming language. The expression is composed of three parts, separated by semicolons. The first part specifies an initial condition, expressed in terms of a control variable; the second part gives a continuation condition (in terms of the same control variable); and the third part specifies an increment (or decrement) for the control variable, to be applied each time round the loop. The entire expression is enclosed in parentheses. For example:

```
loop for (r=0.01; r<.991; r+=.01)
```

In this example the variable `r` will take on the values 0.01, 0.02, ..., 0.99 across the 99 iterations. Note that due to the finite precision of floating point arithmetic on computers it may be necessary to use a continuation condition such as the above, `r<.991`, rather than the more "natural" `r<=.99`. (Using double-precision numbers on an x86 processor, at the point where you would expect `r` to equal 0.99 it may in fact have value 0.990000000000001.)

To expand on the rules for the three components of the control expression:

1. The initial condition must take the form LHS1 = RHS1. RHS1 must be a numeric constant or a predefined variable. If the LHS1 variable does not exist already, it is automatically created.
2. The continuation condition must be of the form LHS1 op RHS2, where op can be <, >, <= or >= and RHS2 must be a numeric constant or a predefined variable. If RHS2 is a variable it is evaluated each time round the loop.
3. The increment or decrement expression must be of the form LHS1 += DELTA or LHS1 -= DELTA, where DELTA is a numeric constant or a predefined variable. If DELTA is a variable, it is evaluated only once, when the loop is set up.

12.3 Progressive mode

If the `--progressive` option is given for a command loop, the effects of the commands `ols`, `print` and `store` are modified as follows.

`ols`: The results from each individual iteration of the regression are not printed. Instead, after the loop is completed you get a printout of (a) the mean value of each estimated coefficient across all the repetitions, (b) the standard deviation of those coefficient estimates, (c) the mean value of the estimated standard error for each coefficient, and (d) the standard deviation of the estimated standard errors. This makes sense only if there is some random input at each step.

`print`: When this command is used to print the value of a variable, you do not get a print each time round the loop. Instead, when the loop is terminated you get a printout of the mean and standard deviation of the variable, across the repetitions of the loop. This mode is intended for use with variables that have a single value at each iteration, for example the error sum of squares from a regression.

`store`: This command writes out the values of the specified variables, from each time round the loop, to a specified file. Thus it keeps a complete record of the variables across the iterations. For example, coefficient estimates could be saved in this way so as to permit subsequent examination of their frequency distribution. Only one such `store` can be used in a given loop.

12.4 Loop examples

Monte Carlo example

A simple example of a Monte Carlo loop in "progressive" mode is shown in Example 12.1.

This loop will print out summary statistics for the 'a' and 'b' estimates and R^2 across the 100 repetitions. After running the loop, `coeffs.gdt`, which contains the individual coefficient estimates from all the runs, can be opened in gretl to examine the frequency distribution of the estimates in detail.

The command `nulldata` is useful for Monte Carlo work. Instead of opening a "real" data set, `nulldata 50` (for instance) opens a dummy data set, containing just a constant and an index variable, with a series length of 50. Constructed variables can then be added using the `genr` command.See the `set` command for information on generating repeatable pseudo-random series.

Iterated least squares

Example 12.2 uses a "while" loop to replicate the estimation of a nonlinear consumption function of the form

$$C = \alpha + \beta Y^{\gamma} + \epsilon$$

as presented in Greene (2000, Example 11.3). This script is included in the gretl distribution under the name `greene11_3.inp`; you can find it in gretl under the menu item "File, Open command file, practice file, Greene...".

Example 12.1: Simple Monte Carlo loop

```
nulldata 50
seed 547
genr x = 100 * uniform()
# open a "progressive" loop, to be repeated 100 times
loop 100 --progressive
   genr u = 10 * normal()
   # construct the dependent variable
   genr y = 10*x + u
   # run OLS regression
   ols y const x
   # grab the coefficient estimates and R-squared
   genr a = $coeff(const)
   genr b = $coeff(x)
   genr r2 = $rsq
   # arrange for printing of stats on these
   print a b r2
   # and save the coefficients to file
   store coeffs.gdt a b
endloop
```

The option `--print-final` for the `ols` command arranges matters so that the regression results will not be printed each time round the loop, but the results from the regression on the last iteration will be printed when the loop terminates.

Example 12.3 shows how a loop can be used to estimate an ARMA model, exploiting the "outer product of the gradient" (OPG) regression discussed by Davidson and MacKinnon in their *Estimation and Inference in Econometrics*.

Indexed loop examples

Example 12.4 shows an indexed loop in which the `smpl` is keyed to the index variable `i`. Suppose we have a panel dataset with observations on a number of hospitals for the years 1991 to 2000 (where the year of the observation is indicated by a variable named `year`). We restrict the sample to each of these years in turn and print cross-sectional summary statistics for variables 1 through 4.

Example 12.5 illustrates string substitution in an indexed loop.

The first time round this loop the variable `V` will be set to equal `COMP1987` and the dependent variable for the `ols` will be `PBT1987`. The next time round `V` will be redefined as equal to `COMP1988` and the dependent variable in the regression will be `PBT1988`. And so on.

Example 12.2: Nonlinear consumption function

```
open greene11_3.gdt
# run initial OLS
ols C 0 Y
genr essbak = $ess
genr essdiff = 1
genr beta = $coeff(Y)
genr gamma = 1
# iterate OLS till the error sum of squares converges
loop while essdiff > .00001
   # form the linearized variables
   genr C0 = C + gamma * beta * Y^gamma * log(Y)
   genr x1 = Y^gamma
   genr x2 = beta * Y^gamma * log(Y)
   # run OLS
   ols C0 0 x1 x2 --print-final --no-df-corr --vcv
   genr beta = $coeff(x1)
   genr gamma = $coeff(x2)
   genr ess = $ess
   genr essdiff = abs(ess - essbak)/essbak
   genr essbak = ess
endloop
# print parameter estimates using their "proper names"
noecho
printf "alpha = %g\n", $coeff(0)
printf "beta  = %g\n", beta
printf "gamma = %g\n", gamma
```

Example 12.3: ARMA 1, 1

```
open armaloop.gdt

genr c = 0
genr a = 0.1
genr m = 0.1

series e = 1.0
genr de_c = e
genr de_a = e
genr de_m = e

genr crit = 1
loop while crit > 1.0e-9

   # one-step forecast errors
   genr e = y - c - a*y(-1) - m*e(-1)

   # log-likelihood
   genr loglik = -0.5 * sum(e^2)
   print loglik

   # partials of forecast errors wrt c, a, and m
   genr de_c = -1 - m * de_c(-1)
   genr de_a = -y(-1) -m * de_a(-1)
   genr de_m = -e(-1) -m * de_m(-1)

   # partials of l wrt c, a and m
   genr sc_c = -de_c * e
   genr sc_a = -de_a * e
   genr sc_m = -de_m * e

   # OPG regression
   ols const sc_c sc_a sc_m --print-final --no-df-corr --vcv

   # Update the parameters
   genr dc = $coeff(sc_c)
   genr c = c + dc
   genr da = $coeff(sc_a)
   genr a = a + da
   genr dm = $coeff(sc_m)
   genr m = m + dm

   printf "  constant        = %.8g (gradient = %#.6g)\n", c, dc
   printf "  ar1 coefficient = %.8g (gradient = %#.6g)\n", a, da
   printf "  ma1 coefficient = %.8g (gradient = %#.6g)\n", m, dm

   genr crit = $T - $ess
   print crit
endloop

genr se_c = $stderr(sc_c)
genr se_a = $stderr(sc_a)
genr se_m = $stderr(sc_m)

noecho
print "
printf "constant = %.8g (se = %#.6g, t = %.4f)\n", c, se_c, c/se_c
printf "ar1 term = %.8g (se = %#.6g, t = %.4f)\n", a, se_a, a/se_a
printf "ma1 term = %.8g (se = %#.6g, t = %.4f)\n", m, se_m, m/se_m
```

Example 12.4: Panel statistics

```
open hospitals.gdt
loop i=1991..2000
  smpl (year=i) --restrict --replace
  summary 1 2 3 4
endloop
```

Example 12.5: String substitution

```
open bea.dat
loop i=1987..2001
  genr V = COMP$i
  genr TC = GOC$i - PBT$i
  genr C = TC - V
  ols PBT$i const TC V
endloop
```

Chapter 13

User-defined functions

13.1 Introduction

As of version 1.4.0, gretl contains a revised mechanism for defining functions in the context of a script. Details follow.[1]

13.2 Defining a function

Functions must be defined before they are called. The syntax for defining a function looks like this

```
function function-name parameters
   function body
end function
```

function-name is the unique identifier for the function. Names must start with a letter. They have a maximum length of 31 characters; if you type a longer name it will be truncated. Function names cannot contain spaces. You will get an error if you try to define a function having the same name as an existing gretl command, or with the same name as a previously defined user function. To avoid an error in the latter case (that is, to be able to redefine a user function), preface the function definition with

```
function function-name clear
```

The *parameters* for a function (if any) are given in the form of a comma-separated list. Parameters can be of three types: ordinary variables (data series), scalar variables, or named lists of variables. Each element in the listing of parameters is composed of two terms: first a type specifier (`series`, `scalar` or `list`) then the name by which the parameter shall be known within the function. An example follows (the parentheses enclosing the list of parameters are optional):

```
function myfunc (series y, list xvars, scalar verbose)
```

When a function is called, the parameters are instantiated by arguments given by the caller. There are automatic checks in place to ensure that the number of arguments given in a function call matches the number of parameters, and that the types of the given arguments match the types specified in the definition of the function. An error is flagged if either of these conditions is violated. A series argument may be specified either using either the name of the variable in question or its ID number. Scalar arguments may be specified by giving the name of a variable or a numerical value (the ID number of a variable is not acceptable). List arguments must be specified by name.

The *function body* is composed of gretl commands, or calls to user-defined functions (that is, functions may be nested). A function may call itself (that is, functions may be recursive). There is a maximum "stacking depth" for user functions: at present this is set to 8. While the function body may contain function calls, it may not contain function definitions. That is, you cannot define a function inside another function.

Functions may be called, but may not be defined, within the context of a command loop (see Chapter 12).

[1] Note that the revised definition of functions represents a backward-incompatible change relative to version 1.3.3 of the program.

13.3 Calling a function

A user function is called or invoked by typing its name followed by zero or more arguments. If there are two or more arguments these should be separated by commas. The following trivial example illustrates a function call that correctly matches the function definition.

```
# function definition
function ols_ess (series y, list xvars)
  ols y 0 xvars --quiet
  scalar myess = $ess
  printf "ESS = %g\n", myess
  return scalar myess
end function
# main script
open data4-1
list xlist = 2 3 4
# function call (the return value is ignored here)
ols_ess price, xlist
```

The function call gives two arguments: the first is a data series specified by name and the second is a named list of regressors. Note that while the function offers the variable `myess` as a return value, it is ignored by the caller in this instance. (As a side note here, if you want a function to calculate some value having to do with a regression, but are not interested in the full results of the regression, you may wish to use the `--quiet` flag with the estimation command as shown above.)

A second example shows how to write a function call that assigns return values to variables in the caller:

```
# function definition
function ess_uhat (series y, list xvars)
  ols y 0 xvars --quiet
  scalar myess = $ess
  printf "ESS = %g\n", myess
  series uh = $uhat
  return scalar myess, series uh
end function
# main script
open data4-1
list xlist = 2 3 4
# function call
(SSR, resids) = ess_uhat price, xlist
```

13.4 Scope of variables

All variables created within a function are local to that function, and are destroyed when the function exits, unless they are made available as return values and these values are "picked up" or assigned by the caller.

Functions do not have access to variables in "outer scope" (that is, variables that exist in the script from which the function is called) except insofar as these are explicitly passed to the function as arguments. Even in this case, what the function actually gets is a copy of the variables in question. Therefore, variables in outer scope are never modified by a function other than via assignment of the return values from the function.

13.5 Return values

Functions can return zero or more values; these can be series or scalars (not lists). Return values are specified via a statement within the function body beginning with the keyword `return`, followed

by a comma-separated list, each element of which is composed of a type specifier and the name of a variable (as in the listing of parameters). There can be only one such statement. An example of a valid return statement is shown below:

```
return scalar SSR, series resid
```

Note that the `return` statement does *not* cause the function to return (exit) at the point where it appears within the body of the function. Rather, it specifies which variables are available for assignment when the function exits, and a function exits only when (a) the end of the function code is reached, or (b) a `funcerr` statement is reached (see below), or (c) a gretl error occurs.

The `funcerr` keyword, which may be followed by a string enclosed in double quotes, causes a function to exit with an error flagged. If a string is provided, this is printed on exit otherwise a generic error message is printed.

13.6 Error checking

When gretl first reads and "compiles" a function definition there is minimal error-checking: the only checks are that the function name is acceptable, and, so far as the body is concerned, that you are not trying to define a function inside a function (see Section 13.2). Otherwise, if the function body contains invalid commands this will become apparent only when the function is called, and its commands are executed.

Chapter 14

Persistent objects

FIXME This chapter needs to deal with saving models too.

14.1 Named lists

Many gretl commands take one or more lists of variables as arguments. To make this easier to handle in the context of command scripts, and in particular within user-defined functions, gretl offers the possibility of *named lists*.

Creating and modifying named lists

A named list is created using the keyword `list`, followed by the name of the list, an equals sign, and either `null` (to create an empty list) or one or more variables to be placed on the list. For example,

```
list xlist = 1 2 3 4
list reglist = income price
list empty_list = null
```

The name of the list must start with a letter, and must be composed entirely of letters, numbers or the underscore character. The maximum length of the name is 15 characters; list names cannot contain spaces. When adding variables to a list, you can refer to them either by name or by their ID numbers.

Once a named list has been created, it will be "remembered" for the duration of the gretl session, and can be used in the context of any gretl command where a list of variables is expected. One simple example is the specification of a list of regressors:

```
list xlist = x1 x2 x3 x4
ols y 0 xlist
```

Lists can be modified in two ways. To *redefine* an existing list altogether, use the same syntax as for creating a list. For example

```
list xlist = 1 2 3
list xlist = 4 5 6
```

After the second assignment, `xlist` contains just variables 4, 5 and 6.

To *append* or *prepend* variables to an existing list, we simply make use of the fact that a named list can stand in for a "longhand" list. For example, we can do

```
list xlist = xlist 5 6 7
list xlist = 9 10 xlist 11 12
```

Querying a list

You can determine whether an unknown variable actually represents a list using the function `islist()`.

```
series xl1 = log(x1)
series xl2 = log(x2)
list xlogs = xl1 xl2
genr is1 = islist(xlogs)
genr is2 = islist(xl1)
```

The first `genr` command above will assign a value of 1 to `is1` since `xlogs` is in fact a named list. The second genr will assign 0 to `is2` since `xl1` is a data series, not a list.

You can also determine the number of variables or elements in a list using the function `nelem()`.

```
list xlist = 1 2 3
genr nl = nelem(xlist)
```

The scalar `nl` will be assigned a value of 3 since `xlist` contains 3 members.

You can display the membership of a named list as illustrated in this interactive session:

```
? list xlist = x1 x2 x3
 # list xlist = x1 x2 x3
Added list 'xlist'
? list xlist print
 # list xlist = x1 x2 x3
```

Note that `print xlist` will do something different, namely print the values of all the variables in `xlist`.

Generating lists of transformed variables

Given a named list of variables, you are able to generate lists of transformations of these variables using a special form of the commands `logs`, `lags`, `diff`, `ldiff`, `sdiff` or `square`. In this context these keywords must be followed directly by a named list in parentheses. For example

```
list xlist = x1 x2 x3
list lxlist = logs(xlist)
list difflist = diff(xlist)
```

When generating a list of *lags* in this way, you can specify the maximum lag order inside the parentheses, before the list name and separated by a comma. For example

```
list xlist = x1 x2 x3
list laglist = lags(2, xlist)
```

or

```
scalar order = 4
list laglist = lags(order, xlist)
```

These command will populate `laglist` with the specified number of lags of the variables in `xlist`. (As with the ordinary `lags` command, you can omit the order, in which case this is determined automatically based on the frequency of the data.) One further special feature is available when generating lags, namely, you can give the name of a single variable in place of a named list on the right-hand side, as in

```
series lx = log(x)
list laglist = lags(4, lx)
```

Note that the ordinary syntax for, e.g., `logs`, is just

```
logs x1 x2 x3
```

If `xlist` is a named list, you can also say

```
logs xlist
```

but this form will not save the logs as a named list; for that you need the form

```
list loglist = logs(xlist)
```

Chapter 15

Time series models

15.1 ARIMA models

Representation and syntax

The `arma` command performs estimation of autoregressive-moving average (ARMA) models; the most general representation of an ARMA model that may be estimated by gretl is as follows:

$$A(L)B(L^s)y_t = x_t\beta + C(L)D(L^s)\epsilon_t, \tag{15.1}$$

where L is the lag operator ($L^n x_t = x_{t-n}$), s is the number of subperiods for seasonal time series (for example, 12 for monthly series), x_t is a vector of exogenous variables and ϵ_t is a white noise process.

The basic ARMA model is obtained when $x_t = 1$ and there are no seasonal operators. In this case, $B(L^s) = D(L^s) = 1$ and the model becomes

$$A(L)y_t = \mu + C(L)\epsilon_t, \tag{15.2}$$

where, in customary notation, the vector β reduces to the intercept μ. The equation above may be written more explicitly as

$$y_t = \mu + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t + \theta_1\epsilon_{t-1} + \ldots + \theta_q\epsilon_{t-q}$$

The corresponding gretl syntax is simply

```
arma p q ; y
```

where `p` and `q` are the desired lag orders; these can be either numbers or pre-defined scalars. The parameter μ can be dropped if necessary by appending the option `--nc` to the command.

If you want to estimate a model with explanatory variables, the above syntax must be extended to

```
arma p q ; y const x1 x2
```

This command would estimate the following model:

$$y_t = \beta_0 + x_{1,t}\beta_1 + x_{2,t}\beta_2 + \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \epsilon_t + \theta_1\epsilon_{t-1} + \ldots + \theta_q\epsilon_{t-q}.$$

What gretl estimates in this case is known as an ARMAX (ARMA + eXogenous variables) model, which is different from what some other packages call "regression model with ARMA errors". The difference is apparent by considering the model gretl estimates:

$$A(L)y_t = x_t\beta + C(L)\epsilon_t, \tag{15.3}$$

and a regression model with ARMA disturbances

$$y_t = x_t\beta + u_t \tag{15.4}$$
$$A(L)u_t = C(L)\epsilon_t; \tag{15.5}$$

the latter would translate into

$$A(L)y_t = A(L)\left(x_t\beta\right) + C(L)\epsilon_t;$$

the ARMAX formulation has the advantage that β can be immediately interpreted as the vector of marginal effects of the x_t variables on the conditional mean of y_t. Note, however, that regression models with purely autoregressive errors can be estimated (albeit not with maximum-likelihood techniques) by other gretl commands such as `corc` and `pwe`.

The `arma` command can be therefore used also for estimating *Transfer Function Models*, a type of generalization of autoregressive-moving average (ARMA) models adding the effects of exogenous variable distributed along time, as in:

$$\phi(L) \cdot \Phi(L^s) y_t = \sum_{i=1}^{k} v_i(L) x_{it} + \theta(L) \cdot \Theta(L^s) \epsilon_t, \tag{15.6}$$

The structure of the `arma` command does not let you specify models with gaps in the lag structure. A more flexible lag structure is especially desirable when analyzing time series that display strong seasonal patterns. In these cases, the full model (15.1) can be used. For example, the syntax

```
arma 1 1 ; 1 1 ; y
```

would be used to estimate a model with four parameters:

$$(1 - \phi L)(1 - \Phi L^s) y_t = \mu + (1 + \theta L)(1 + \Theta L^s) \epsilon_t;$$

assuming that y_t is a quarterly series (and therefore $s = 4$), the above equation can be written more explicitly as

$$y_t = \mu + \phi y_{t-1} + \Phi y_{t-4} - (\phi \cdot \Phi) y_{t-5} + \epsilon_t + \theta \epsilon_{t-1} + \Theta \epsilon_{t-4} + (\theta \cdot \Theta) \epsilon_{t-5}.$$

Such a model is known as a "multiplicative seasonal ARMA model".

For more general models, this limitation can be circumvented for the autoregressive part by including lags of the dependent variable in the exogenous list. As an example, the following command

```
arma 0 0 ; 0 1 ; y const y(-2)
```

on a quarterly series would estimate the parameters of the model

$$y_t = \mu + \phi y_{t-2} + \epsilon_t + \Theta \epsilon_{t-4}.$$

However, this workaround is not recommended: although this would deliver correct estimates, it would break the existing mechanism for forecasting.

The above discussion presupposes that the time series y_t has already been subjected to all the transformations deemed necessary for ensuring stationarity (see also section 15.2). Differencing is the most common of these transformations, and gretl provides a mechanism to include this step into the `arma` command: the syntax

```
arma p d q ; y
```

would estimate and ARMA(p,q) model on $\Delta^d y_t$, and is functionally equivalent to

```
series tmp = y
loop for i=1..d
  tmp = diff(tmp)
end loop
arma p q ; tmp
```

Such a model is known as an ARIMA (AutoRegressive Integrated Moving-Average) model; for this reason, gretl provides the `arima` command as an alias for `arma`. Seasonal differencing is handled similarly, with the syntax

```
arma p d q ; P D Q ; y
```

Thus, the command

```
arma 1 0 0 ; 1 1 1 ; y
```

would produce the same results as

```
genr dsy = sdiff(y)
arma 1 0 ; 1 1 ; dsy
```

Estimation

The algorithm gretl uses to estimate the parameters of an ARMA model is conditional maximum likelihood (CML), also known as "conditional sum of squares" — see Hamilton (1994, p. 132). This method was exemplified in the script 12.3, and only a brief description will be given here. Given a sample of size T, the CML method minimizes the sum of squared one-step-ahead prediction errors generated by the model for the observations $t_0, \ldots, T$. The starting point t_0 depends on the orders of the AR polynomials in the model.

This method is nearly equivalent to maximum likelihood under the hypothesis of normality; the difference is that the first $(t_0 - 1)$ observations are considered fixed and only enter the likelihood function as conditioning variables. As a consequence, the two methods are asymptotically equivalent under standard conditions.

The numerical method used for maximizing the log-likelihood is BHHH. The covariance matrix for the parameters (hence the standard errors) are computed via the OPG (Outer Product of the Gradients) method.

Full ML estimation under normality is available in gretl via the `x-12` plugin, which is automatically used if the option `--x-12-arima` is appendend to the `arma` command. For example, the following code

```
open data10-1
arma 1 1 ; r
arma 1 1 ; r --x-12-arima
```

produces the estimates shown in Table 15.1.

Table 15.1: CML and ML estimates

Parameter	CML		ML (X-12-ARIMA plugin)	
μ	1.07322	(0.488661)	1.00232	(0.133002)
ϕ	0.852772	(0.0450252)	0.855373	(0.0496304)
θ	0.591838	(0.0456662)	0.587986	(0.0799962)

For comparability with gretl's own estimates, you can instruct X-12-ARIMA to produce CML estimates by appending the option `--conditional` along with `--x-12-arima`.

Forecasting

To be written

15.2 Unit root tests

To be completed

The ADF test

The ADF (Augmented Dickey-Fuller) test is, as implemented in gretl, the t-statistic on φ in the following regression:

$$\Delta y_t = \mu_t + \varphi y_{t-1} + \sum_{i=1}^{p} \gamma_i \Delta y_{t-i} + \epsilon_t. \tag{15.7}$$

This test statistic is probably the best-known and most widely used unit root test. It is a one-sided test whose null hypothesis is $\varphi = 0$ versus the alternative $\varphi < 0$. Under the null, y_t must be differenced at least once to achieve stationarity; under the alternative, y_t is already stationary and no differencing is required. Hence, large negative values of the test statistic lead to the rejection of the null.

One peculiar aspect of this test is that its limit distribution is non-standard under the null hypothesis: moreover, the shape of the distribution, and consequently the critical values for the test, depends on the form of the μ_t term. A full analysis of the various cases is inappropriate here: Hamilton (1994) contains an excellent discussion, but any recent time series textbook covers this topic. Suffice it to say that gretl allows the user to choose the specification for μ_t among four different alternatives:

μ_t	command option
0	`--nc`
μ_0	`--c`
$\mu_0 + \mu_1 t$	`--ct`
$\mu_0 + \mu_1 t + \mu_1 t^2$	`--ctt`

These options are not mutually exclusive and can be used together; in this case, the statistic will be reported separately for each case. By default, gretl uses by default the combination `--c --ct --ctt`. For each case, approximate p-values are calculated by means of the algorithm developed in MacKinnon (1996).

The gretl command used to perform the test is `adf`; for example

```
adf 4 x1 --c --ct
```

would compute the test statistic as the t-statistic for φ in equation 15.7 with $p = 4$ in the two cases $\mu_t = \mu_0$ and $\mu_t = \mu_0 + \mu_1 t$.

The number of lags (p in equation 15.7) should be chosen as to ensure that (15.7) is a parametrization flexible enough to represent adequately the short-run persistence of Δy_t. Setting p too low results in size distortions in the test, whereas setting p too high would lead to low power. As a convenience to the user, the parameter p can be automatically determined. Setting p to a negative number triggers a sequential procedure that starts with p lags and decrements p until the t-statistic for the parameter γ_p exceeds 1.645 in absolute value.

The KPSS test

The KPSS test (Kwiatkowski, Phillips, Schmidt and Shin, 1992) is a unit root test in which the null hypothesis is opposite to that in the ADF test: under the null, the series in question is stationary; the alternative is that the series is $I(1)$.

The basic intuition behind this test statistic is very simple: if y_t can be written as $y_t = \mu + u_t$, where u_t is some zero-mean stationary process, then not only does the sample average of the y_t's provide a consistent estimator of μ, but the long-run variance of u_t is a well-defined, finite number. Neither of these properties hold under the alternative.

The test itself is based on the following statistic:

$$\eta = \frac{\sum_{i=1}^{T} S_t^2}{T^2 \bar{\sigma}^2} \tag{15.8}$$

where $S_t = \sum_{s=1}^{t} e_s$ and $\bar{\sigma}^2$ is an estimate of the long-run variance of $e_t = (y_t - \bar{y})$. Under the null, this statistic has a well-defined (nonstandard) asymptotic distribution, which is free of nuisance parameters and has been tabulated by simulation. Under the alternative, the statistic diverges.

As a consequence, it is possible to construct a one-sided test based on η, where H_0 is rejected if η is bigger than the appropriate critical value; gretl provides the 90%, 95%, 97.5% and 99% quantiles.

Usage example:

```
kpss m y
```

where `m` is an integer representing the bandwidth or window size used in the formula for estimating the long run variance:

$$\bar{\sigma}^2 = \sum_{i=-m}^{m} \left(1 - \frac{|i|}{m+1}\right) \hat{\gamma}_i$$

The $\hat{\gamma}_i$ terms denote the empirical autocovariances of e_t from order $-m$ through m. For this estimator to be consistent, m must be large enough to accommodate the short-run persistence of e_t, but not too large compared to the sample size T. In the GUI interface of gretl, this value defaults to the integer part of $4\left(\frac{T}{100}\right)^{1/4}$.

The above concept can be generalized to the case where y_t is thought to be stationary around a deterministic trend. In this case, formula (15.8) remains unchanged, but the series e_t is defined as the residuals from an OLS regression of y_t on a constant and a linear trend. This second form of the test is obtained by appending the `--trend` option to the `kpss` command:

```
kpss n y --trend
```

Note that in this case the asymptotic distribution of the test is different and the critical values reported by gretl differ accordingly.

The Johansen tests

Strictly speaking, these are tests for cointegration. However, they can be used as multivariate unit-root tests since they are the multivariate generalization of the ADF test.

$$\Delta y_t = \mu_t + \Pi y_{t-1} + \sum_{i=1}^{p} \Gamma_i \Delta y_{t-i} + \epsilon_t \tag{15.9}$$

If the rank of Π is 0, the processes are all I(1); If the rank of Π is full, the processes are all I(0); in between, Π can be written as $\alpha\beta'$ and you have cointegration.

The rank of Π is investigated by computing the eigenvalues of a closely related matrix (call it M) whose rank is the same as Π: however, M is by construction symmetric and positive semidefinite. As a consequence, all its eigenvalues are real and non-negative; tests on the rank of Π can therefore be carried out by testing how many eigenvalues of M are 0.

If all the eigenvalues are significantly different from 0, then all the processes are stationary. If, on the contrary, there is at least one zero eigenvalue, then the y_t process is integrated, although some linear combination $\beta' y_t$ might be stationary. On the other extreme, if no eigenvalues are significantly different from 0, then not only the process y_t is non-stationary, but the same holds for any linear combination $\beta' y_t$; in other words, no cointegration occurs.

The two Johansen tests are the "λ-max" test, for hypotheses on individual eigenvalues, and the "trace" test, for joint hypotheses. The gretl command `coint2` performs these two tests.

As in the ADF test, the asymptotic distribution of the tests varies with the deterministic kernel μ_t one includes in the VAR. gretl provides the following options (for a short discussion of the meaning of the five options, see section 15.4 below):

μ_t	command option
0	`--nc`
$\mu_0, \alpha'_\perp \mu_0 = 0$	`--rc`
μ_0	default
$\mu_0 + \mu_1 t, \alpha'_\perp \mu_1 = 0$	`--crt`
$\mu_0 + \mu_1 t$	`--ct`

Note that for this command the above options are mutually exclusive. In addition, you have the option of using the `--seasonal` options, for augmenting μ_t with centered seasonal dummies. In each case, p-values are computed via the approximations by Doornik (1998).

The following code uses the `denmark` database, supplied with gretl, to replicate Johansen's example found in his 1995 book.

```
open denmark
coint2 2 LRM LRY IBO IDE --rc --seasonal
```

In this case, the vector y_t in equation (15.9) comprises the four variables `LRM`, `LRY`, `IBO`, `IDE`. The number of lags equals p in (15.9) plus one. Part of the output is reported below:

```
Johansen test:
Number of equations = 4
Lag order = 2
Estimation period: 1974:3 - 1987:3 (T = 53)

Case 2: Restricted constant
Rank Eigenvalue Trace test p-value   Lmax test  p-value
   0    0.43317     49.144 [0.1284]     30.087 [0.0286]
   1    0.17758     19.057 [0.7833]     10.362 [0.8017]
   2    0.11279     8.6950 [0.7645]     6.3427 [0.7483]
   3   0.043411     2.3522 [0.7088]     2.3522 [0.7076]

eigenvalue     0.43317      0.17758      0.11279     0.043411
```

Since both the trace and λ-max accept the null hypothesis that the smallest eigenvalue is in fact 0, we may conclude that the series are in fact non-stationary. However, some linear combination may be $I(0)$, as indicated by the rejection of the λ-max of the hypothesis that the rank of Π is 0 (the trace test gives less clear-cut evidence for this).

15.3 ARCH and GARCH

Heteroskedasticity means a non-constant variance of the error term in a regression model. Autoregressive Conditional Heteroskedasticity (ARCH) is a phenomenon specific to time series models, whereby the variance of the error displays autoregressive behavior; for instance, the time series exhibits successive periods where the error variance is relatively large, and successive periods where it is relatively small. This sort of behavior is reckoned to be quite common in asset markets: an unsettling piece of news can lead to a period of increased volatility in the market.

An ARCH error process of order q can be represented as

$$u_t = \sigma_t \varepsilon_t; \qquad \sigma_t^2 \equiv \mathrm{E}(u_t^2 | \Omega_{t-1}) = \alpha_0 + \sum_{i=1}^{q} \alpha_i u_{t-i}^2$$

where the ε_ts are independently and identically distributed (iid) with mean zero and variance 1, and where σ_t is taken to be the positive square root of σ_t^2. Ω_{t-1} denotes the information set as of time $t-1$ and σ_t^2 is the conditional variance: that is, the variance conditional on information dated $t-1$ and earlier.

It is important to notice the difference between ARCH and an ordinary autoregressive error process. The simplest (first-order) case of the latter can be written as

$$u_t = \rho u_{t-1} + \varepsilon_t; \qquad -1 < \rho < 1$$

where the ε_ts are iid with mean zero and constant variance σ^2. With an AR(1) error, if ρ is positive then a positive value of u_t will tend to be followed, with probability greater than 0.5, by a positive u_{t+1}. With an ARCH error process, a disturbance u_t of large absolute value will tend to be followed by further large absolute values, but with no presumption that the successive values will be of the same sign. ARCH in asset prices is a "stylized fact" and is consistent with market efficiency; on the other hand autoregressive behavior of asset prices would violate market efficiency.

One can test for ARCH of order q in the following way:

1. Estimate the model of interest via OLS and save the squared residuals, $\hat{u}_t^2$.
2. Perform an auxiliary regression in which the current squared residual is regressed on a constant and q lags of itself.
3. Find the TR^2 value (sample size times unadjusted R^2) for the auxiliary regression.
4. Refer the TR^2 value to the χ^2 distribution with q degrees of freedom, and if the p-value is "small enough" reject the null hypothesis of homoskedasticity in favor of the alternative of ARCH(q).

This test is implemented in gretl via the `arch` command. This command may be issued following the estimation of a time-series model by OLS, or by selection from the "Tests" menu in the model window (again, following OLS estimation). The result of the test is reported and if the TR^2 from the auxiliary regression has a p-value less than 0.10, ARCH estimates are also reported. These estimates take the form of Generalized Least Squares (GLS), specifically weighted least squares, using weights that are inversely proportional to the predicted standard deviations of the disturbances, $\hat{\sigma}_t$, derived from the auxiliary regression.

In addition, the ARCH test is available after estimating a vector autoregression (VAR). In this case, however, there is no provision to re-estimate the model via GLS.

GARCH

The simple ARCH(q) process is useful for introducing the general concept of conditional heteroskedasticity in time series, but it has been found to be insufficient in empirical work. The dynamics of the error variance permitted by ARCH(q) are not rich enough to represent the patterns found in financial data. The generalized ARCH or GARCH model is now more widely used.

The representation of the variance of a process in the GARCH model is somewhat (but not exactly) analogous to the ARMA representation of the level of a time series. The variance at time t is allowed to depend on both past values of the variance and past values of the realized squared disturbance, as shown in the following system of equations:

$$y_t = X_t\beta + u_t \tag{15.10}$$

$$u_t = \sigma_t \varepsilon_t \tag{15.11}$$

$$\sigma_t^2 = \alpha_0 + \sum_{i=1}^{q} \alpha_i u_{t-i}^2 + \sum_{j=1}^{p} \delta_i \sigma_{t-j}^2 \tag{15.12}$$

As above, ε_t is an iid sequence with unit variance. X_t is a matrix of regressors (or in the simplest case, just a vector of 1s allowing for a non-zero mean of y_t). Note that if $p = 0$, GARCH collapses to

ARCH(q): the generalization is embodied in the δ_i terms that multiply previous values of the error variance.

In principle the underlying innovation, ε_t, could follow any suitable probability distribution, and besides the obvious candidate of the normal or Gaussian distribution the t distribution has been used in this context. Currently gretl only handles the case where ε_t is assumed to be Gaussian. However, when the `--robust` option to the `garch` command is given, the estimator gretl uses for the covariance matrix can be considered Quasi-Maximum Likelihood even with non-normal disturbances. See below for more on the options regarding the GARCH covariance matrix.

Example:

```
garch p q ; y const x
```

where `p` ≥ 0 and `q` > 0 denote the respective lag orders as shown in equation (15.12). These values can be supplied in numerical form or as the names of pre-defined scalar variables.

GARCH estimation

Estimation of the parameters of a GARCH model is by no means a straightforward task. (Consider equation 15.12: the conditional variance at any point in time, σ_t^2, depends on the conditional variance in earlier periods, but σ_t^2 is not observed, and must be inferred by some sort of Maximum Likelihood procedure.) Gretl uses the method proposed by Fiorentini, Calzolari and Panattoni (1996),[1] which was adopted as a benchmark in the study of GARCH results by McCullough and Renfro (1998). It employs analytical first and second derivatives of the log-likelihood, and uses a mixed-gradient algorithm, exploiting the information matrix in the early iterations and then switching to the Hessian in the neighborhood of the maximum likelihood. (This progress can be observed if you append the `--verbose` option to gretl's `garch` command.)

Several options are available for computing the covariance matrix of the parameter estimates in connection with the `garch` command. At a first level, one can choose between a "standard" and a "robust" estimator. By default, the Hessian is used unless the `--robust` option is given, in which case the QML estimator is used. A finer choice is available via the `set` command, as shown in Table 15.2.

Table 15.2: Options for the GARCH covariance matrix

command	*effect*
`set garch_vcv hessian`	Use the Hessian
`set garch_vcv im`	Use the Information Matrix
`set garch_vcv op`	Use the Outer Product of the Gradient
`set garch_vcv qml`	QML estimator
`set garch_vcv bw`	Bollerslev–Wooldridge "sandwich" estimator

It is not uncommon, when one estimates a GARCH model for an arbitrary time series, to find that the iterative calculation of the estimates fails to converge. For the GARCH model to make sense, there are strong restrictions on the admissible parameter values, and it is not always the case that there exists a set of values inside the admissible parameter space for which the likelihood is maximized.

The restrictions in question can be explained by reference to the simplest (and much the most common) instance of the GARCH model, where $p = q = 1$. In the GARCH(1, 1) model the conditional variance is

$$\sigma_t^2 = \alpha_0 + \alpha_1 u_{t-1}^2 + \delta_1 \sigma_{t-1}^2 \tag{15.13}$$

[1] The algorithm is based on Fortran code deposited in the archive of the *Journal of Applied Econometrics* by the authors, and is used by kind permission of Professor Fiorentini.

Taking the unconditional expectation of (15.13) we get

$$\sigma^2 = \alpha_0 + \alpha_1\sigma^2 + \delta_1\sigma^2$$

so that

$$\sigma^2 = \frac{\alpha_0}{1 - \alpha_1 - \delta_1}$$

For this unconditional variance to exist, we require that $\alpha_1 + \delta_1 < 1$, and for it to be positive we require that $\alpha_0 > 0$.

A common reason for non-convergence of GARCH estimates (that is, a common reason for the non-existence of α_i and δ_i values that satisfy the above requirements and at the same time maximize the likelihood of the data) is misspecification of the model. It is important to realize that GARCH, in itself, allows *only* for time-varying volatility in the data. If the *mean* of the series in question is not constant, or if the error process is not only heteroskedastic but also autoregressive, it is necessary to take this into account when formulating an appropriate model. For example, it may be necessary to take the first difference of the variable in question and/or to add suitable regressors, X_t, as in (15.10).

15.4 Cointegration and Vector Error Correction Models

The Johansen cointegration test

The Johansen test for cointegration has to take into account what hypotheses one is willing to make on the deterministic terms, which leads to the famous "five cases." A full and general illustration of the five cases requires a fair amount of matrix algebra, but an intuitive understanding of the issue can be gained by means of a simple example.

Consider a series x_t which behaves as follows

$$x_t = m + x_{t-1} + \varepsilon_t$$

where m is a real number and ε_t is a white noise process. As is easy to show, x_t is a random walk which fluctuates around a deterministic trend with slope m. In the special case $m = 0$, the deterministic trend disappears and x_t is a pure random walk.

Consider now another process y_t, defined by

$$y_t = k + x_t + u_t$$

where, again, k is a real number and u_t is a white noise process. Since u_t is stationary by definition, x_t and y_t cointegrate: that is, their difference

$$z_t = y_t - x_t = k + u_t$$

is a stationary process. For $k = 0$, z_t is simple zero-mean white noise, whereas for $k \neq 0$ the process z_t is white noise with a non-zero mean.

After some simple substitutions, the two equations above can be represented jointly as a VAR(1) system

$$\begin{bmatrix} y_t \\ x_t \end{bmatrix} = \begin{bmatrix} k+m \\ m \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} y_{t-1} \\ x_{t-1} \end{bmatrix} + \begin{bmatrix} u_t + \varepsilon_t \\ \varepsilon_t \end{bmatrix}$$

or in VECM form

$$\begin{aligned} \begin{bmatrix} \Delta y_t \\ \Delta x_t \end{bmatrix} &= \begin{bmatrix} k+m \\ m \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} y_{t-1} \\ x_{t-1} \end{bmatrix} + \begin{bmatrix} u_t + \varepsilon_t \\ \varepsilon_t \end{bmatrix} = \\ &= \begin{bmatrix} k+m \\ m \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} \begin{bmatrix} y_{t-1} \\ x_{t-1} \end{bmatrix} + \begin{bmatrix} u_t + \varepsilon_t \\ \varepsilon_t \end{bmatrix} = \\ &= \mu_0 + \alpha\beta' \begin{bmatrix} y_{t-1} \\ x_{t-1} \end{bmatrix} + \eta_t = \mu_0 + \alpha z_{t-1} + \eta_t, \end{aligned}$$

where β is the cointegration vector and α is the "loadings" or "adjustments" vector.

We are now in a position to consider three possible cases:

1. $m \neq 0$: In this case x_t is trended, as we just saw; it follows that y_t also follows a linear trend because on average it keeps at a distance k from x_t. The vector μ_0 is unrestricted. This case is the default for gretl's `vecm` command.

2. $m = 0$ and $k \neq 0$: In this case, x_t is not trended and as a consequence neither is y_t. However, the mean distance between y_t and x_t is non-zero. The vector μ_0 is given by

$$\mu_0 = \begin{bmatrix} k \\ 0 \end{bmatrix}$$

   which is not null and therefore the VECM shown above does have a constant term. The constant, however, is subject to the restriction that its second element must be 0. More generally, μ_0 is a multiple of the vector α. Note that the VECM could also be written as

$$\begin{bmatrix} \Delta y_t \\ \Delta x_t \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & -1 & -k \end{bmatrix} \begin{bmatrix} y_{t-1} \\ x_{t-1} \\ 1 \end{bmatrix} + \begin{bmatrix} u_t + \varepsilon_t \\ \varepsilon_t \end{bmatrix}$$

   which incorporates the intercept into the cointegration vector. This is known as the "restricted constant" case; it may be specified in gretl's `vecm` command using the option flag `--rc`.

3. $m = 0$ and $k = 0$: This case is the most restrictive: clearly, neither x_t nor y_t are trended, and the mean distance between them is zero. The vector μ_0 is also 0, which explains why this case is referred to as "no constant." This case is specified using the option flag `--nc` with `vecm`.

In most cases, the choice between the three possibilities is based on a mix of empirical observation and economic reasoning. If the variables under consideration seem to follow a linear trend then we should not place any restriction on the intercept. Otherwise, the question arises of whether it makes sense to specify a cointegration relationship which includes a non-zero intercept. One example where this is appropriate is the relationship between two interest rates: generally these are not trended, but the VAR might still have an intercept because the difference between the two (the "interest rate spread") might be stationary around a non-zero mean (for example, because of a risk or liquidity premium).

The previous example can be generalized in three directions:

1. If a VAR of order greater than 1 is considered, the algebra gets more convoluted but the conclusions are identical.

2. If the VAR includes more than two endogenous variables the cointegration rank r can be greater than 1. In this case, α is a matrix with r columns, and the case with restricted constant entails the restriction that μ_0 should be some linear combination of the columns of α.

3. If a linear trend is included in the model, the deterministic part of the VAR becomes $\mu_0 + \mu_1 t$. The reasoning is practically the same as above except that the focus now centers on μ_1 rather than μ_0. The counterpart to the "restricted constant" case discussed above is a "restricted trend" case, such that the cointegration relationships include a trend but the first differences of the variables in question do not. In the case of an unrestricted trend, the trend appears in both the cointegration relationships and the first differences, which corresponds to the presence of a quadratic trend in the variables themselves (in levels). These two cases are specified by the option flags `--crt` and `--ct`, respectively, with the `vecm` command.

Chapter 16

Matrix manipulation

16.1 Introduction

As of version 1.5.1, gretl offers the facility of creating and manipulating user-defined matrices. This is currently experimental.

16.2 Creating matrices

Matrices can be created in any one of four ways:

1. By direct specification of the scalar values that compose the matrix, in numerical form or by reference to pre-existing scalar variables, or both; or
2. by providing a list of data series; or
3. by providing a *named list* of series; or
4. using a formula of the same general type that is used with the `genr` command, whereby a new matrix is defined in terms of existing matrices and/or scalars, or via some special functions.

These methods cannot be mixed in the specification of a given matrix. Examples of each follow.

To specify a matrix *directly in terms of scalars*, the syntax is, for example:

```
matrix A = { 1, 2, 3 ; 4, 5, 6 }
```

The matrix is defined by rows; the elements on each row are separated by commas and the rows are separated by semi-colons. The whole expression must be wrapped in braces. Spaces within the braces are not significant. The above expression defines a 2×3 matrix. Each element must be either a numerical value or the name of a pre-existing scalar variable. Directly after the closing brace you can append a single quote (') to obtain the transpose.

To specify a matrix *in terms of data series* the syntax is, for example,

```
matrix A = { x1, x2, x3 }
```

where the names of the variables are separated by commas. By default, each variable occupies a column (and there can only be one variable per column). The range of data values included in the matrix depends on the current setting of the sample range.

Please note that while gretl's built-in statistical functions are capable of handling missing values, the matrix arithmetic functions are not, and you will get an error if you try to build a matrix from series that include missing values.

Instead of giving an explicit list of variables, you may instead provide the *name of a saved list* (see Chapter 14), as in

```
list xlist = x1 x2 x3
matrix A = { xlist }
```

When you provide a named list, the data series are by default placed in columns, as is natural in an econometric context: if you want them in rows, append the transpose symbol.

You can create new matrices, or replace existing matrices, by means of various transformations, in a manner similar to the `genr` command for scalars and data series. To get a matrix result, however, the command must start with the keyword `matrix`, not `genr`. The relevant mechanisms are discussed in the next several sections.

☞ Names of matrices must satisfy the same requirements as names of gretl variables in general: the name can be no longer than 15 characters, must start with a letter, and must be composed of nothing but letters, numbers and the underscore character.

16.3 Matrix operators

The following operators are available for matrices:

`+`	addition
`-`	subtraction
`*`	ordinary matrix multiplication
`/`	matrix "division" (see below)
`.*`	element-wise multiplication
`./`	element-wise division
`.^`	element-wise exponentiation
`~`	column-wise concatenation
`**`	Kronecker product
`=`	test for equality

Here are explanations of the less obvious cases. First, in matrix "division", A/B is algebraically equivalent to $B^{-1}A$ (pre-multiplication by the inverse of the "divisor"). Therefore the following two expressions are equivalent in principle:

```
matrix C = A / B
matrix C = inv(B) * A
```

where `inv()` is the matrix inversion function (see below for more on matrix functions). The first form, however, may be more accurate than the second; the solution is obtained via LU decomposition, without the explicit calculation of the inverse matrix.

In element-wise multiplication if we write

```
matrix C = A .* B
```

then the result depends on the dimensions of A and B. Let A be an $m \times n$ matrix and let B be $p \times q$.

- If $m = p$ and $n = q$ then C is $m \times n$ with $c_{ij} = a_{ij} \times b_{ij}$. This is technically known as the *Hadamard product*.
- Otherwise, if $m = 1$ and $n = q$, or $n = 1$ and $m = p$, then C is $p \times q$ with $c_{ij} = a_k \times b_{ij}$, where $k = j$ if $m = 1$ else $k = i$.
- Otherwise, if $p = 1$ and $n = q$, or $q = 1$ and $m = p$, then C is $m \times n$ with $c_{ij} = a_{ij} \times b_k$, where $k = j$ if $p = 1$ else $k = i$.
- If none of the above conditions are satisfied the product is undefined and an error is flagged.

Element-wise division works in a manner exactly analogous to element-wise multiplication, simply replacing $\times$ by $\div$ in the account given for multiplication.

Element-wise exponentiation, as in

```
matrix C = A .^ B
```

produces $c_{ij} = a_{ij}^k$. The variable k must be a scalar or 1×1 matrix.

In column-wise concatenation of an $m \times n$ matrix A and an $m \times p$ matrix B, the result is an $m \times (n + p)$ matrix. That is,

```
C = A ~ B
```

produces $C = \left[\begin{array}{cc} A & B \end{array} \right]$.

16.4 Matrix functions

The following functions are available for *element-by-element transformations* of matrices: `log`, `exp`, `sin`, `cos`, `tan`, `atan`, `int`, `abs`, `sqrt`, `dnorm`, `cnorm`, `qnorm`, `gamma` and `lngamma`. These functions have the same meanings as in `genr`. For example, if a matrix `A` is already defined, then

```
matrix B = sqrt(A)
```

generates a matrix such that $b_{ij} = \sqrt{a_{ij}}$. All of these functions require a single matrix as argument, or an expression which evaluates to a single matrix.

The functions `sort()` and `dsort()` are available for matrices as well as data series. In the matrix case the argument to these functions must be a vector ($p \times 1$ or $1 \times p$). The return value is a vector containing the elements of the input vector sorted in ascending order of magnitude (`sort`) or descending order (`dsort`).

Several matrix-specific functions are available. These functions fall into four categories:

1. Those taking a single matrix as argument and returning a scalar.
2. Those taking a single matrix as argument and returning a matrix.
3. Those taking one or two dimensions as arguments and returning a matrix.
4. Those taking one or two matrices as arguments and returning one or two matrices.

These sets of functions are discussed in turn below.

Matrix to scalar functions

The functions which take a single matrix as argument and return a scalar are:

`det()`	determinant
`ldet()`	log-determinant
`tr()`	trace
`onenorm()`	1-norm
`rcond()`	reciprocal condition number
`rows()`	number of rows
`cols()`	number of columns

The single matrix argument to these functions may be given as the name of an existing matrix or as an expression that evaluates to a single matrix. Note that the functions `det`, `ldet` and `tr` require a square matrix as input.

The `onenorm` function returns the 1-norm of a matrix — that is, the maximum across the columns of the matrix of the sums of the absolute values of the column elements. The function `rcond` returns the reciprocal condition number for a symmetric, positive definite matrix.

Matrix to matrix functions

The functions which take a single matrix as argument and return a matrix are:

`sumc()`	sum by column
`sumr()`	sum by row
`inv()`	inverse
`cholesky()`	Cholesky decomposition
`diag()`	extract principal diagonal
`transp()`	transpose
`cdemean()`	subtract column means

As with the previous set of functions, the argument may be given as the name of an existing matrix or as an expression that evaluates to a single matrix.

For a matrix A with m rows and n columns, `sumc(A)` returns a row vector with the n column sums; `sumr(A)` returns a column vector with the m row sums.

The `cholesky` function computes the Cholesky decomposition L of a symmetric positive definite matrix A: $A = LL'$; L is lower triangular (has zeros above the diagonal).

The `diag` function returns the principal diagonal of an $n \times n$ matrix A as a column vector — that is, an n-vector v such that $v_i = a_{ii}$.

The `cdemean` function applied to an $m \times n$ matrix A returns an $m \times n$ matrix B such that $b_{ij} = a_{ij} - \bar{A}_j$, where $\bar{A}_j$ denotes the mean of column j of A.

Matrix filling functions

The functions taking one or two dimensions as arguments and returning a matrix are:

`I(`n`)`	$n \times n$ identity matrix
`zeros(`m`,`n`)`	$m \times n$ zero matrix
`ones(`m`,`n`)`	$m \times n$ matrix filled with 1s
`uniform(`m`,`n`)`	$m \times n$ matrix filled with uniform random values
`normal(`m`,`n`)`	$m \times n$ matrix filled with normal random values

The dimensions m and n may be given numerically, or by reference to pre-existing scalar variables, as in

```
scalar m = 4
scalar n = 5
matrix A = normal(m,n)
```

The `uniform()` and `normal()` matrix functions fill the matrix with drawings from the uniform (0–1) distribution and the standard normal distribution respectively.

Multiple-return matrix functions

The functions that take one or two matrices as arguments and return one or two matrices are:

`qrdecomp()`	QR decomposition
`eigensym()`	Eigen-analysis of symmetric matrix
`eigengen()`	Eigen-analysis of general matrix

The syntax for these functions is of the form

```
matrix B = func(A, C)
```

The first argument, *A*, represents the input data, that is, the matrix whose decomposition or analysis is required. This must be given as the name of an existing matrix; a compound expression that evaluates to a matrix is not accepted in this context.

The second argument, *C*, may be either the name of a matrix, in which case an auxiliary result is written to that matrix, or the keyword `null`, in which case the auxiliary result is not produced, or is discarded.

In case the name of a matrix is given as the second argument, this matrix does not have to be previously defined; a new matrix of this name will be created. If a matrix of the given name already exists, it will be over-written with the auxiliary result. (It is not required that the existing matrix, if any, be of the right dimensions to receive the result.)

The `qrdecomp` function computes the QR decomposition of an $m \times n$ matrix A: $A = QR$, where Q is an $m \times n$ orthogonal matrix and R is an $n \times n$ upper triangular matrix. The matrix Q is returned directly, while R can be retrieved via the second argument. Here are two examples:

```
matrix Q = qrdecomp(M, R)
matrix Q = qrdecomp(M, null)
```

In the first example, the triangular R is saved as `R`; in the second, R is discarded.

The function `eigensym` computes the eigenvalues, and optionally the right eigenvectors, of a symmetric $n \times n$ matrix. The eigenvalues are returned directly in a column vector of length n; if the eigenvectors are required, they are returned in an $n \times n$ matrix. For example:

```
matrix E = eigensym(M, V)
matrix E = eigensym(M, null)
```

In the first case `E` holds the eigenvalues of `M` and `V` holds the eigenvectors. In the second, `E` holds the eigenvalues but the eigenvectors are not computed.

The function `eigengen` computes the eigenvalues, and optionally the eigenvectors, of a general $n \times n$ matrix. The eigenvalues are returned directly in a column vector of length $2n$: the first n elements are the real components and the remaining n are the imaginary components. If the eigenvectors are required (that is, if the second argument to `eigengen` is not `null`), they are returned in an $n \times n$ matrix.

16.5 Matrix accessors

In addition to the matrix functions discussed above, various "accessor" strings allow you to create copies of internal matrices associated with models previously estimated:

`$coeff`	vector of estimated coefficients
`$stderr`	vector of estimated standard errors
`$uhat`	vector of residuals
`$yhat`	vector of fitted values
`$vcv`	covariance matrix for coefficients
`$rho`	autoregressive coefficients for error process

If these accessors are given without any prefix, they retrieve results from the last model estimated, if any. Alternatively, they may be prefixed with the name of a saved model plus a period (`.`), in which case they retrieve results from the specified model. Here are some examples:

```
matrix u = $uhat
matrix b = m1.$coeff
matrix v2 = m1.$vcv[1:2,1:2]
```

The first command grabs the residuals from the last model; the second grabs the coefficient vector from model `m1`; and the third (which uses the mechanism of sub-matrix selection described in the following section) grabs a portion of the covariance matrix from model `m1`.

If the "model" in question is actually a system (a VAR or VECM, or system of simultaneous equations), `$uhat` retrieves the matrix of residuals (one column per equation) and `$vcv` gets the cross-equation covariance matrix; in the special case of a VAR or a VECM, `$coeff` returns the companion matrix. At present the other accessors are not available for equation systems.

16.6 Selecting sub-matrices

You can select sub-matrices of a given matrix using the syntax

`A[`*rows*`,`*cols*`]`

where *rows* can take one of four forms:

empty	selects all rows
a single integer	selects the single specified row
two integers separated by a colon	selects a range of rows
the name of a matrix	selects the specified rows

With regard to the second option, the integer value can be given numerically, or as the name of an existing scalar variable. With the last option, the index matrix given in the *rows* field must be either $p \times 1$ or $1 \times p$, and should contain integer values in the range 1 to n, where n is the number of rows in the matrix from which the selection is to be made.

The *cols* specification works in the same way, *mutatis mutandis*. Here are some examples.

```
matrix B = A[1,]
matrix B = A[2:3,3:5]
matrix B = A[2,2]
matrix idx = { 1, 2, 6 }
matrix B = A[idx,]
```

The first example selects row 1 from matrix `A`; the second selects a 2×3 submatrix; the third selects a scalar; and the fourth selects rows 1, 2, and 6 from matrix `A`.

In addition there is a special pre-defined "index matrix" specification, `diag`, which selects the principal diagonal of a square matrix, as in `B[diag]`, where `B` is square.

You can use selections of this sort on either the right-hand side of a matrix-generating formula or the left. Here is an example of use of a selection on the right, to extract a 2×2 submatrix B from a 3×3 matrix A:

```
matrix A = { 1, 2, 3; 4, 5, 6; 7, 8, 9 }
matrix B = A[1:2,2:3]
```

And here are examples of selection on the left. The second line below writes a 2×2 identity matrix into the bottom right corner of the 3×3 matrix A. The fourth line replaces the diagonal of A with 1s.

```
matrix A = { 1, 2, 3; 4, 5, 6; 7, 8, 9 }
matrix A[2:3,2:3] = I(2)
matrix d = { 1, 1, 1 }
matrix A[diag] = d
```

16.7 Namespace issues

Matrices share a common namespace with data series and scalar variables. In other words, no two objects of any of these types can have the same name. In case of potential collisions — where an object of one type already exists with a certain name, and you try to create an object of a different type with the same name — gretl follows the policy of allowing you to overwrite the existing object, with the exception that *data series are protected and cannot be over-written by scalars or matrices*. Some implications of this policy are noted below.

- If a series called, say, X, exists and you try to create a matrix named X, an error is flagged.
- If you create a series named X — using the `genr` or `series` commands, or by reading from a data file, or by importation from a database — then any pre-existing matrix named X is automatically deleted.
- If you create a scalar named X, any existing matrix X is deleted.

If you really want to create a matrix using a name that is currently assigned to a data series, you must first delete the data series using the `delete` command or rename it using `rename`.

16.8 Creating a data series from a matrix

Section 16.2 above describes how to create a matrix from a data series or set of series. You may sometimes wish to go in the opposite direction, that is, to copy values from a matrix into a regular data series. The syntax for this operation is

`series` *sname* `=` *mspec*

where *sname* is the name of the series to create and *mspec* is the name of the matrix to copy from, possibly followed by a matrix selection expression. Here are two examples.

```
series s = x
series u1 = U[,1]
```

It is assumed that `x` and `U` are pre-existing matrices. In the second example the series `u1` is formed from the first column of the matrix `U`.

For this operation to work, the matrix (or matrix selection) must be a vector with length equal to either the full length of the current dataset, n, or the length of the current sample range, n'. If $n' < n$ then only n' elements are drawn from the matrix; if the matrix or selection comprises n elements, the n' values starting at element t_1 are used, where t_1 represents the starting observation of the sample range. Any values in the series that are not assigned from the matrix are set to the missing code.

Please note that when forming a series in this way, the right-hand side of the `series` command can be *only* the name of a matrix, or the name of a matrix plus a selection expression. There is no provision for matrix calculation in this context.

16.9 Deleting matrices

To delete a matrix, use the syntax

```
matrix A delete
```

where A is the name of the matrix to be deleted.

16.10 Further points and example

Example 16.1 shows how matrix methods can be used to replicate gretl's built-in OLS functionality. The example illustrates various additional points.

First, if you just write `matrix A`, where a matrix A is already defined, the effect is to print the matrix.

Second, there is some "cross over" between matrix expressions and `genr` (actually the synonym `scalar` is used in the script). In a `genr` formula, you can use matrix functions that produce scalar results (e.g. `rows()`). You can also reference 1×1 matrices as if they were ordinary scalars. And in a `matrix` formula you can reference scalar variables where appropriate.

Note, however, that ordinary data series cannot be used in `matrix` expressions, other than in the special case of defining a matrix from a list of series as in section 16.2 above. Similarly, matrices larger than 1×1 cannot be used in the generation of a data series, other than as described in section 16.8.

Example 16.1: OLS via matrix methods

```
open data4-1
matrix X = { const, sqft }
matrix y = { price }
matrix b = inv(X'*X) * X'*y
printf "estimated coefficient vector\n"
matrix b
matrix uh = y - X*b
scalar SSR = uh'*uh
scalar s2 = SSR / (rows(X) - rows(b))
matrix V = s2 * inv(X'*X)
matrix V
matrix se = sqrt(diag(V))
printf "estimated standard errors\n"
matrix se
# compare with built-in function
ols price const sqft --vcv
```

Chapter 17

Troubleshooting gretl

17.1 Bug reports

Bug reports are welcome. Hopefully, you are unlikely to find bugs in the actual calculations done by gretl (although this statement does not constitute any sort of warranty). You may, however, come across bugs or oddities in the behavior of the graphical interface. Please remember that the usefulness of bug reports is greatly enhanced if you can be as specific as possible: what *exactly* went wrong, under what conditions, and on what operating system? If you saw an error message, what precisely did it say?

17.2 Auxiliary programs

As mentioned above, gretl calls some other programs to accomplish certain tasks (gnuplot for graphing, LaTeX for high-quality typesetting of regression output, GNU R). If something goes wrong with such external links, it is not always easy for gretl to produce an informative error message. If such a link fails when accessed from the gretl graphical interface, you may be able to get more information by starting gretl from the command prompt rather than via a desktop menu entry or icon. On the X window system, start gretl from the shell prompt in an `xterm`; on MS Windows, start the program `gretlw32.exe` from a console window or "DOS box". Additional error messages may be displayed on the terminal window.

Also please note that for most external calls, gretl assumes that the programs in question are available in your "path" — that is, that they can be invoked simply via the name of the program, without supplying the program's full location.[1] Thus if a given program fails, try the experiment of typing the program name at the command prompt, as shown below.

	Graphing	*Typesetting*	*GNU R*
X window system	gnuplot	latex, xdvi	R
MS Windows	wgnuplot.exe	latex, windvi	RGui.exe

If the program fails to start from the prompt, it's not a gretl issue but rather that the program's home directory is not in your path, or the program is not installed (properly). For details on modifying your path please see the documentation or online help for your operating system or shell.

[1] The exception to this rule is the invocation of gnuplot under MS Windows, where a full path to the program is given.

Chapter 18

The command line interface

18.1 Gretl at the console

The gretl package includes the command-line program gretlcli. On Linux it can be run from the console, or in an xterm (or similar). Under MS Windows it can be run in a console window (sometimes inaccurately called a "DOS box"). gretlcli has its own help file, which may be accessed by typing "help" at the prompt. It can be run in batch mode, sending outout directly to a file (see also the *Gretl Command Reference*).

If gretlcli is linked to the readline library (this is automatically the case in the MS Windows version; also see Appendix B), the command line is recallable and editable, and offers command completion. You can use the Up and Down arrow keys to cycle through previously typed commands. On a given command line, you can use the arrow keys to move around, in conjunction with Emacs editing keystokes.[1] The most common of these are:

Keystroke	*Effect*
`Ctrl-a`	go to start of line
`Ctrl-e`	go to end of line
`Ctrl-d`	delete character to right

where "`Ctrl-a`" means press the "`a`" key while the "`Ctrl`" key is also depressed. Thus if you want to change something at the beginning of a command, you *don't* have to backspace over the whole line, erasing as you go. Just hop to the start and add or delete characters. If you type the first letters of a command name then press the Tab key, readline will attempt to complete the command name for you. If there's a unique completion it will be put in place automatically. If there's more than one completion, pressing Tab a second time brings up a list.

18.2 Changes from Ramanathan's ESL

gretlcli inherits its basic command syntax from Ramu Ramanathan's ESL, and command scripts developed for ESL should be usable with few or no changes: the only things to watch for are multi-line commands and the `freq` command.

- In ESL, a semicolon is used as a terminator for many commands. I decided to remove this in gretlcli. Semicolons are simply ignored, apart from a few special cases where they have a definite meaning: as a separator for two lists in the `ar` and `tsls` commands, and as a marker for an unchanged starting or ending observation in the `smpl` command. In ESL semicolon termination gives the possibility of breaking long commands over more than one line; in gretlcli this is done by putting a trailing backslash \ at the end of a line that is to be continued.
- With `freq`, you can't at present specify user-defined ranges as in ESL. A chi-square test for normality has been added to the output of this command.

Note also that the command-line syntax for running a batch job is simplified. For ESL you typed, e.g.

[1] Actually, the key bindings shown below are only the defaults; they can be customized. See the readline manual.

```
esl -b datafile < inputfile > outputfile
```

while for gretlcli you type:

```
gretlcli -b inputfile > outputfile
```

The inputfile is treated as a program argument; it should specify a datafile to use internally, using the syntax `open datafile` or the special comment `(* ! datafile *)`

Appendix A

Data file details

A.1 Basic native format

In gretl's native data format, a data set is stored in XML (extensible mark-up language). Data files correspond to the simple DTD (document type definition) given in `gretldata.dtd`, which is supplied with the gretl distribution and is installed in the system data directory (e.g. `/usr/share/gretl/data` on Linux.) Data files may be plain text or gzipped. They contain the actual data values plus additional information such as the names and descriptions of variables, the frequency of the data, and so on.

Most users will probably not have need to read or write such files other than via gretl itself, but if you want to manipulate them using other software tools you should examine the DTD and also take a look at a few of the supplied practice data files: `data4-1.gdt` gives a simple example; `data4-10.gdt` is an example where observation labels are included.

A.2 Traditional ESL format

For backward compatibility, gretl can also handle data files in the "traditional" format inherited from Ramanathan's ESL program. In this format (which was the default in gretl prior to version 0.98) a data set is represented by two files. One contains the actual data and the other information on how the data should be read. To be more specific:

1. *Actual data*: A rectangular matrix of white-space separated numbers. Each column represents a variable, each row an observation on each of the variables (spreadsheet style). Data columns can be separated by spaces or tabs. The filename should have the suffix `.gdt`. By default the data file is ASCII (plain text). Optionally it can be gzip-compressed to save disk space. You can insert comments into a data file: if a line begins with the hash mark (#) the entire line is ignored. This is consistent with gnuplot and octave data files.
2. *Header*: The data file must be accompanied by a header file which has the same basename as the data file plus the suffix `.hdr`. This file contains, in order:
   - (Optional) *comments* on the data, set off by the opening string `(*` and the closing string `*)`, each of these strings to occur on lines by themselves.
   - (Required) list of white-space separated *names of the variables* in the data file. Names are limited to 8 characters, must start with a letter, and are limited to alphanumeric characters plus the underscore. The list may continue over more than one line; it is terminated with a semicolon, `;`.
   - (Required) *observations* line of the form `1 1 85`. The first element gives the data frequency (1 for undated or annual data, 4 for quarterly, 12 for monthly). The second and third elements give the starting and ending observations. Generally these will be 1 and the number of observations respectively, for undated data. For time-series data one can use dates of the form `1959.1` (quarterly, one digit after the point) or `1967.03` (monthly, two digits after the point). See Chapter 6 for special use of this line in the case of panel data.
   - The keyword `BYOBS`.

Here is an example of a well-formed data header file.

```
(*
  DATA9-6:
  Data on log(money), log(income) and interest rate from US.
  Source: Stock and Watson (1993) Econometrica
  (unsmoothed data) Period is 1900-1989 (annual data).
  Data compiled by Graham Elliott.
*)
lmoney lincome intrate ;
1 1900 1989 BYOBS
```

The corresponding data file contains three columns of data, each having 90 entries. Three further features of the "traditional" data format may be noted.

1. If the BYOBS keyword is replaced by BYVAR, and followed by the keyword BINARY, this indicates that the corresponding data file is in binary format. Such data files can be written from gretlcli using the `store` command with the `-s` flag (single precision) or the `-o` flag (double precision).
2. If BYOBS is followed by the keyword MARKERS, gretl expects a data file in which the *first column* contains strings (8 characters maximum) used to identify the observations. This may be handy in the case of cross-sectional data where the units of observation are identifiable: countries, states, cities or whatever. It can also be useful for irregular time series data, such as daily stock price data where some days are not trading days — in this case the observations can be marked with a date string such as 10/01/98. (Remember the 8-character maximum.) Note that BINARY and MARKERS are mutually exclusive flags. Also note that the "markers" are not considered to be a variable: this column does not have a corresponding entry in the list of variable names in the header file.
3. If a file with the same base name as the data file and header files, but with the suffix `.lbl`, is found, it is read to fill out the descriptive labels for the data series. The format of the label file is simple: each line contains the name of one variable (as found in the header file), followed by one or more spaces, followed by the descriptive label. Here is an example: `price New car price index, 1982 base year`

If you want to save data in traditional format, use the `-t` flag with the `store` command, either in the command-line program or in the console window of the GUI program.

A.3 Binary database details

A gretl database consists of two parts: an ASCII index file (with filename suffix `.idx`) containing information on the series, and a binary file (suffix `.bin`) containing the actual data. Two examples of the format for an entry in the `idx` file are shown below:

```
GOM910  Composite index of 11 leading indicators (1987=100)
M 1948.01 - 1995.11  n = 575
currbal Balance of Payments: Balance on Current Account; SA
Q 1960.1 - 1999.4 n = 160
```

The first field is the series name. The second is a description of the series (maximum 128 characters). On the second line the first field is a frequency code: `M` for monthly, `Q` for quarterly, `A` for annual, `B` for business-daily (daily with five days per week) and `D` for daily (seven days per week). No other frequencies are accepted at present. Then comes the starting date (N.B. with two digits following the point for monthly data, one for quarterly data, none for annual), a space, a hyphen, another space, the ending date, the string "`n = `" and the integer number of observations. In the

case of daily data the starting and ending dates should be given in the form YYYY/MM/DD. This format must be respected exactly.

Optionally, the first line of the index file may contain a short comment (up to 64 characters) on the source and nature of the data, following a hash mark. For example:

```
# Federal Reserve Board (interest rates)
```

The corresponding binary database file holds the data values, represented as "floats", that is, single-precision floating-point numbers, typically taking four bytes apiece. The numbers are packed "by variable", so that the first *n* numbers are the observations of variable 1, the next *m* the observations on variable 2, and so on.

Appendix B

Technical notes

Gretl is written in the C programming language, abiding as far as possible by the ISO/ANSI C Standard (C90) although the graphical user interface and some other components necessarily make use of platform-specific extensions.

The program was developed under Linux. The shared library and command-line client should compile and run on any platform that (a) supports ISO/ANSI C, and (b) has the following libraries installed: zlib (data compression), libxml (XML manipulation), and LAPACK (linear algebra support). The homepage for zlib can be found at info-zip.org; libxml is at xmlsoft.org; LAPACK is at netlib.org. If the GNU readline library is found on the host system this will be used for gretcli, providing a much enhanced editable command line. See the readline homepage.

The graphical client program should compile and run on any system that, in addition to the above requirements, offers GTK version 1.2.3 or higher (see gtk.org). As of this writing there are two main variants of the GTK libraries: the 1.2 series and the 2.0 series which was launched in summer 2002. These variants are mutually incompatible. gretl can be built using either one — the source code package includes two sub-directories, `gui` for GTK 1.2 and `gui2` for GTK 2.0. Use of GTK 2.0 is recommended, since it offers many enhancements over GTK 1.2.

Gretl calls gnuplot for graphing. You can find gnuplot at gnuplot.info. As of this writing the most recent official release is 4.0 (of April, 2004). The MS Windows version of gretl comes with a Windows version gnuplot 4.0; the gretl website also offers an rpm of gnuplot 3.8j0 for x86 Linux systems.

Some features of gretl make use of Adrian Feguin's gtkextra library. You can find gtkextra at gtkextra.sourceforge.net. The relevant parts of this package are included (in slightly modified form) with the gretl source distribution.

A binary version of the program is available for the Microsoft Windows platform (32-bit version, i.e. Windows 95 or higher). This version was cross-compiled under Linux using mingw (the GNU C compiler, gcc, ported for use with win32) and linked against the Microsoft C library, `msvcrt.dll`. It uses Tor Lillqvist's port of GTK 2.0 to win32. The (free, open-source) Windows installer program is courtesy of Jordan Russell (jrsoftware.org).

We're hopeful that some users with coding skills may consider gretl sufficiently interesting to be worth improving and extending. The documentation of the libgretl API is by no means complete, but you can find some details by following the link "Libgretl API docs" on the gretl homepage.

Appendix C

Numerical accuracy

Gretl uses double-precision arithmetic throughout — except for the multiple-precision plugin invoked by the menu item "Model/High precision OLS" which represents floating-point values using a number of bits given by the environment variable GRETL_MP_BITS (default value 256). The normal equations of Least Squares are by default solved via Cholesky decomposition, which is accurate enough for most purposes (with the option of using QR decomposition instead). The program has been tested rather thoroughly on the statistical reference datasets provided by NIST (the U.S. National Institute of Standards and Technology) and a full account of the results may be found on the gretl website (follow the link "Numerical accuracy").

Giovanni Baiocchi and Walter Distaso published a review of gretl in the *Journal of Applied Econometrics* (2003). We are grateful to Baiocchi and Distaso for their careful examination of the program, which prompted the following modifications.

1. The reviewers pointed out that there was a bug in gretl's "p-value finder", whereby the program printed the complement of the correct probability for negative values of z. This was fixed in version 0.998 of the program (released July 9, 2002).

2. They also noted that the p-value finder produced inaccurate results for extreme values of x (e.g. values of around 8 to 10 in the t distribution with 100 degrees of freedom). This too was fixed in gretl version 0.998, with a switch to more accurate probability distribution code.

3. The reviewers noted a flaw in the presentation of regression coefficients in gretl, whereby some coefficients could be printed to an unacceptably small number of significant figures. This was fixed in version 0.999 (released August 25, 2002): now all the statistics associated with a regression are printed to 6 significant figures.

4. It transpired from the reviewer's tests that the numerical accuracy of gretl on MS Windows was less than on Linux. For example, on the Longley data — a well-known "ill-conditioned" dataset often used for testing econometrics programs — the Windows version of gretl was getting some coefficients wrong at the 7th digit while the same coefficients were correct on Linux. This anomaly was fixed in gretl version 1.0pre3 (released October 10, 2002).

The current version of gretl includes a "plugin" that runs the NIST linear regression test suite. You can find this under the "Utilities" menu in the main window. When you run this test, the introductory text explains the expected result. If you run this test and see anything other than the expected result, please send a bug report to cottrell@wfu.edu.

As mentioned above, all regression statistics are printed to 6 significant figures in the current version of gretl (except when the multiple-precision plugin is used, then results are given to 12 figures). If you want to examine a particular value more closely, first save it (for example, using the genr command) then print it using `print --ten` (see the *Gretl Command Reference*). This will show the value to 10 digits.

Appendix D

Advanced econometric analysis with free software

Gretl offers a reasonably full (and expanding) selection of estimators, and also offers various scripting constructs that may be useful in the creation of additional estimators. However, its scripting capabilities and its support for the creation and manipulation of matrices are not quite adequate for large-scale, complex computational tasks. If you are looking for this functionality in the realm of free, open-source software we recommend taking a look at either GNU R (r-project.org) or GNU Octave (www.octave.org). These programs are very close to the commercial programs S and Matlab respectively.

As mentioned above, gretl offers the facility of exporting data in the formats of both Octave and R. In the case of Octave, the gretl data set is saved as a single matrix, X. You can pull the X matrix apart if you wish, once the data are loaded in Octave; see the Octave manual for details. As for R, the exported data file preserves any time series structure that is apparent to gretl. The series are saved as individual structures. The data should be brought into R using the `source()` command.

Of these two programs, R is perhaps more likely to be of immediate interest to econometricians since it offers more in the way of specialized statistical routines. Gretl therefore has a convenience function for moving data quickly into R. Under gretl's Session menu, you will find the entry "Start GNU R". This writes out an R version of the current gretl data set (`Rdata.tmp`, in the user's gretl directory), and sources it into a new R session. A few details on this follow.

First, the data are brought into R by writing a temporary version of `.Rprofile` in the current working directory. (If such a file exists it is referenced by R at startup.) In case you already have a personal `.Rprofile` in place, the original file is temporarily moved to `.Rprofile.gretltmp`, and on exit from gretl it is restored. (If anyone can suggest a cleaner way of doing this I'd be happy to hear of it.)

Second, the particular way R is invoked depends on the internal gretl variable `Rcommand`, whose value may be set under the File, Preferences menu. The default command is `RGui.exe` under MS Windows. Under X it is `xterm -e R`. Please note that at most three space-separated elements in this command string will be processed; any extra elements are ignored.

Appendix E

Listing of URLs

Below is a listing of the full URLs of websites mentioned in the text.

Estima (RATS) http://www.estima.com/

Gnome desktop homepage http://www.gnome.org/

GNU Multiple Precision (GMP) library http://swox.com/gmp/

GNU Octave homepage http://www.octave.org/

GNU R homepage http://www.r-project.org/

GNU R manual http://cran.r-project.org/doc/manuals/R-intro.pdf

Gnuplot homepage http://www.gnuplot.info/

Gnuplot manual http://ricardo.ecn.wfu.edu/gnuplot.html

Gretl data page http://gretl.sourceforge.net/gretl_data.html

Gretl homepage http://gretl.sourceforge.net/

GTK+ homepage http://www.gtk.org/

GTK+ port for win32 http://user.sgic.fi/~tml/gimp/win32/

Gtkextra homepage http://gtkextra.sourceforge.net/

InfoZip homepage http://www.info-zip.org/pub/infozip/zlib/

JRSoftware http://www.jrsoftware.org/

Mingw (gcc for win32) homepage http://www.mingw.org/

Minpack http://www.netlib.org/minpack/

Penn World Table http://pwt.econ.upenn.edu/

Readline homepage http://cnswww.cns.cwru.edu/~chet/readline/rltop.html

Readline manual http://cnswww.cns.cwru.edu/~chet/readline/readline.html

Xmlsoft homepage http://xmlsoft.org/

Bibliography

Akaike, H. (1974) "A New Look at the Statistical Model Identification", *IEEE Transactions on Automatic Control*, AC-19, pp. 716–23.

Baiocchi, G. and Distaso, W. (2003) "GRETL: Econometric software for the GNU generation", *Journal of Applied Econometrics*, 18, pp. 105–10.

Baxter, M. and King, R. G. (1995) "Measuring Business Cycles: Approximate Band-Pass Filters for Economic Time Series", National Bureau of Economic Research, Working Paper No. 5022.

Belsley, D., Kuh, E. and Welsch, R. (1980) *Regression Diagnostics*, New York: Wiley.

Berndt, E., Hall, B., Hall, R. and Hausman, J. (1974) "Estimation and Inference in Nonlinear Structural Models", *Annals of Economic and Social Measurement*, 3/4, pp. 653–65.

Box, G. E. P. and Muller, M. E. (1958) "A Note on the Generation of Random Normal Deviates", *Annals of Mathematical Statistics*, 29, pp. 610–11.

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

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

Doornik, J. A. and Hansen, H. (1994) "An Omnibus Test for Univariate and Multivariate Normality", working paper, Nuffield College, Oxford.

Doornik, J. A. (1998) "Approximations to the Asymptotic Distribution of Cointegration Tests", *Journal of Economic Surveys*, 12, pp. 573–93. Reprinted with corrections in M. McAleer and L. Oxley (1999) *Practical Issues in Cointegration Analysis*, Oxford: Blackwell.

Fiorentini, G., Calzolari, G. and Panattoni, L. (1996) "Analytic Derivatives and the Computation of GARCH Etimates", *Journal of Applied Econometrics*, 11, pp. 399–417.

Greene, William H. (2000) *Econometric Analysis*, 4th edition, Upper Saddle River, NJ: Prentice-Hall.

Gujarati, Damodar N. (2003) *Basic Econometrics*, 4th edition, Boston, MA: McGraw-Hill.

Hamilton, James D. (1994) *Time Series Analysis*, Princeton, NJ: Princeton University Press.

Hannan, E. J. and B. G. Quinn (1979) "The Determination of the Order of an Autoregression", *Journal of the Royal Statistical Society*, B, 41, pp. 190–195.

Hodrick, Robert and Edward C. Prescott (1997) "Postwar U.S. Business Cycles: An Empirical Investigation", *Journal of Money, Credit and Banking*, 29, pp. 1–16.

Johansen, Søren (1995) *Likelihood-Based Inference in Cointegrated Vector Autoregressive Models*, Oxford: Oxford University Press.

Kiviet, J. F. (1986) "On the Rigour of Some Misspecification Tests for Modelling Dynamic Relationships", *Review of Economic Studies*, 53, pp. 241–261.

Kwiatkowski, D., Phillips, P. C. B., Schmidt, P. and Shin, Y. (1992) "Testing the Null of Stationarity Against the Alternative of a Unit Root: How Sure Are We That Economic Time Series Have a Unit Root?", *Journal of Econometrics*, 54, pp. 159–178.

Locke, C. (1976) "A Test for the Composite Hypothesis that a Population has a Gamma Distribution", *Communications in Statistics — Theory and Methods*, A5(4), pp. 351–364.

Lucchetti, R., Papi, L., and Zazzaro, A. (2001) "Banks' Inefficiency and Economic Growth: A Micro Macro Approach", *Scottish Journal of Political Economy*, 48, pp. 400–424.

McCullough, B. D. and Renfro, Charles G. (1998) "Benchmarks and software standards: A case study of GARCH procedures", *Journal of Economic and Social Measurement*, 25, pp. 59–71.

MacKinnon, J. G. (1996) “Numerical Distribution Functions for Unit Root and Cointegration Tests”, *Journal of Applied Econometrics*, 11, pp. 601–618.

MacKinnon, J. G. and White, H. (1985) “Some Heteroskedasticity-Consistent Covariance Matrix Estimators with Improved Finite Sample Properties”, *Journal of Econometrics*, 29, pp. 305–25.

Maddala, G. S. (1992) *Introduction to Econometrics*, 2nd edition, Englewood Cliffs, NJ: Prentice-Hall.

Matsumoto, M. and Nishimura, T. (1998) “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator”, *ACM Transactions on Modeling and Computer Simulation*, 8: 1.

Neter, J. Wasserman, W. and Kutner, M. H. (1990) *Applied Linear Statistical Models*, 3rd edition, Boston, MA: Irwin.

R Core Development Team (2000) *An Introduction to R*, version 1.1.1.

Ramanathan, Ramu (2002) *Introductory Econometrics with Applications*, 5th edition, Fort Worth: Harcourt.

Schwarz, G. (1978) “Estimating the dimension of a model”, *Annals of Statistics*, 6, pp. 461–64.

Shapiro, S. and Chen, L. (2001) “Composite Tests for the Gamma Distribution”, *Journal of Quality Technology*, 33, pp. 47–59.

Silverman, B. W. (1986) *Density Estimation for Statistics and Data Analysis*, London: Chapman and Hall.

Stock, James H. and Watson, Mark W. (2003) *Introduction to Econometrics*, Boston, MA: Addison-Wesley.

Wooldridge, Jeffrey M. (2002) *Introductory Econometrics, A Modern Approach*, 2nd edition, Mason, Ohio: South-Western.