Gretl Manual

Gnu Regression, Econometrics and Time-series Library

Allin Cottrell
Department of Economics
Wake Forest University

August, 2005

Gretl Manual: Gnu Regression, Econometrics and Time-series Library
by Allin Cottrell

Table of Contents

List of Tables

Chapter 1. Introduction

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. I have compiled it on Mac OS X and AIX and it 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. The author welcomes any bug reports.

Reasonably sophisticated

gretl offers a full range of least-squares based estimators, including two-stage least squares and nonlinear least squares. It also offers (binomial) logit and probit estimation, and has a loop construct for running Monte Carlo analyses or iterative estimation of non-linear models. While it does not include all the estimators and tests that a professional econometrician might require, it supports the export of data to the formats of (GNU R) and (GNU Octave) for further analysis (see Appendix D).

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 or Polish, depending on your computer's native language setting.

Acknowledgements

My primary debt is to Professor Ramu Ramanathan of the University of California, San Diego. A few years back he was kind enough to provide me with the source code for his program ESL ("Econometrics Software Library"), which I ported to Linux, and since then I have collaborated with him on updating and extending the program. For the gretl project I have made extensive changes to the original ESL code. New econometric functionality has been added, and the graphical interface is entirely new. Please note that Professor Ramanathan is not responsible for any bugs in gretl.

I am 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 I am grateful to the authors for permission to use their work.

With regard to the internationalization of gretl, I wish to thank Ignacio Díaz-Emparanza, Michel Robitaille, Cristian Rigamonti and Tadeusz Kufel, who prepared the Spanish, French, Italian and Polish translations respectively.

I have benefitted greatly from the work of numerous developers of open-source software: for specifics please see Appendix B to this manual. My 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. Besides making good suggestions, Riccardo "Jack" Lucchetti has contributed substantial code and econometric expertise.

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). I am grateful to Dirk Eddelbüttel for making the latter. 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

1. Terminology is a bit of a problem here, but in this manual I will 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.

take a look at Appendix B of this manual.

As of version 0.97 `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`.

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

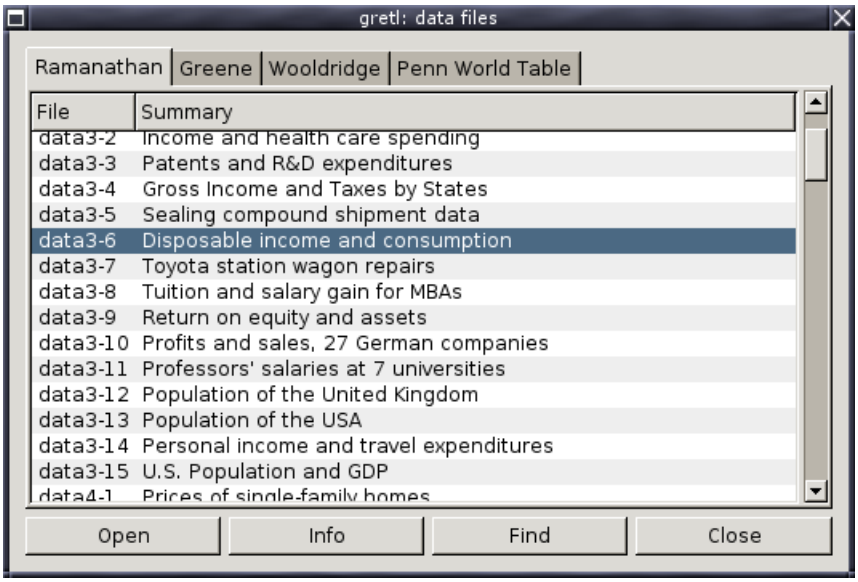
Let's run a regression

This introduction is mostly angled towards the graphical client program; please see Chapter 14 and Chapter 16 below 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`.

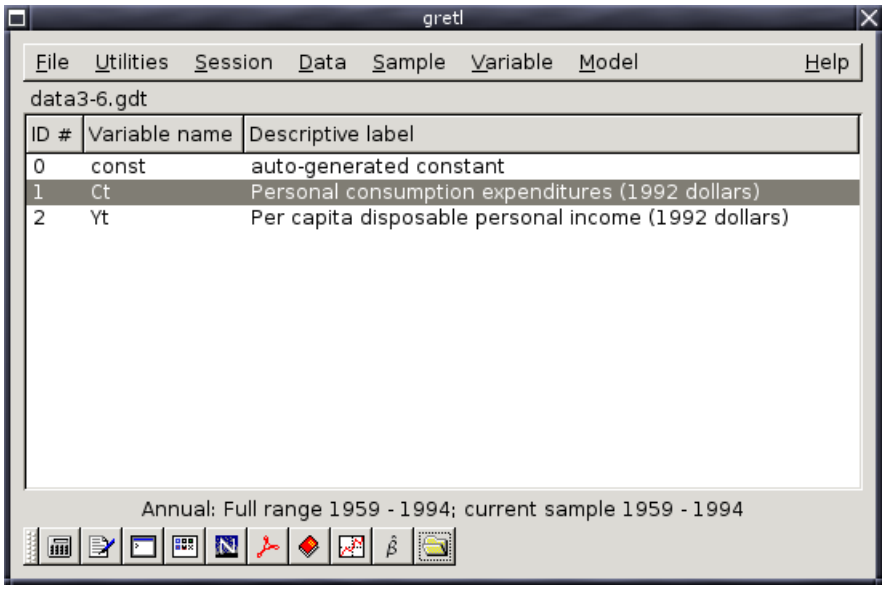
In `gretl` windows containing lists, double-clicking on a line launches a default action for the associated list entry: e.g. displaying the values of a data series, opening a file.

This file contains data pertaining to a classic econometric "chestnut", the consumption function. The data window should now display the name of the current data file, the overall data

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 Chapter 13.

range and sample range, and the names of the variables along with brief descriptive tags — see Figure 2-2.

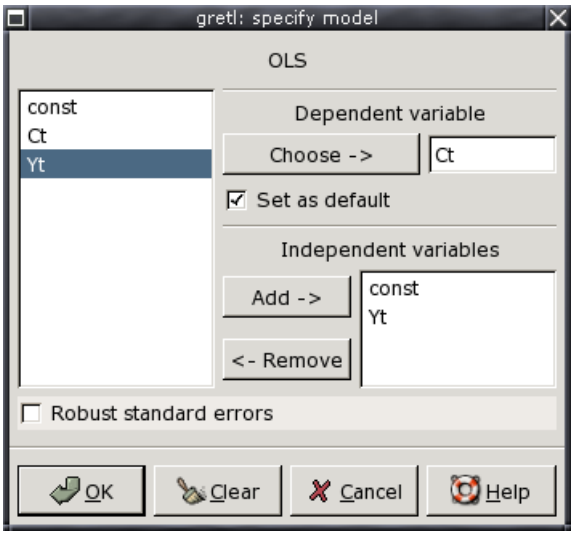
Figure 2-2. Main window, with a practice data file open



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 the Section called *The main window menus* 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



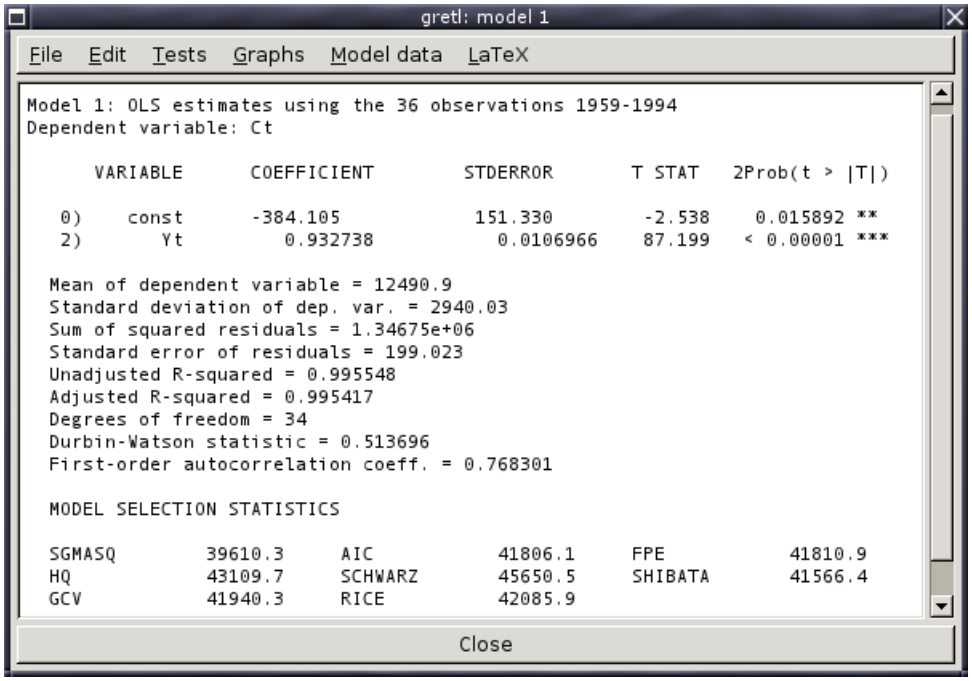
To select the dependent variable, highlight the variable you want in the list on the left and click the "Choose" button that points to the Dependent variable slot. If you check the "Set as default" box this variable will be pre-selected as dependent when you next open the model dialog box. Shortcut: 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.

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 Chapter 13). 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 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.

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.

— 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 (see the Section called *Creating a data file from scratch* in Chapter 4).

— Browse databases: See the Section called *Binary databases* in Chapter 4 and the Section called *Creating a data file from scratch* in Chapter 4.

— Create data set: Initialize the built-in spreadsheet for entering data manually. See the Section called *Creating a data file from scratch* in Chapter 4.

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 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.

— 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 Chapter 13 for details.

— Exit: Quit the program. If expert mode is not selected you'll be prompted to save any unsaved work.

§ Utilities menu

— 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 Chapter 14.

— 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). See Chapter 14.

— 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 the Section called *The Session concept* in Chapter 3.

— 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 7.

— 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).

— 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 help for `genr` in Chapter 14 for details on the Boolean operators that can be used.

— Drop all obs with missing values: Drop from the current sample all observations for which at least one variable has a missing value (see the Section called *Missing data values* in Chapter 4).

— 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 Chapter 14. 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 Section called *Estimators and tests: summary* in Chapter 14 and Chapter 14 below, 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.

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 (the Section called *The main window menus* 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 Jeffrey Wooldridge's text, if these are installed — see Chapter 13).

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

Chapter 3. Modes of working

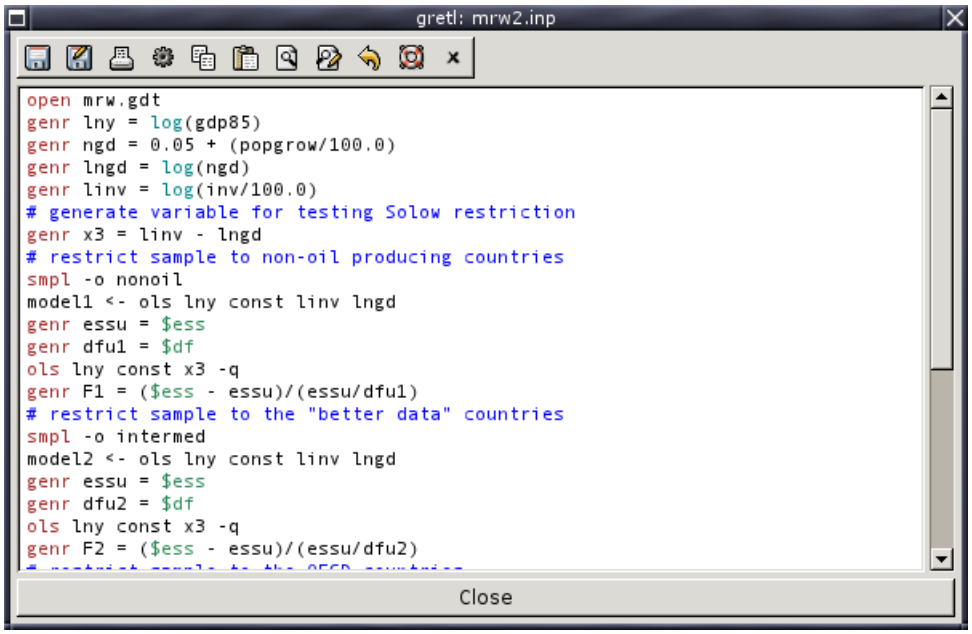
Command scripts

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 Chapter 14 in this manual. In addition, the `gretl` package includes over 70 "practice" scripts. Most of these relate to Ramanathan (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 Chapter 14 below). 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.

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 the Section called *The Session concept*) 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`.

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.

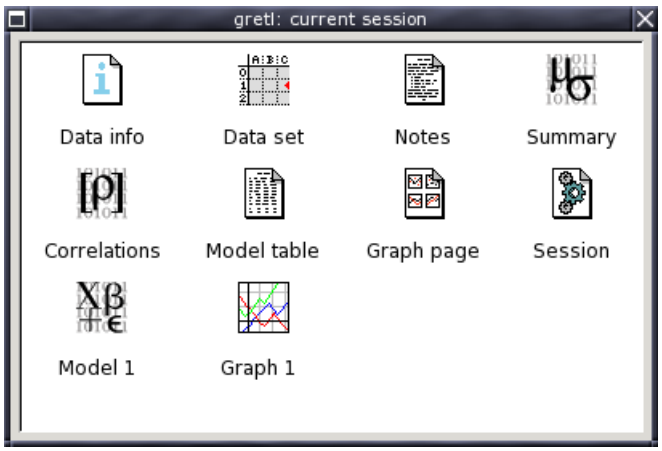
In console mode, you can create, display and free objects (models, graphs or text) aa described above for script mode.

The Session concept

Introduction

`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.

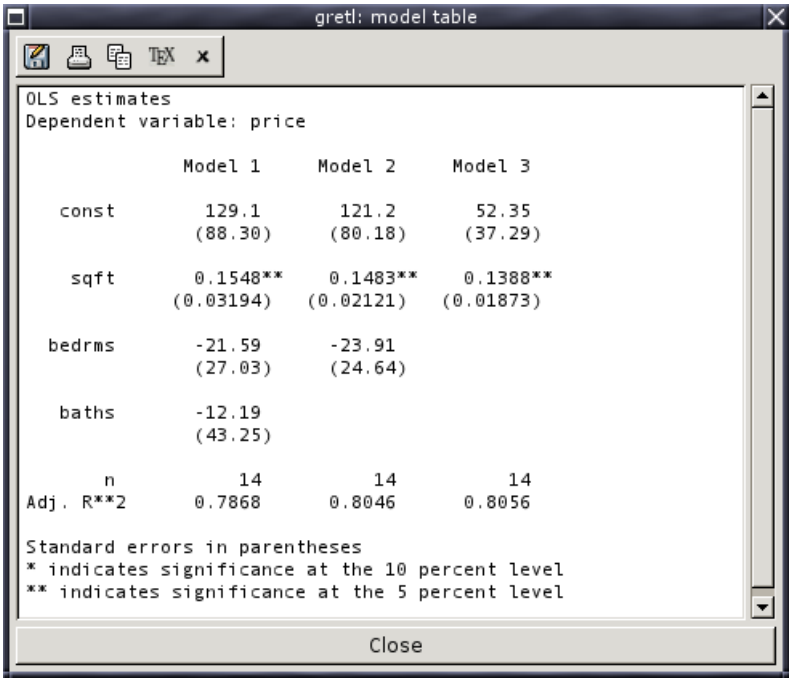
In the Icon View window `gretl` provides a means of constructing such a table (and copying it in plain text, LaTeX or Rich Text Format). Here is how to do it:[1]

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.

1. The model table can also be built non-interactively, in script mode. For details on how to do this, see the modeltab command.

Figure 3-3. Example of model table



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]

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

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`.

§ From the command line, type `gretl -r sessionfile`, where *sessionfile* is the name under which the session was saved.

Chapter 4. Data files

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.

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 the Section called *Creating a data file from scratch*.

§ 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 the Section called *Creating a data file from scratch*.

§ 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 the Section called *Creating a data file from scratch*.

§ BOX1 format data. Large amounts of micro data are available (for free) in this format via the Data Extraction Service of the US Bureau of the Census. BOX1 data may be imported using the "File, Open Data, Import BOX..." menu item or the import -o script command.

When you import data from the ASCII, CSV or BOX 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).

Binary databases

For working with large amounts of data I have supplied gretl 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.

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.

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 8 characters maximum; starts with a letter; and contains nothing but letters, numbers and the underscore character, _. (Longer variable names will be truncated to 8 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 the Section called *Missing data values*. 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.[1] 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

1. See www.estima.com

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 Chapter 4.

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.

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 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.[2]

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.)

Table 4-1. Typical locations for file collections

	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`

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 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,

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

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.inp","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

Introduction

The `genr` command provides a flexible means of defining new variables. It is documented in the command reference chapter of this manual (see genr). 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.

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

To be written.

The Baxter and King filter

Consider the spectral representation of a time series y_t:

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

if we wanted to extract only that 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`.

Resampling and bootstrapping

Another specialized function is the resampling, with replacement, of a series. To be written.

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).

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 genr 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. Sub-sampling a dataset

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.

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).

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".

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.

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.

4. Proceed with your analysis.

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.

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 mostly work in the same way as the corresponding `smpl` variant, except that 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 (the Section called *Restricting the sample*).

Chapter 7. Panel data

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
```

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 `gretl-cli`.

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
```

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 (as described in the previous section), `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.

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, to 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. Provided 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.

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 7-1 below 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 7-1. 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 8. Graphs and plots

Gnuplot graphs

General gnuplot options

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.
- 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".

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.

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.

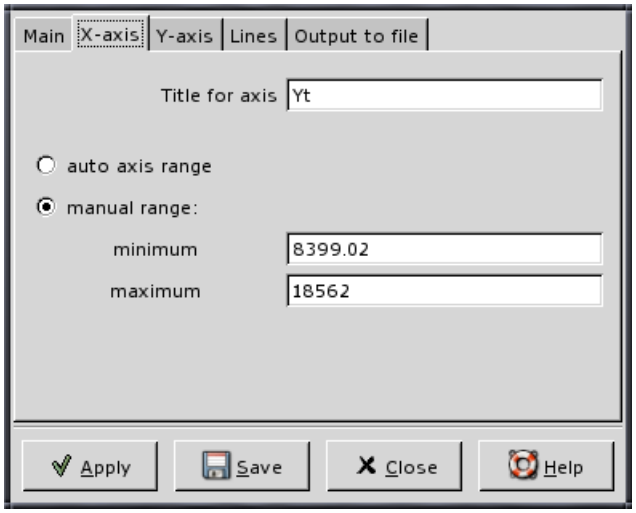
§ Once the graph is saved as a session icon, you can right-click on its icon for a further pop-up menu. One of the otions here is "Edit plot commands", which opens an editing window with the actual gnuplot commands displayed. You can edit these commands and either save them for future processing or send 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 Chapter 14 below — and the `graph` and `plot` commands for "quick and dirty" ASCII graphs.

Figure 8-1. gretl's gnuplot controller



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

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.

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
```

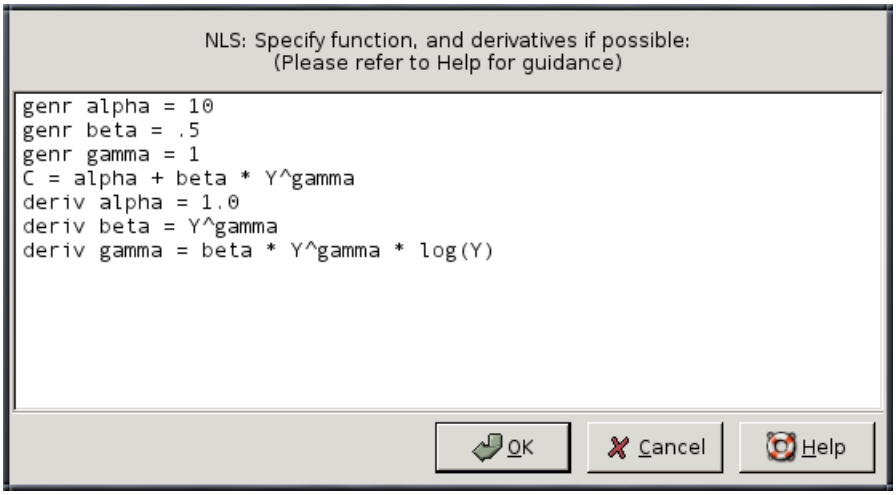
And in relation to the second example one might do:

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

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



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 the Section called *Numerical accuracy*).

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.

Controlling termination

The NLS estimation procedure is an iterative process. Iteration is terminated when a convergence criterion is met or when a set maximum number of iterations is reached, whichever comes first. The maximum number of iterations is `100*(k+1)` when analytical derivatives are given and `200*(k+1)` when numerical derivatives are used, where `k` denotes the number of parameters being estimated. The convergence criterion is that the relative error in the sum of squares, and/or the relative error between the the coefficient vector and the solution, is estimated to be no larger than some small value. This "small value" is by default the machine precision to the power 3/4, but it can be set with the `genr` command using the special variable `toler`. For example

```
genr toler = .0001
```

will relax the tolerance to 0.0001.

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.

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.[1] 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.[2]

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).

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.[3]

1. For a discussion of `gretl`'s accuracy in the estimation of linear models, see Appendix C.
2. 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).
3. For the standard errors, I excluded one outlier from the statistics shown in the table, namely `Lanczos1`. This is an odd

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
Avg. of min. correct figures, parameters	8.120	6.980
Worst of min. correct figures, parameters	4	3
Avg. 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.

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. Loop constructs

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 ]
   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 `--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 the Section called *Progressive mode*).

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.

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 10-4) or you can use its string representation, `$i` (see Example 10-5).

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.

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.

Loop examples

Monte Carlo example

A simple example of a Monte Carlo loop in "progressive" mode is shown in Example 10-1.

Example 10-1. 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
```

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 10-2 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...".

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 10-2. 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 10-3 (kindly contributed by Riccardo "Jack" Lucchetti of Ancona University) 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*.

Example 10-3. ARMA 1, 1

```
open armaloop.gdt

genr c = 0
genr a = 0.1
```

```
genr m = 0.1

genr e = const * 0.0
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
```

Indexed loop examples

Example 10-4 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 10-4. Panel statistics

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

Example 10-5 illustrates string substitution in an indexed loop.

Example 10-5. 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
```

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.

Chapter 11. User-defined functions

Introduction

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

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 10).

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.

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

```
# 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
```

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.

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.

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 the Section called *Defining a function*). Otherwise, if the function body contains invalid commands this will become apparent only when the function is called, and its commands are executed.

Chapter 12. 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 13. Options, arguments and path-searching

gretl

`gretl` (under MS Windows, `gretlw32.exe`)[1]

— Opens the program and waits for user input.

`gretl` *`datafile`*

— Starts the program with the specified datafile in its workspace. The data file may be in native `gretl` format, CSV format, or BOX1 format (see Chapter 4 above). The program will try to detect the format of the file and treat it appropriately. See also the Section called *Path searching* below for path-searching behavior.

`gretl --help` (or `gretl -h`)

— Print a brief summary of usage and exit.

`gretl --version` (or `gretl -v`)

— Print version identification for the program and exit.

`gretl --english` (or `gretl -e`)

— Force use of English instead of translation.

`gretl --run` *`scriptfile`* (or `gretl -r` *`scriptfile`*)

— Start the program and open a window displaying the specified script file, ready to run. See the Section called *Path searching* below for path-searching behavior.

`gretl --db` *`database`* (or `gretl -d` *`database`*)

— Start the program and open a window displaying the specified database. If the database files (the `.bin` file and its accompanying `.idx` file — see the Section called *Binary databases* in Chapter 4) are not in the default system database directory, you must specify the full path.

`gretl --dump` (or `gretl -c`)

— Dump the program's configuration information to a plain text file (the name of the file is printed on standard output). May be useful for trouble-shooting.

Various things in `gretl` are configurable under the "File, Preferences" menu.

§ The base directory for `gretl`'s shared files.

§ The user's base directory for `gretl`-related files.

§ The command to launch `gnuplot`.

§ The command to launch GNU R (see Appendix D).

§ The command with which to view TeX DVI files.

§ The directory in which to start looking for native `gretl` databases.

§ The directory in which to start looking for RATS 4 databases.

§ The host name of the `gretl` database server to access.

§ The IP number and port number of the HTTP proxy server to use when contacting the database server, if applicable (if you're behind a firewall).

§ The calculator and editor programs to launch from the toolbar.

§ The monospaced font to be used in `gretl` screen output.

1. On Linux, a "wrapper" script named `gretl` is installed. This script checks whether the `DISPLAY` environment variable is set; if so, it launches the GUI program, `gretl_x11`, and if not it launches the command-line program, `gretlcli`.

§ The font to be used for menus and other messages. (Note: this item is not present when `gretl` is compiled for the `gnome` desktop, since the choice of fonts is handled centrally by `gnome`.)

There are also some check boxes. Checking the "expert" box quells some warnings that are otherwise issued. Checking "Tell me about gretl updates" makes `gretl` attempt to query the update server at start-up. Unchecking "Show gretl toolbar" turns the icon toolbar off. If your native language setting is not English and the local decimal point character is not the period ("`.`"), unchecking "Use locale setting for decimal point" will make `gretl` use the period regardless.

Finally, there are some binary choices: Under the "Open/Save path" tab you can set where `gretl` looks by default when you go to open or save a file — either the `gretl` user directory or the current working directory. Under the "Data files" tab you can set the default filename suffix for data files. The standard suffix is `.gdt` but if you wish you can set this to `.dat`, which was standard in earlier versions of the program. If you set the default to `.dat` then data files will be saved in the "traditional" format (see Chapter 4). Also under the "Data files" tab you can select the action for the little folder icon on the toolbar: whether it should open a listing of the data files associated with Ramanathan's textbook, or those associated with Wooldridge's text.

Under the "General" tab you may select the algorithm used by `gretl` for calculating least squares estimates. The default is Cholesky decomposition, which is fast, relatively economical in terms of memory requirements, and accurate enough for most purposes. The alternative is QR decomposition, which is computationally more expensive and requires more temporary storage, but which is more accurate. You are unlikely to need the extra accuracy of QR decomposition unless you are dealing with very ill-conditioned data and are concerned with coefficient or standard error values to more than 7 digits of precision.[2]

Settings chosen in this way are handled differently depending on the context. Under MS Windows they are stored in the Windows registry. Under the `gnome` desktop they are stored in `.gnome/gretl` in the user's home directory. Otherwise they are stored in a file named `.gretlrc` in the user's home directory.

gretlcli

`gretlcli`

— Opens the program and waits for user input.

`gretlcli` *`datafile`*

— Starts the program with the specified datafile in its workspace. The data file may be in native `gretl` format, CSV format, or BOX1 format (see Chapter 4). The program will try to detect the format of the file and treat it appropriately. See also the Section called *Path searching* for path-searching behavior.

`gretlcli --help` (or `gretlcli -h`)

— Prints a brief summary of usage.

`gretlcli --version` (or `gretlcli -v`)

— Prints version identification for the program.

`gretlcli --pvalue` (or `gretlcli -p`)

— Starts the program in a mode in which you can interactively determine p-values for various common statistics.

2. The option of using QR decomposition can also be activated by setting the environment variable `GRETL_USE_QR` to any non-NULL value.

`gretlcli --english` (or `gretlcli -e`)

— Force use of English instead of translation.

`gretlcli --run` *`scriptfile`* (or `gretlcli -r` *`scriptfile`*)

— Execute the commands in *`scriptfile`* then hand over input to the command line. See the Section called *Path searching* for path-searching behavior.

`gretlcli --batch` *`scriptfile`* (or `gretlcli -b` *`scriptfile`*)

— Execute the commands in *`scriptfile`* then exit. When using this option you will probably want to redirect output to a file. See the Section called *Path searching* for path-searching behavior.

When using the `--run` and `--batch` options, the script file in question must call for a data file to be opened. This can be done using the `open` command within the script. For backward compatibility with Ramanathan's original `ESL` program another mechanism is offered (`ESL` doesn't have the `open` command). A line of the form:

```
(* ! myfile.gdt *)
```

will (a) cause `gretlcli` to load `myfile.gdt`, but will (b) be ignored as a comment by the original `ESL`. Note the specification carefully: There is exactly one space between the begin comment marker, `(*`, and the `!`; there is exactly one space between the `!` and the name of the data file.

One further kludge enables `gretl` and `gretlcli` to get datafile information from the `ESL` "practice files" included with the `gretl` package. A typical practice file begins like this:

```
(* PS4.1, using data file DATA4-1, for reproducing Table 4.2 *)
```

This algorithm is used: if an input line begins with the comment marker, search it for the string `DATA` (upper case). If this is found, extract the string from the `D` up to the next space or comma, put it into lower case, and treat it as the name of a data file to be opened.

Path searching

When the name of a data file or script file is supplied to `gretl` or `gretlcli` on the command line (see the Section called *gretl* and the Section called *gretlcli*), the file is looked for as follows:

1. "As is". That is, in the current working directory or, if a full path is specified, at the specified location.
2. In the user's gretl directory (see Table 13-1 for the default values).
3. In any immediate sub-directory of the user's gretl directory.
4. In the case of a data file, search continues with the main `gretl` data directory. In the case of a script file, the search proceeds to the system script directory. See Table 13-1 for the default settings.
5. In the case of data files the search then proceeds to all immediate sub-directories of the main data directory.

Table 13-1. Default path settings

	Linux	MS Windows
User directory	`$HOME/gretl`	`PREFIX\gretl\user`
System data directory	`PREFIX/share/gretl/data`	`PREFIX\gretl\data`
System script directory	`PREFIX/share/gretl/scripts`	`PREFIX\gretl\scripts`

Note: PREFIX denotes the base directory chosen at the time gretl is installed.

Thus it is not necessary to specify the full path for a data or script file unless you wish to override the automatic searching mechanism. (This also applies within gretlcli, when you supply a filename as an argument to the open or run commands.)

When a command script contains an instruction to open a data file, the search order for the data file is as stated above, except that the directory containing the script is also searched, immediately after trying to find the data file "as is".

MS Windows

Under MS Windows configuration information for gretl and gretlcli is stored in the Windows registry. A suitable set of registry entries is created when gretl is first installed, and the settings can be changed under gretl's "File, Preferences" menu. In case anyone needs to make manual adjustments to this information, the entries can be found (using the standard Windows program regedit.exe) under Software\gretl in HKEY_CLASSES_ROOT (the main gretl directory and the command to invoke gnuplot) and HKEY_CURRENT_USER (all other configurable variables).

Chapter 14. Command Reference

Introduction

The commands defined below may be executed in the command-line client program. They may also be placed in a "script" file for execution in the GUI, or entered using the latter's "console mode". In most cases the syntax given below also applies when you are presented with a line to type in a dialog box in the GUI (but see also `gretl`'s online help), except that you should *not* type the initial command word — it is implicit from the context. One other difference is that some commands support option flags, but you cannot enter these in GUI dialog boxes; generally there are menu items which achieve the same effect.

The option flags shown below are in "long form", with a somewhat mnemonic term preceded by two dashes. There are also corresponding short forms: these are shown in Table 14-4.

The following conventions are used below:

§ A `typewriter font` is used for material that you would type directly, and also for internal names of variables.

§ Terms in *italics* are place-holders: you should substitute something specific, e.g. you might type `income` in place of the generic *xvar*.

§ `The construction [ arg ]` means that the argument *arg* is optional: you may supply it or not (but in any case don't type the brackets).

§ The phrase "estimation command" means a command that generates estimates for a given model, for example `ols`, `ar` or `wls`.

Section and Chapter references below are to Ramanathan (2002).

gretl commands

add

Argument:	*varlist*
Options:	`--vcv` (print covariance matrix)
	`--quiet` (don't print estimates for augmented model)
	`--silent` (don't print anything)
Examples:	`add 5 7 9`
	`add xx yy zz --quiet`

Must be invoked after an estimation command. The variables in *varlist* are added to the previous model and the new model is estimated. A test statistic for the joint significance of the added variables is printed, along with its p-value. The test statistic is F in the case of OLS estimation, an asymptotic Wald chi-square value otherwise. A p-value below 0.05 means that the coefficients are jointly significant at the 5 percent level.

If the `--quiet` option is given the printed results are confined to the test for the joint significance of the added variables, otherwise the estimates for the augmented model are also printed. In the latter case, the `--vcv` flag causes the covariance matrix for the coefficients to be printed also. If the `--silent` option is given, nothing is printed; nonetheless, the results of the test can be retrieved using the special variables `$test` and `$pvalue`.

Menu path: Model window, /Tests/add variables

addobs

Argument:	*nobs*
Example:	`addobs 10`

Adds the specified number of extra observations to the end of the working dataset. This is primarily intended for forecasting purposes. The values of most variables over the additional range will be set to missing, but certain deterministic variables are recognized and extended, namely, a simple linear trend and periodic dummy variables.

This command is not available if the dataset is currently subsampled by selection of cases on some Boolean criterion.

Menu path: /Data/Add observations

addto

Arguments:	*modelID varlist*
Option:	`--quiet` (don't print estimates for augmented model)
Example:	`addto 2 5 7 9`

Works like the `add` command, except that you specify a previous model (using its ID number, which is printed at the start of the model output) to take as the base for adding variables. The example above adds variables number 5, 7 and 9 to Model 2.

Menu path: Model window, /Tests/add variables

adf

Arguments:	*order varname*
Options:	`--nc` (test without a constant)
	`--c` (with constant only)
	`--ct` (with constant and trend)
	`--ctt` (with constant, trend and trend squared)
	`--verbose` (print regression results)
Examples:	`adf 0 y`
	`adf 2 y --nc --c --ct`

Computes statistics for a set of Dickey–Fuller tests on the specified variable, the null hypothesis being that the variable has a unit root.

By default, three variants of the test are shown: one based on a regression containing a constant, one using a constant and linear trend, and one using a constant and a quadratic trend. You can control the variants that are presented by specifying one or more of the option flags.

In all cases the dependent variable is the first difference of the specified variable, y, and the key independent variable is the first lag of y. The model is constructed so that the coefficient on lagged y equals 1 minus the root in question. For example, the model with a constant may be written as

$$(1 - L)y_t = \beta_0 + (1 - \alpha)y_{t-1} + \epsilon_t$$

If the lag order, k, is greater than 0, then k lags of the dependent variable are included on the right-hand side of each test regression.

If the given lag order is prefaced with a minus sign, it is taken as the maximum lag and the actual lag order used is obtained by testing down. Let the given order be –*k*: the testing-down algorithm is then:

1. Estimate the Dickey–Fuller regression with *k* lags of the dependent variable.
2. Is the last lag significant? If so, execute the test with lag order *k*. Otherwise, let $k = k - 1$; if $k = 0$, execute the test with lag order 0, else go to step 1.

In the context of step 2 above, "significant" means that the *t*-statistic for the last lag has an asymptotic two-sided *p*-value, against the normal distribution, of 0.10 or less.

P-values for the Dickey–Fuller tests are based on MacKinnon (1996). The relevant code is included by kind permission of the author.

Menu path: /Variable/Augmented Dickey-Fuller test

append

Argument: *datafile*

Opens a data file and appends the content to the current dataset, if the new data are compatible. The program will try to detect the format of the data file (native, plain text, CSV or BOX1).

Menu path: /File/Append data

ar

Arguments: *lags* ; *depvar indepvars*

Option: `--vcv` (print covariance matrix)

Example: `ar 1 3 4 ; y 0 x1 x2 x3`

Computes parameter estimates using the generalized Cochrane–Orcutt iterative procedure (see Section 9.5 of Ramanathan). Iteration is terminated when successive error sums of squares do not differ by more than 0.005 percent or after 20 iterations.

lags is a list of lags in the residuals, terminated by a semicolon. In the above example, the error term is specified as

$$u_t = \rho_1 u_{t-1} + \rho_3 u_{t-3} + \rho_4 u_{t-4} + e_t$$

Menu path: /Model/Time series/Autoregressive estimation

arch

Arguments: *order depvar indepvars*

Example: `arch 4 y 0 x1 x2 x3`

Tests the model for ARCH (Autoregressive Conditional Heteroskedasticity) of the specified lag order. If the LM test statistic has a p-value below 0.10, then ARCH estimation is also carried out. If the predicted variance of any observation in the auxiliary regression is not positive, then the corresponding squared residual is used instead. Weighted least squares estimation is then performed on the original model.

See also garch.

Menu path: Model window, /Tests/ARCH

arma

Arguments:	*p q* ; [*P Q* ;] *depvar* [*indepvars*]
Options:	`--native` (Use native plugin (default))
	`--x-12-arima` (use X-12-ARIMA for estimation)
	`--verbose` (print details of iterations)
	`--vcv` (print covariance matrix)
	`--nc` (do not include a constant)
Examples:	`arma 1 2 ; y`
	`arma 2 2 ; y 0 x1 x2 --verbose`
	`arma 1 1 ; 1 0 ; y 0 x1 x2`

If no *indepvars* list is given, estimates a univariate ARMA (Autoregressive, Moving Average) model. The integer values *p* and *q* represent the AR and MA orders respectively. The optional integer values *P* and *Q* represent seasonal AR and MA orders; these are relevant only if the data have a frequency greater than 1 (for example, quarterly or monthly data).

In the univariate case the default is to include an intercept in the model but this can be suppressed with the `--nc` flag. If *indepvars* are added, the model becomes ARMAX; in this case the constant should be included explicitly if you want an intercept (as in the second example above).

The default is to use the "native" gretl ARMA function; in the case of a univariate ARMA model `X-12-ARIMA` may be used instead (if the `X-12-ARIMA` package for gretl is installed).

The options given above may be combined, except that the covariance matrix is not available when estimation is by `X-12-ARIMA`.

The native gretl ARMA algorithm is largely due to Riccardo "Jack" Lucchetti. It uses a conditional maximum likelihood procedure, implemented via iterated least squares estimation of the outer product of the gradient (OPG) regression. See Example 10-3 for the logic of the procedure. The AR coefficients (and those for any additional regressors) are initialized using an OLS auto-regression, and the MA coefficients are initialized at zero.

The AIC value given in connection with ARMA models is calculated according to the definition used in `X-12-ARIMA`, namely

$$\text{AIC} = -2L + 2k$$

where L is the log-likelihood and k is the total number of parameters estimated. The "frequency" figure printed in connection with AR and MA roots is the λ value that solves

$$z = re^{i2\pi\lambda}$$

where z is the root in question and r is its modulus.

Menu path: /Variable/ARMA model, /Model/Time series/ARMAX

Other access: Main window pop-up menu (single selection)

boxplot

Argument:	*varlist*
Option:	`--notches` (show 90 percent interval for median)

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 for the median. This is obtained by the bootstrap method.

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 *varlist*:

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

Some details of gretl's boxplots can be controlled via a (plain text) file named `.boxplotrc`. For details on this see the Section called *Boxplots* in Chapter 8.

Menu path: /Data/Graph specified vars/Boxplots

break

Break out of a loop. This command can be used only within a loop; it causes command execution to break out of the current (innermost) loop. See also loop.

chow

Argument:	*obs*
Examples:	`chow 25`
	`chow 1988:1`

Must follow an OLS regression. Creates a dummy variable which equals 1 from the split point specified by *obs* to the end of the sample, 0 otherwise, and also creates interaction terms between this dummy and the original independent variables. An augmented regression is run including these terms and an F statistic is calculated, taking the augmented regression as the unrestricted and the original as restricted. This statistic is appropriate for testing the null hypothesis of no structural break at the given split point.

Menu path: Model window, /Tests/Chow test

coeffsum

Argument:	*varlist*
Example:	`coeffsum xt xt_1 xr_2`

Must follow a regression. Calculates the sum of the coefficients on the variables in *varlist*. Prints this sum along with its standard error and the p-value for the null hypothesis that the sum is zero.

Note the difference between this and omit, which tests the null hypothesis that the coefficients on a specified subset of independent variables are *all* equal to zero.

Menu path: Model window, /Tests/sum of coefficients

coint

Arguments:	*order depvar indepvars*
Option:	`--nc` (do not include a constant)
Example:	`coint 4 y x1 x2`

The Engle–Granger cointegration test. Carries out Augmented Dickey–Fuller tests on the null hypothesis that each of the variables listed has a unit root, using the given lag order. The cointegrating regression is estimated, and an ADF test is run on the residuals from this regression. The Durbin–Watson statistic for the cointegrating regression is also given.

P-values for this test are based on MacKinnon (1996). The relevant code is included by kind permission of the author.

By default, the cointegrating regression contains a constant. If you wish to suppress the constant, add the `--nc` flag.

Menu path: /Model/Time series/Cointegration test/Engle-Granger

coint2

Arguments:	*order depvar indepvars*
Options:	`--nc` (no constant)
	`--rc` (restricted constant)
	`--crt` (constant and restricted trend)
	`--ct` (constant and unrestricted trend)
	`--seasonals` (include centered seasonal dummies)
	`--verbose` (print details of auxiliary regressions)
Examples:	`coint2 2 y x`
	`coint2 4 y x1 x2 --verbose`
	`coint2 3 y x1 x2 --rc`

Carries out the Johansen test for cointegration among the listed variables for the given lag order. Critical values are computed via J. Doornik's gamma approximation (Doornik, 1998). For details of this test see Hamilton, *Time Series Analysis* (1994), Chapter 20.

The inclusion of deterministic terms in the model is controlled by the option flags. The default if no option is specified is to include an "unrestricted constant", which allows for the presence of a non-zero intercept in the cointegrating relations as well as a trend in the levels of the endogenous variables. In the literature stemming from the work of Johansen (see for example his 1995 book) this is often referred to as "case 3". The first four options given above, which are mutually exclusive, produce cases 1, 2, 4 and 5 respectively. The meaning of these cases and the criteria for selecting a case are explained in Chapter 12.

The `--seasonals` option, which may be combined with any of the other options, specifies the inclusion of a set of centered seasonal dummy variables. This option is available only for quarterly or monthly data.

The following table is offered as a guide to the interpretation of the results shown for the test, for the 3-variable case. `H0` denotes the null hypothesis, `H1` the alternative hypothesis, and `c` the number of cointegrating relations.

```
Rank     Trace test         Lmax test
```

```
          H0      H1           H0      H1
-----------------------------------------
 0       c = 0   c = 3        c = 0   c = 1
 1       c = 1   c = 3        c = 1   c = 2
 2       c = 2   c = 3        c = 2   c = 3
-----------------------------------------
```

See also the vecm command.

Menu path: /Model/Time series/Cointegration test/Johansen

corc

Arguments:	*depvar indepvars*
Option:	--vcv (print covariance matrix)
Example:	`corc 1 0 2 4 6 7`

Computes parameter estimates using the Cochrane–Orcutt iterative procedure (see Section 9.4 of Ramanathan). Iteration is terminated when successive estimates of the autocorrelation coefficient do not differ by more than 0.001 or after 20 iterations.

Menu path: /Model/Time series/Cochrane-Orcutt

corr

Argument:	`[` *varlist* `]`
Example:	`corr y x1 x2 x3`

Prints the pairwise correlation coefficients for the variables in *varlist*, or for all variables in the data set if *varlist* is not given.

Menu path: /Data/Correlation matrix

Other access: Main window pop-up menu (multiple selection)

corrgm

Arguments:	*variable* `[` *maxlag* `]`
Example:	`corrgm x 12`

Prints the values of the autocorrelation function for the *variable* specified (either by name or number). See Ramanathan, Section 11.7. It is thus $\rho(u_t, u_{t-s})$ where u_t is the tth observation of the variable u and s is the number of lags.

The partial autocorrelations are also shown: these are net of the effects of intervening lags. The command also graphs the correlogram and prints the Box–Pierce Q statistic for testing the null hypothesis that the series is "white noise". This is asymptotically distributed as chi-square with degrees of freedom equal to the number of lags used.

If a *maxlag* value is specified the length of the correlogram is limited to at most that number of lags, otherwise the length is determined automatically.

Menu path: /Variable/Correlogram

Other access: Main window pop-up menu (single selection)

criteria

Arguments: *ess T k*

Example: `criteria 23.45 45 8`

Computes the Akaike Information Criterion (AIC) and Schwarz's Bayesian Information Criterion (BIC), given *ess* (error sum of squares), the number of observations (*T*), and the number of coefficients (*k*). *T*, *k*, and *ess* may be numerical values or names of previously defined variables.

critical

Arguments: *dist param1* [*param2*]

Examples: `critical t 20`

`critical X 5`

`critical F 3 37`

If *dist* is `t`, `X` or `F`, prints out the critical values for the student's *t*, chi-square or *F* distribution respectively, for the common significance levels and using the specified degrees of freedom, given as *param1* for t and chi-square, or *param1* and *param2* for *F*. If *dist* is `d`, prints the upper and lower values of the Durbin-Watson statistic at 5 percent significance, for the given number of observations, *param1*, and for the range of 1 to 5 explanatory variables.

Menu path: /Utilities/Statistical tables

cusum

Must follow the estimation of a model via OLS. Performs the CUSUM test for parameter stability. A series of (scaled) one-step ahead forecast errors is obtained by running a series of regressions: the first regression uses the first *k* observations and is used to generate a prediction of the dependent variable at observation $k + 1$; the second uses the first $k + 1$ observations and generates a prediction for observation $k + 2$, and so on (where *k* is the number of parameters in the original model). The cumulated sum of the scaled forecast errors is printed and graphed. The null hypothesis of parameter stability is rejected at the 5 percent significance level if the cumulated sum strays outside of the 95 percent confidence band.

The Harvey–Collier *t*-statistic for testing the null hypothesis of parameter stability is also printed. See Chapter 7 of Greene's *Econometric Analysis* for details.

Menu path: Model window, /Tests/CUSUM

data

Argument: *varlist*

Reads the variables in *varlist* from a database (gretl or RATS 4.0), which must have been opened previously using the open command. In addition, a data frequency and sample range must be established using the setobs and smpl commands prior to using this command. Here is a full example:

```
open macrodat.rat
setobs 4 1959:1
smpl ; 1999:4
data GDP_JP GDP_UK
```

These commands open a database named `macrodat.rat`, establish a quarterly data set starting in the first quarter of 1959 and ending in the fourth quarter of 1999, and then import the

series named GDP_JP and GDP_UK.

If the series to be read are of higher frequency than the working data set, you must specify a compaction method as below:

```
data (compact=average) LHUR PUNEW
```

The four available compaction methods are "average" (takes the mean of the high frequency observations), "last" (uses the last observation), "first" and "sum".

Menu path: /File/Browse databases

delete

Argument: [*varlist*]

Removes the listed variables (given by name or number) from the dataset. *Use with caution*: no confirmation is asked, and any variables with higher ID numbers will be re-numbered.

If no *varlist* is given with this command, it deletes the last (highest numbered) variable from the dataset.

Menu path: Main window pop-up (single selection)

diff

Argument: *varlist*

The first difference of each variable in *varlist* is obtained and the result stored in a new variable with the prefix d_. Thus `diff x y` creates the new variables `d_x = x(t) - x(t-1)` and `d_y = y(t) - y(t-1)`.

Menu path: /Data/Add variables/first differences

else

See if.

end

Ends a block of commands of some sort. For example, `end system` terminates an equation system.

endif

See if.

endloop

Marks the end of a command loop. See loop.

eqnprint

Argument: [*-f filename*]

Option: `--complete` (Create a complete document)

Must follow the estimation of a model. Prints the estimated model in the form of a LaTeX equation. If a filename is specified using the `-f` flag output goes to that file, otherwise it goes to a file with a name of the form `equation_N.tex`, where `N` is the number of models estimated to date in the current session. See also tabprint.

If the `--complete` flag is given, the LaTeX file is a complete document, ready for processing; otherwise it must be included in a document.

Menu path: Model window, /LaTeX

equation

Arguments:	*depvar indepvars*
Example:	`equation y x1 x2 x3 const`

Specifies an equation within a system of equations (see system). The syntax for specifying an equation within an SUR system is the same as that for, e.g., ols. For an equation within a Three-Stage Least Squares system you may either (a) give an OLS-type equation specification and provide a common list of instruments using the `instr` keyword (again, see system), or (b) use the same equation syntax as for tsls.

estimate

Arguments:	*systemname estimator*
Options:	`--iterate` (iterate to convergence)
	`--no-df-corr` (no degrees of freedom correction)
	`--geomean` (see below)
Examples:	`estimate "Klein Model 1" method=fiml`
	`estimate Sys1 method=sur`
	`estimate Sys1 method=sur --iterate`

Calls for estimation of a system of equations, which must have been previously defined using the system command. The name of the system should be given first, surrounded by double quotes if the name contains spaces. The estimator, which must be one of `ols`, `tsls`, `sur`, `3sls`, `fiml` or `liml`, is preceded by the string `method=`.

If the system in question has had a set of restrictions applied (see the restrict command), estimation will be subject to the specified restrictions.

If the estimation method is `sur` or `3sls` and the `--iterate` flag is given, the estimator will be iterated. In the case of SUR, if the procedure converges the results are maximum likelihood estimates. Iteration of three-stage least squares, however, does not in general converge on the full-information maximum likelihood results. The `--iterate` flag is ignored for other methods of estimation.

If the equation-by-equation estimators `ols` or `tsls` are chosen, the default is to apply a degrees of freedom correction when calculating standard errors. This can be suppressed using the `--no-df-corr` flag. This flag has no effect with the other estimators; no degrees of freedom correction is applied in any case.

By default, the formula used in calculating the elements of the cross-equation covariance matrix is

$$\hat{\sigma}_{i,j} = \frac{\hat{u}_i' \hat{u}_j}{T}$$

If the `--geomean` flag is given, a degrees of freedom correction is applied: the formula is

$$\hat{\sigma}_{i,j} = \frac{\hat{u}_i' \hat{u}_j}{\sqrt{(T - k_i)(T - k_j)}}$$

where the *k*s denote the number of independent parameters in each equation.

fcast

Arguments:	`[ startobs endobs ] fitvar`
Options:	`--dynamic` (create dynamic forecast)
	`--static` (create static forecast)
Examples:	`fcast 1997:1 2001:4 f1`
	`fcast fit2`

Must follow an estimation command. Forecasts are generated for the specified range (or the largest possible range if no *startobs* and *endobs* are given) and the values saved as *fitvar*, which can be printed, graphed, or plotted. The right-hand side variables are those in the original model. There is no provision to substitute other variables. If an autoregressive error process is specified the forecast incorporates the predictable fraction of the error process.

The choice between a static and a dynamic forecast applies only in the case of dynamic models, with an autoregressive error process or including one or more lagged values of the dependent variable as regressors. See fcasterr for more details.

Menu path: Model window, /Model data/Forecasts

fcasterr

Arguments:	*startobs endobs*
Options:	`--plot` (display graph)
	`--dynamic` (create dynamic forecast)
	`--static` (create static forecast)

After estimating a model you can use this command to print out fitted values over the specified observation range, along with (depending on the nature of the model and the available data) estimated standard errors of those predictions and 95 percent confidence intervals.

The choice between a static and a dynamic forecast applies only in the case of dynamic models, with an autoregressive error process or including one or more lagged values of the dependent variable as regressors. Static forecasts are one step ahead, based on realized values from the previous period, while dynamic forecasts employ the chain rule of forecasting. For example, if a forecast for y in 2008 requires as input a value of y for 2007, a static forecast is impossible without actual data for 2007. A dynamic forecast for 2008 is possible if a prior forecast can be substituted for y in 2007.

The default is to give a static forecast for any portion of the forecast range that lies with the sample range over which the model was estimated, and a dynamic forecast (if relevant) out of sample. The `dynamic` option requests a dynamic forecast from the earliest possible date, and the `static` option requests a static forecast even out of sample.

The nature of the forecast standard errors (if available) depends on the nature of the model and the forecast. For static linear models standard errors are computed using the method outlined by Davidson and MacKinnon (2004); they incorporate both uncertainty due to the error process and parameter uncertainty (summarized in the covariance matrix of the parameter estimates). For dynamic models, forecast standard errors are computed only in the case of a

dynamic forecast, and they do not incorporate parameter uncertainty. For nonlinear models, forecast standard errors are not presently available.

Menu path: Model window, /Model data/Forecasts

fit

A shortcut to `fcast`. Must follow an estimation command. Generates fitted values, in a series called `autofit`, for the current sample, based on the last regression. In the case of time-series models, also pops up a graph of fitted and actual values of the dependent variable against time.

freq

Argument:	*var*
Options:	`--quiet` (suppress printing of histogram)
	`--gamma` (test for gamma distribution)

With no options given, displays the frequency distribution for *var* (given by name or number) and shows the results of the Doornik–Hansen chi-square test for normality.

If the `--quiet` option is given, the histogram is not shown. If the `--gamma` option is given, the test for normality is replaced by Locke's nonparametric test for the null hypothesis that the variable follows the gamma distribution; see Locke (1976), Shapiro and Chen (2001).

In interactive mode a graph of the distribution is displayed.

Menu path: /Variable/Frequency distribution

function

Argument:	*fnname*

Opens a block of statements in which a function is defined. This block must be closed with `end function`. Please see Chapter 11 for details.

garch

Arguments:	*p q* ; *depvar* [*indepvars*]
Options:	`--robust` (robust standard errors)
	`--verbose` (print details of iterations)
	`--vcv` (print covariance matrix)
	`--arma-init` (initial variance parameters from ARMA)
Examples:	`garch 1 1 ; y`
	`garch 1 1 ; y 0 x1 x2 --robust`

Estimates a GARCH model (GARCH = Generalized Autoregressive Conditional Heteroskedasticity), either a univariate model or, if *indepvars* are specified, including the given exogenous variables. The integer values *p* and *q* represent the lag orders in the conditional variance equation:

$$h_t = \alpha_0 + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_i h_{t-j}$$

The gretl GARCH algorithm is basically that of Fiorentini, Calzolari and Panattoni (1996), used by kind permission of Professor Fiorentini.

Several variant estimates of the coefficient covariance matrix are available with this command. By default, the Hessian is used unless the `--robust` option is given, in which case the QML (White) covariance matrix is used. Other possibilities (e.g. the information matrix, or the Bollerslev–Wooldridge estimator) can be specified using the set command.

By default, the estimates of the variance parameters are initialized using the unconditional error variance from initial OLS estimation for the constant, and small positive values for the coefficients on the past values of the squared error and the error variance. The flag `--arma-init` calls for the starting values of these parameters to be set using an initial ARMA model, exploiting the relationship between GARCH and ARMA set out in Chapter 21 of Hamilton's Time Series Analysis. In some cases this may improve the chances of convergence.

Menu path: /Model/Time series/GARCH model

genr

Arguments: *newvar = formula*

Creates new variables, usually through transformations of existing variables. See also diff, logs, lags, ldiff, multiply and square for shortcuts. In the context of a `genr` formula, existing variables must be referenced by name, not ID number. The formula should be a well-formed combination of variable names, constants, operators and functions (described below). Note that further details on some aspects of this command can be found in Chapter 5.

This command may yield either a series or a scalar result. For example, the formula `x2 = x * 2` naturally yields a series if the variable `x` is a series and a scalar if `x` is a scalar. The formulae `x = 0` and `mx = mean(x)` naturally return scalars. Under some circumstances you may want to have a scalar result expanded into a series or vector. You can do this by using `series` as an "alias" for the `genr` command. For example, `series x = 0` produces a series all of whose values are set to 0. You can also use `scalar` as an alias for `genr`. It is not possible to coerce a vector result into a scalar, but use of this keyword indicates that the result *should be* a scalar: if it is not, an error occurs.

When a formula yields a series or vector result, the range over which the result is written to the target variable depends on the current sample setting. It is possible, therefore, to define a series piecewise using the `smpl` command in conjunction with `genr`.

Supported *arithmetical operators* are, in order of precedence: `^` (exponentiation); `*`, `/` and `%` (modulus or remainder); `+` and `-`.

The available *Boolean operators* are (again, in order of precedence): `!` (negation), `&` (logical AND), `|` (logical OR), `>`, `<`, `=`, `>=` (greater than or equal), `<=` (less than or equal) and `!=` (not equal). The Boolean operators can be used in constructing dummy variables: for instance `(x > 10)` returns 1 if `x` > 10, 0 otherwise.

Built-in constants are `pi` and `NA`. The latter is the missing value code: you can initialize a variable to the missing value with `scalar x = NA`.

Supported *functions* fall into these groups:

§ Standard mathematical functions: `abs`, `cos`, `exp`, `int` (integer part), `ln` (natural logarithm: `log` is a synonym), `sin`, `sqrt`. All of these take a single argument, which may be either a series or a scalar.

§ Standard statistical functions taking a single argument and yielding a scalar result: `max` (maximum value in a series), `min` (minimum value in series), `mean` (arithmetic mean), `median`,

`var` (variance), `sd` (standard deviation), `sst` (sum of squared deviations from the mean), `sum`.

§ Statistical functions taking one series as argument and yielding a series or vector result: `sort` (sort a series is ascending order of magnitude), `cum` (cumulate, or running sum).

§ Statistical functions taking two series as arguments and yielding a scalar result: `cov` (covariance), `corr` (correlation coefficient).

§ Special statistical functions: `pvalue` (see below), `cnorm` (standard normal CDF), `dnorm` (standard normal PDF), `resample` (resample a series with replacement, for bootstrap purposes), `hpfilt` (Hodrick–Prescott filter: this function returns the "cycle" component of the series), `bkfilt` (Baxter–King bandpass filter).

§ Time-series functions: `diff` (first difference), `ldiff` (log-difference, or first difference of natural logs), `fracdiff` (fractional difference). To generate lags of a variable `x`, use the syntax `x(-N)`, where `N` represents the desired lag length; to generate leads, use `x(+N)`.

§ Dataset functions yielding a series: `misszero` (replaces the missing observation code in a given series with zeros); `zeromiss` (the inverse operation to `misszero`); `missing` (at each observation, 1 if the argument has a missing value, 0 otherwise); `ok` (the opposite of `missing`).

§ Dataset functions yielding a scalar: `nobs` (gives the number of valid observations in a data series), `firstobs` (gives the 1-based observation number of the first non-missing value in a series), `lastobs` (observation number of the last non-missing observation in a series).

§ Pseudo-random numbers: `uniform` and `normal`. These functions do not take an argument and should be written with empty parentheses: `uniform()`, `normal()`. They create pseudo-random series drawn from the uniform (0–1) and standard normal distributions respectively. See also the set command, `seed` option. Uniform series are generated using the Mersenne Twister;[1] for normal series the method of Box and Muller (1958) is used, taking input from the Mersenne Twister.

All of the above functions with the exception of `cov`, `corr`, `pvalue`, `fracdiff`, `uniform` and `normal` take as their single argument either the name of a variable or an expression that evaluates to a variable (e.g. `ln((x1+x2)/2)`). The `pvalue` function takes the same arguments as the pvalue command, but in this context commas should be placed between the arguments. This function returns a one-tailed p-value, and in the case of the normal and *t* distributions, it is for the "short tail". With the normal, for example, both 1.96 and –1.96 will give a result of around 0.025. The `fracdiff` takes two arguments: the name of the series and a fraction, in the range –1 to 1.

Besides the operators and functions noted above there are some special uses of `genr`:

§ `genr time` creates a time trend variable (1,2,3,...) called `time`. `genr index` does the same thing except that the variable is called `index`.

§ `genr dummy` creates dummy variables up to the periodicity of the data. E.g. in the case of quarterly data (periodicity 4), the program creates `dummy_1` = 1 for first quarter and 0 in other quarters, `dummy_2` = 1 for the second quarter and 0 in other quarters, and so on.

§ `genr paneldum` creates a set of special dummy variables for use with a panel data set — see panel.

§ Various internal variables defined in the course of running a regression can be retrieved using `genr`, as follows:

`$ess`	error sum of squares
`$rsq`	unadjusted *R*-squared
`$T`	number of observations used

1. See Matsumoto and Nishimura (1998). The implementation is provided by `glib`, if available, or by the C code written by Nishimura and Matsumoto.

`$df`	degrees of freedom
`$trsq`	*TR*-squared (sample size times *R*-squared)
`$sigma`	standard error of residuals
`$aic`	Akaike Information Criterion
`$bic`	Schwarz's Bayesian Information Criterion
`$lnl`	log-likelihood (where applicable)
coeff(*var*)	estimated coefficient for variable *var*
stderr(*var*)	estimated standard error for variable *var*
rho(*i*)	*i*th order autoregressive coefficient for residuals
vcv(*x1*,*x2*)	estimated covariance between coefficients for the named variables *x1* and *x2*

Note: In the command-line program, `genr` commands that retrieve model-related data always reference the model that was estimated most recently. This is also true in the GUI program, if one uses `genr` in the "gretl console" or enters a formula using the "Define new variable" option under the Variable menu in the main window. With the GUI, however, you have the option of retrieving data from any model currently displayed in a window (whether or not it's the most recent model). You do this under the "Model data" menu in the model's window.

The internal series `$uhat` and `$yhat` hold, respectively, the residuals and fitted values from the last regression.

Three "internal" dataset variables are available: `$nobs` holds the number of observations in the current sample range (which may or may not equal the value of `$T`, the number of observations used in estimating the last model); `$nvars` holds the number of variables in the dataset (including the constant); and `$pd` holds the frequency or periodicity of the data (e.g. 4 for quarterly data).

Two special internal scalars, `$test` and `$pvalue`, hold respectively the value and the p-value of the test statistic that was generated by the last explicit hypothesis-testing command, if any (e.g. `chow`). Please see Chapter 5 for details on this.

The variable `t` serves as an index of the observations. For instance `genr dum = (t=15)` will generate a dummy variable that has value 1 for observation 15, 0 otherwise. The variable `obs` is similar but more flexible: you can use this to pick out particular observations by date or name. For example, `genr d = (obs>1986:4)` or `genr d = (obs="CA")`. The last form presumes that the observations are labeled; the label must be put in double quotes.

Scalar values can be pulled from a series in the context of a `genr` formula, using the syntax *varname*`[`*obs*`]`. The *obs* value can be given by number or date. Examples: `x[5]`, `CPI[1996:01]`. For daily data, the form *YYYY/MM/DD* should be used, e.g. `ibm[1970/01/23]`.

An individual observation in a series can be modified via `genr`. To do this, a valid observation number or date, in square brackets, must be appended to the name of the variable on the left-hand side of the formula. For example, `genr x[3] = 30` or `genr x[1950:04] = 303.7`.

Table 14-1 gives several examples of uses of `genr` with explanatory notes; here are a couple of tips on dummy variables:

§ Suppose `x` is coded with values 1, 2, or 3 and you want three dummy variables, `d1` = 1 if `x` = 1, 0 otherwise, `d2` = 1 if `x` = 2, and so on. To create these, use the commands:

```
genr d1 = (x=1)
genr d2 = (x=2)
```

```
genr d3 = (x=3)
```

§ To create `z = max(x,y)` do

```
genr d = x>y
genr z = (x*d)+(y*(1-d))
```

Table 14-1. Examples of use of genr command

Formula	Comment
`y = x1^3`	`x1` cubed
`y = ln((x1+x2)/x3)`	
`z = x>y`	`z(t)` = 1 if `x(t) > y(t)`, otherwise 0
`y = x(-2)`	`x` lagged 2 periods
`y = x(+2)`	`x` led 2 periods
`y = diff(x)`	`y(t) = x(t) - x(t-1)`
`y = ldiff(x)`	`y(t) = log x(t) - log x(t-1)`, the instantaneous rate of growth of `x`
`y = sort(x)`	sorts `x` in increasing order and stores in `y`
`y = -sort(-x)`	sort `x` in decreasing order
`y = int(x)`	truncate `x` and store its integer value as `y`
`y = abs(x)`	store the absolute values of `x`
`y = sum(x)`	sum `x` values excluding missing –999 entries
`y = cum(x)`	cumulation: $y_t = \sum_{\tau=1}^{t} x_\tau$
`aa = $ess`	set `aa` equal to the Error Sum of Squares from last regression
`x = coeff(sqft)`	grab the estimated coefficient on the variable `sqft` from the last regression
`rho4 = rho(4)`	grab the 4th-order autoregressive coefficient from the last model (presumes an `ar` model)
`cvx1x2 = vcv(x1, x2)`	grab the estimated coefficient covariance of vars `x1` and `x2` from the last model
`foo = uniform()`	uniform pseudo-random variable in range 0–1
`bar = 3 * normal()`	normal pseudo-random variable, $\mu = 0$, $\sigma = 3$
`samp = ok(x)`	= 1 for observations where `x` is not missing.

Menu path: /Variable/Define new variable

Other access: Main window pop-up menu

gnuplot

Arguments:	*yvars xvar* [*dumvar*]
Options:	`--with-lines` (use lines, not points)
	`--with-impulses` (use vertical lines)
	`--suppress-fitted` (don't show least squares fit)
	`--dummy` (see below)

Examples:
```
gnuplot y1 y2 x
gnuplot x time --with-lines
gnuplot wages educ gender --dummy
```

Without the `--dummy` option, the *yvars* are graphed against *xvar*. With `--dummy`, *yvar* is graphed against *xvar* with the points shown in different colors depending on whether the value of *dumvar* is 1 or 0.

The `time` variable behaves specially: if it does not already exist then it will be generated automatically.

In interactive mode the result is displayed immediately. In batch mode a gnuplot command file is written, with a name on the pattern `gpttmpN.plt`, starting with N = `01`. The actual plots may be generated later using `gnuplot` (under MS Windows, `wgnuplot`).

A further option to this command is available: following the specification of the variables to be plotted and the option flag (if any), you may add literal gnuplot commands to control the appearance of the plot (for example, setting the plot title and/or the axis ranges). These commands should be enclosed in braces, and each gnuplot command must be terminated with a semi-colon. A backslash may be used to continue a set of gnuplot commands over more than one line. Here is an example of the syntax:

```
{ set title 'My Title'; set yrange [0:1000]; }
```

Menu path: /Data/Graph specified vars

Other access: Main window pop-up menu, graph button on toolbar

graph

Arguments: *yvars xvar*

Option: `--tall` (use 40 rows)

ASCII graphics. The *yvars* (which may be given by name or number) are graphed against *xvar* using ASCII symbols. The `--tall` flag will produce a graph with 40 rows and 60 columns. Without it, the graph will be 20 by 60 (for screen output). See also gnuplot.

hausman

This test is available only after estimating a model using the pooled command (see also `panel` and `setobs`). It tests the simple pooled model 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. The random effects model decomposes the residual variance into two parts, one part specific to the cross-sectional unit and the other specific to the particular observation. (This estimator can be computed only 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 (that the pooled OLS estimator is adequate) against the random effects alternative.

The pooled OLS model may be rejected against both of the alternatives, fixed effects and random effects. Provided 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 and 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). A low p-value for this test counts against the random effects model and in favor of fixed effects.

Menu path: Model window, /Tests/panel diagnostics

hccm

Arguments:	*depvar indepvars*
Option:	`--vcv` (print covariance matrix)

Heteroskedasticity-Consistent Covariance Matrix: this command runs a regression where the coefficients are estimated via the standard OLS procedure, but the standard errors of the coefficient estimates are computed in a manner that is robust in the face of heteroskedasticity, namely using the MacKinnon–White "jackknife" procedure.

Menu path: /Model/HCCM

help

Gives a list of available commands. `help` *command* describes *command* (e.g. `help smpl`). You can type `man` instead of `help` if you like.

Menu path: /Help

hilu

Arguments:	*depvar indepvars*
Options:	`--vcv` (print covariance matrix)
	`--no-corc` (do not fine-tune results with Cochrane-Orcutt)

Computes parameter estimates for the specified model using the Hildreth–Lu search procedure (fine-tuned by the CORC procedure). This procedure is designed to correct for serial correlation of the error term. The error sum of squares of the transformed model is graphed against the value of rho from –0.99 to 0.99.

Menu path: /Model/Time Series/Hildreth-Lu

hsk

Arguments:	*depvar indepvars*
Option:	`--vcv` (print covariance matrix)

An OLS regression is run and the residuals are saved. The logs of the squares of these residuals then become the dependent variable in an auxiliary regression, on the right-hand side of which are the original independent variables plus their squares. The fitted values from the auxiliary regression are then used to construct a weight series, and the original model is re-estimated using weighted least squares. This final result is reported.

The weight series is formed as $1/\sqrt{e^{y^*}}$, where y^* denotes the fitted values from the auxiliary regression.

Menu path: /Model/Heteroskedasticity corrected

hurst

Argument: *varname*

Calculates the Hurst exponent (a measure of persistence or long memory) for a time-series variable having at least 128 observations.

The Hurst exponent is discussed by Mandelbrot. In theoretical terms it is the exponent, H, in the relationship

$$\mathrm{RS}(x) = an^H$$

where RS is the "rescaled range" of the variable x in samples of size n and a is a constant. The rescaled range is the range (maximum minus minimum) of the cumulated value or partial sum of x over the sample period (after subtraction of the sample mean), divided by the sample standard deviation.

As a reference point, if x is white noise (zero mean, zero persistence) then the range of its cumulated "wandering" (which forms a random walk), scaled by the standard deviation, grows as the square root of the sample size, giving an expected Hurst exponent of 0.5. Values of the exponent significantly in excess of 0.5 indicate persistence, and values less than 0.5 indicate anti-persistence (negative autocorrelation). In principle the exponent is bounded by 0 and 1, although in finite samples it is possible to get an estimated exponent greater than 1.

In gretl, the exponent is estimated using binary sub-sampling: we start with the entire data range, then the two halves of the range, then the four quarters, and so on. For sample sizes smaller than the data range, the RS value is the mean across the available samples. The exponent is then estimated as the slope coefficient in a regression of the log of RS on the log of sample size.

Menu path: /Variable/Hurst exponent

if

Flow control for command execution. The syntax is:

```
if   condition
     commands1
else

     commands2
endif
```

condition must be a Boolean expression, for the syntax of which see genr. The `else` block is optional; `if ... endif` blocks may be nested.

import

Argument: *filename*
Option: `--box1` (BOX1 data)

Brings in data from a comma-separated values (CSV) format file, such as can easily be written from a spreadsheet program. The file should have variable names on the first line and a rectangular data matrix on the remaining lines. Variables should be arranged "by observation" (one column per variable; each row represents an observation). See Chapter 4 for details.

With the `--box1` flag, reads a data file in BOX1 format, as can be obtained using the Data Extraction Service of the US Bureau of the Census.

Menu path: /File/Open data/import

include

Argument: *inputfile*

Intended for use in a command script, primarily for including definitions of functions. Executes the commands in *inputfile* then returns control to the main script.

info

Prints out any supplementary information stored with the current datafile.

Menu path: /Data/Read info

Other access: Data browser windows

label

Arguments: *varname* `-d` *description* `-n` *displayname*

Example: `label x1 -d "Description of x1" -n "Graph name"`

Sets the descriptive label for the given variable (if the `-d` flag is given, followed by a string in double quotes) and/or the "display name" for the variable (if the `-n` flag is given, followed by a quoted string). If a variable has a display name, this is used when generating graphs.

If neither flag is given, this command prints the current descriptive label for the specified variable, if any.

Menu path: /Variable/Edit attributes

Other access: Main window pop-up menu

kpss

Arguments: *order varname*

Options: `--trend` (include a trend)

`--verbose` (print regression results)

Examples: `kpss 8 y`

`kpss 4 x1 --trend`

Computes the KPSS test (Kwiatkowski, Phillips, Schmidt and Shin, 1992) for stationarity of a variable. The null hypothesis is that the variable in question is stationary, either around a level or, if the `--trend` option is given, around a deterministic linear trend.

The order argument determines the size of the window used for Bartlett smoothing. If the `--verbose` option is chosen the results of the auxiliary regression are printed, along with the estimated variance of the random walk component of the variable.

Menu path: /Variable/KPSS test

labels

Prints out the informative labels for any variables that have been generated using `genr`, and any labels added to the data set via the GUI.

lad

Arguments:	*depvar indepvars*
Option:	`--vcv` (print covariance matrix)

Calculates a regression that minimizes the sum of the absolute deviations of the observed from the fitted values of the dependent variable. Coefficient estimates are derived using the Barrodale–Roberts simplex algorithm; a warning is printed if the solution is not unique.

Standard errors are derived using the bootstrap procedure with 500 drawings. The covariance matrix for the parameter estimates, printed when the `--vcv` flag is given, is based on the same bootstrap.

Menu path: /Model/Least Absolute Deviation

lags

Alternate forms:	`lags` *varlist*
	`lags` *order* `;` *varlist*
Examples:	`lags x y`
	`lags 12 ; x y`

Creates new variables which are lagged values of each of the variables in *varlist*. By default the number of lagged variables equals the periodicity of the data. For example, if the periodicity is 4 (quarterly), the command `lags x` creates `x_1` = `x(t-1)`, `x_2` = `x(t-2)`, `x_3` = `x(t-3)` and `x_4` = `x(t-4)`. The number of lags created can be controlled by the optional first parameter.

Menu path: /Data/Add variables/lags of selected variables

ldiff

Argument:	*varlist*

The first difference of the natural log of each variable in *varlist* is obtained and the result stored in a new variable with the prefix `ld_`. Thus `ldiff x y` creates the new variables `ld_x` = $\ln(x_t) - \ln(x_{t-1})$ and `ld_y` = $\ln(y_t) - \ln(y_{t-1})$.

Menu path: /Data/Add variables/log differences

leverage

Option:	`--save` (save variables)

Must immediately follow an `ols` command. Calculates the leverage (h, which must lie in the range 0 to 1) for each data point in the sample on which the previous model was estimated. Displays the residual (u) for each observation along with its leverage and a measure of its influence on the estimates, $uh/(1-h)$. "Leverage points" for which the value of h exceeds $2k/n$ (where k is the number of parameters being estimated and n is the sample size) are flagged with an asterisk. For details on the concepts of leverage and influence see Davidson

and MacKinnon (1993, Chapter 2).

DFFITS values are also shown: these are "studentized residuals" (predicted residuals divided by their standard errors) multiplied by $\sqrt{h/(1-h)}$. For a discussion of studentized residuals and DFFITS see G. S. Maddala, *Introduction to Econometrics*, chapter 12; also Belsley, Kuh and Welsch (1980). Briefly, a "predicted residual" is the difference between the observed value of the dependent variable at observation *t*, and the fitted value for observation *t* obtained from a regression in which that observation is omitted (or a dummy variable with value 1 for observation *t* alone has been added); the studentized residual is obtained by dividing the predicted residual by its standard error.

If the `--save` flag is given with this command, then the leverage, influence and DFFITS values are added to the current data set.

Menu path: Model window, /Tests/influential observations

lmtest

Options:	`--logs` (non-linearity, logs)
	`--autocorr` (serial correlation)
	`--squares` (non-linearity, squares)
	`--white` (heteroskedasticity (White's test))

Must immediately follow an `ols` command. Carries out some combination of the following: Lagrange Multiplier tests for nonlinearity (logs and squares), White's test for heteroskedasticity, and the LMF test for serial correlation up to the periodicity (see Kiviet, 1986). The corresponding auxiliary regression coefficients are also printed out. See Ramanathan, Chapters 7, 8, and 9 for details. In the case of White's test, only the squared independent variables are used and not their cross products. In the case of the autocorrelation test, if the p-value of the LMF statistic is less than 0.05 (and the model was not originally estimated with robust standard errors) then serial correlation-robust standard errors are calculated and displayed. For details on the calculation of these standard errors see Wooldridge (2002, Chapter 12).

Menu path: Model window, /Tests

logistic

Arguments:	*depvar indepvars* `[ ymax=`*value*` ]`
Option:	`--vcv` (print covariance matrix)
Examples:	`logistic y const x`
	`logistic y const x ymax=50`

Logistic regression: carries out an OLS regression using the logistic transformation of the dependent variable,

$$\log\left(\frac{y}{y^* - y}\right)$$

The dependent variable must be strictly positive. If it is a decimal fraction, between 0 and 1, the default is to use a y^* value (the asymptotic maximum of the dependent variable) of 1. If the dependent variable is a percentage, between 0 and 100, the default y^* is 100. If you wish to set a different maximum, use the optional `ymax=`*value* syntax following the list of regressors. The supplied value must be greater than all of the observed values of the dependent variable.

The fitted values and residuals from the regression are automatically transformed using

$$y = \frac{y^*}{1 + e^{-x}}$$

where x represents either a fitted value or a residual from the OLS regression using the transformed dependent variable. The reported values are therefore comparable with the original dependent variable.

Note that if the dependent variable is binary, you should use the `logit` command instead.

Menu path: /Model/Logistic

logit

Arguments:	*depvar indepvars*
Options:	`--robust` (robust standard errors)
	`--vcv` (print covariance matrix)

Binomial logit regression. The dependent variable should be a binary variable. Maximum likelihood estimates of the coefficients on *indepvars* are obtained via the "binary response model regression" (BRMR) method outlined by Davidson and MacKinnon (2004). As the model is nonlinear the slopes depend on the values of the independent variables: the reported slopes are evaluated at the means of those variables. The chi-square statistic tests the null hypothesis that all coefficients are zero apart from the constant.

By default, standard errors are computed using the negative inverse of the Hessian. If the `--robust` flag is given, then QML or Huber–White standard errors are calculated instead. In this case the estimated covariance matrix is a "sandwich" of the inverse of the estimated Hessian and the outer product of the gradient. See Davidson and MacKinnon (2004, Chapter 10) for details.

If you want to use logit for analysis of proportions (where the dependent variable is the proportion of cases having a certain characteristic, at each observation, rather than a 1 or 0 variable indicating whether the characteristic is present or not) you should not use the `logit` command, but rather construct the logit variable (e.g. `genr lgt_p = log(p/(1 - p))`) and use this as the dependent variable in an OLS regression. See Ramanathan, Chapter 12.

Menu path: /Model/Logit

logs

Argument:	*varlist*

The natural log of each of the variables in *varlist* is obtained and the result stored in a new variable with the prefix `l_` which is "el" underscore. `logs x y` creates the new variables `l_x` = ln(x) and `l_y` = ln(y).

Menu path: /Data/Add variables/logs of selected variables

loop

Argument:	*control*
Options:	`--progressive` (enable special forms of certain commands)
	`--verbose` (report details of genr commands)
Examples:	`loop 1000`
	`loop 1000 --progressive`

```
loop while essdiff > .00001
loop for i=1991..2000
loop for (r=-.99; r<=.99; r+=.01)
```

The parameter *control* must take one of four forms, as shown in the examples: an integer number of times to repeat the commands within the loop; "`while`" plus a numerical condition; "`for`" plus a range of values for the internal index variable `i`; or "`for`" plus three expressions in parentheses, separated by semicolons. In the last form the left-hand expression initializes a variable, the middle expression sets a condition for iteration to continue, and the right-hand expression sets an increment or decrement to be applied at the start of the second and subsequent iterations. (This is a restricted form of the `for` statement in the C programming language.)

This command opens a special mode in which the program accepts commands to be executed repeatedly. Within a loop, only certain commands can be used: `genr`, `ols`, `print`, `printf`, `pvalue`, `sim`, `smpl`, `store`, `summary`, `if`, `else` and `endif`. You exit the mode of entering loop commands with `endloop`: at this point the stacked commands are executed.

See Chapter 10 for further details and examples. The effect of the `--progressive` option (which is designed for use in Monte Carlo simulations) is explained there.

mahal

Argument:	*varlist*
Options:	`--save` (add distances to the dataset)
	`--vcv` (print covariance matrix)

The Mahalanobis distance is the distance between two points in an k-dimensional space, scaled by the statistical variation in each dimension of the space. For example, if p and q are two observations on a set of k variables with covariance matrix C, then the Mahalanobis distance between the observations is given by

$$\sqrt{(p-q)'C^{-1}(p-q)}$$

where $(p-q)$ is a k-vector. This reduces to Euclidean distance if the covariance matrix is the identity matrix.

The space for which distances are computed is defined by the selected variables. For each observation in the current sample range, the distance is computed between the observation and the centroid of the selected variables. This distance is the multidimensional counterpart of a standard z-score, and can be used to judge whether a given observation "belongs" with a group of other observations.

If the `--vcv` option is given, the covariance matrix and its inverse are printed. If the `--save` option is given, the distances are saved to the dataset under the name `mdist` (or `mdist1`, `mdist2` and so on if there is already a variable of that name).

Menu path: /Data/Mahalanobis distances

meantest

Arguments:	*var1 var2*
Option:	`--unequal-vars` (assume variances are unequal)

Calculates the *t* statistic for the null hypothesis that the population means are equal for the variables *var1* and *var2*, and shows its p-value. By default the test statistic is calculated on the assumption that the variances are equal for the two variables; with the `--unequal-vars` option the variances are assumed to be different. This will make a difference to the test statistic only if there are different numbers of non-missing observations for the two variables.

Menu path: /Data/Difference of means

modeltab

Arguments: *add* or *show* or *free*

Manipulates the gretl "model table". See Chapter 3 for details. The sub-commands have the following effects: `add` adds the last model estimated to the model table, if possible; `show` displays the model table in a window; and `free` clears the table.

Menu path: Session window, Model table icon

mpols

Arguments: *depvar indepvars*

Computes OLS estimates for the specified model using multiple precision floating-point arithmetic. This command is available only if `gretl` is compiled with support for the Gnu Multiple Precision library (GMP).

To estimate a polynomial fit, using multiple precision arithmetic to generate the required powers of the independent variable, use the form, e.g. `mpols y 0 x ; 2 3 4` This does a regression of `y` on `x`, `x` squared, `x` cubed and `x` to the fourth power. That is, the numbers (which must be positive integers) to the right of the semicolon specify the powers of `x` to be used. If more than one independent variable is specified, the last variable before the semicolon is taken to be the one that should be raised to various powers.

Menu path: /Model/High precision OLS

multiply

Arguments: *x suffix varlist*

Examples:
```
multiply invpop pc 3 4 5 6
multiply 1000 big x1 x2 x3
```

The variables in *varlist* (referenced by name or number) are multiplied by *x*, which may be either a numerical value or the name of a variable already defined. The products are named with the specified *suffix* (maximum 3 characters). The original variable names are truncated first if need be. For instance, suppose you want to create per capita versions of certain variables, and you have the variable `pop` (population). A suitable set of commands is then:

```
genr invpop = 1/pop
multiply invpop pc income
```

which will create `incomepc` as the product of `income` and `invpop`, and `expendpc` as `expend` times `invpop`.

nls

Arguments: *function derivatives*

Option: `--vcv` (print covariance matrix)

Performs Nonlinear Least Squares (NLS) estimation using a modified version of the Levenberg–Marquandt algorithm. The user must supply a function specification. The parameters of this function must be declared and given starting values (using the `genr` command) prior to estimation. Optionally, the user may specify the derivatives of the regression function with respect to each of the parameters; if analytical derivatives are not supplied, a numerical approximation to the Jacobian is computed.

It is easiest to show what is required by example. The following is a complete script to estimate the nonlinear consumption function set out in William Greene's *Econometric Analysis* (Chapter 11 of the 4th edition, or Chapter 9 of the 5th). The numbers to the left of the lines are for reference and are not part of the commands. Note that the `--vcv` option, for printing the covariance matrix of the parameter estimates, attaches to the final command, `end nls`.

```
1   open greene11_3.gdt
2   ols C 0 Y
3   genr alpha = coeff(0)
4   genr beta = coeff(Y)
5   genr gamma = 1.0
6   nls C = alpha + beta * Y^gamma
7   deriv alpha = 1
8   deriv beta = Y^gamma
9   deriv gamma = beta * Y^gamma * log(Y)
10  end nls --vcv
```

It is often convenient to initialize the parameters by reference to a related linear model; that is accomplished here on lines 2 to 5. The parameters alpha, beta and gamma could be set to any initial values (not necessarily based on a model estimated with OLS), although convergence of the NLS procedure is not guaranteed for an arbitrary starting point.

The actual NLS commands occupy lines 6 to 10. On line 6 the `nls` command is given: a dependent variable is specified, followed by an equals sign, followed by a function specification. The syntax for the expression on the right is the same as that for the `genr` command. The next three lines specify the derivatives of the regression function with respect to each of the parameters in turn. Each line begins with the keyword `deriv`, gives the name of a parameter, an equals sign, and an expression whereby the derivative can be calculated (again, the syntax here is the same as for `genr`). These `deriv` lines are optional, but it is recommended that you supply them if possible. Line 10, `end nls`, completes the command and calls for estimation.

For further details on NLS estimation please see Chapter 9.

Menu path: /Model/Nonlinear Least Squares

noecho

Obsolete command. See set.

nulldata

Argument: *series_length*

Example: `nulldata 500`

Establishes a "blank" data set, containing only a constant and an index variable, with periodicity 1 and the specified number of observations. This may be used for simulation purposes: some of the `genr` commands (e.g. `genr uniform()`, `genr normal()`) will generate dummy data

from scratch to fill out the data set. This command may be useful in conjunction with `loop`. See also the "seed" option to the set command.

Menu path: /File/Create data set

ols

Arguments:	*depvar indepvars*
Options:	`--vcv` (print covariance matrix)
	`--robust` (robust standard errors)
	`--quiet` (suppress printing of results)
	`--no-df-corr` (suppress degrees of freedom correction)
	`--print-final` (see below)
Examples:	`ols 1 0 2 4 6 7`
	`ols y 0 x1 x2 x3 --vcv`
	`ols y 0 x1 x2 x3 --quiet`

Computes ordinary least squares (OLS) estimates with *depvar* as the dependent variable and *indepvars* as the list of independent variables. Variables may be specified by name or number; use the number zero for a constant term.

Besides coefficient estimates and standard errors, the program also prints p-values for *t* (two-tailed) and *F*-statistics. A p-value below 0.01 indicates statistical significance at the 1 percent level and is marked with ***. ** indicates significance between 1 and 5 percent and * indicates significance between the 5 and 10 percent levels. Model selection statistics (the Akaike Information Criterion or AIC and Schwarz's Bayesian Information Criterion) are also printed. The formula used for the AIC is that given by Akaike (1974), namely minus two times the maximized log-likelihood plus two times the number of parameters estimated.

If the option `--no-df-corr` is given, the usual degrees of freedom correction is not applied when calculating the estimated error variance (and hence also the standard errors of the parameter estimates).

The option `--print-final` is applicable only in the context of a loop. It arranges for the regression to be run silently on all but the final iteration of the loop. See the Section called *Loop examples* in Chapter 10 for details.

Various internal variables may be retrieved using the genr command, provided `genr` is invoked immediately after this command.

The specific formula used for generating robust standard errors (when the `--robust` option is given) can be adjusted via the set command.

Menu path: /Model/Ordinary Least Squares

Other access: Beta-hat button on toolbar

omit

Argument:	*varlist*
Options:	`--vcv` (print covariance matrix)
	`--quiet` (don't print estimates for reduced model)
	`--silent` (don't print anything)
Examples:	`omit 5 7 9`
	`omit seasonals --quiet`

This command must follow an estimation command. The selected variables are omitted from the previous model and the new model estimated. A test statistic for the joint significance of the omitted variables is printed, along with its p-value. The test statistic is F in the case of OLS estimation, an asymptotic Wald chi-square value otherwise. A p-value below 0.05 means that the coefficients are jointly significant at the 5 percent level.

If the `--quiet` option is given the printed results are confined to the test for the joint significance of the omitted variables, otherwise the estimates for the reduced model are also printed. In the latter case, the `--vcv` flag causes the covariance matrix for the coefficients to be printed also. If the `--silent` option is given, nothing is printed; nonetheless, the results of the test can be retrieved using the special variables `$test` and `$pvalue`.

Menu path: Model window, /Tests/omit variables

omitfrom

Arguments:	*modelID varlist*
Option:	`--quiet` (don't print estimates for reduced model)
Example:	`omitfrom 2 5 7 9`

Works like omit, except that you specify a previous model (using its ID number, which is printed at the start of the model output) to take as the base for omitting variables. The example above omits variables number 5, 7 and 9 from Model 2.

Menu path: Model window, /Tests/omit variables

open

Argument:	*datafile*

Opens a data file. If a data file is already open, it is replaced by the newly opened one. The program will try to detect the format of the data file (native, plain text, CSV or BOX1).

This command can also be used to open a database (gretl or RATS 4.0) for reading. In that case it should be followed by the data command to extract particular series from the database.

Menu path: /File/Open data

Other access: Drag a data file into gretl (MS Windows or Gnome)

outfile

Arguments:	*filename option*
Options:	`--append` (append to file)
	`--close` (close file)
	`--write` (overwrite file)
Examples:	`outfile --write regress.txt`
	`outfile --close`

Diverts output to *filename*, until further notice. Use the flag `--append` to append output to an existing file or `--write` to start a new file (or overwrite an existing one). Only one file can be opened in this way at any given time.

The `--close` flag is used to close an output file that was previously opened as above. Output will then revert to the default stream.

In the first example command above, the file `regress.txt` is opened for writing, and in the second it is closed. This would make sense as a sequence only if some commands were issued before the `--close`. For example if an `ols` command intervened, its output would go to `regress.txt` rather than the screen.

panel

Options:	`--cross-section` (stacked cross sections)
	`--time-series` (stacked time series)

Request that the current data set be interpreted as a panel (pooled cross section and time series). By default, or with the `--time-series` flag, the data set is taken to be in the form of stacked time series (successive blocks of data contain time series for each cross-sectional unit). With the `--cross-section` flag, the data set is read as stacked cross-sections (successive blocks contain cross sections for each time period). See also setobs.

pca

Argument:	*varlist*
Options:	`--save-all` (Save all components)
	`--save` (Save major components)

Principal Components Analysis. Prints the eigenvalues of the correlation matrix for the variables in *varlist* along with the proportion of the joint variance accounted for by each component. Also prints the corresponding eigenvectors (or "component loadings").

If the `--save` flag is given, components with eigenvalues greater than 1.0 are saved to the dataset as variables, with names `PC1`, `PC2` and so on. These artificial variables are formed as the sum of (component loading) times (standardized `Xi`), where `Xi` denotes the *i*th variable in *varlist*.

If the `--save-all` flag is given, all of the components are saved as described above.

Menu path: Main window pop-up (multiple selection)

pergm

Argument:	*varname*
Option:	`--bartlett` (use Bartlett lag window)

Computes and displays (and if not in batch mode, graphs) the spectrum of the specified variable. Without the `--bartlett` flag the sample periodogram is given; with the flag a Bartlett lag window of length $2\sqrt{T}$ (where T is the sample size) is used in estimating the spectrum (see Chapter 18 of Greene's *Econometric Analysis*). When the sample periodogram is printed, a t-test for fractional integration of the series ("long memory") is also given: the null hypothesis is that the integration order is zero.

Menu path: /Variable/spectrum

Other access: Main window pop-up menu (single selection)

poisson

Arguments:	*depvar indepvars* `[ ; ` *offset* `]`
Options:	`--vcv` (print covariance matrix)

	`--verbose` (print details of iterations)
Examples:	`poisson y 0 x1 x2`
	`poisson y 0 x1 x2 ; S`

Estimates a poisson regression. The dependent variable is taken to represent the occurrence of events of some sort, and must take on only non-negative integer values.

If a discrete random variable Y follows the Poisson distribution, then

$$\Pr(Y = y) = \frac{e^{-v} v^y}{y!}$$

for $y = 0, 1, 2, \ldots$. The mean and variance of the distribution are both equal to v. In the Poisson regression model, the parameter v is represented as a function of one or more independent variables. The most common version (and the only one supported by gretl) has

$$v = \exp(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots)$$

or in other words the log of v is a linear function of the independent variables.

Optionally, you may add an "offset" variable to the specification. This is a scale variable, the log of which is added to the linear regression function (implicitly, with a coefficient of 1.0). This makes sense if you expect the number of occurrences of the event in question to be proportional, other things equal, to some known factor. For example, the number of traffic accidents might be supposed to be proportional to traffic volume, other things equal, and in that case traffic volume could be specified as an "offset" in a Poisson model of the accident rate. The offset variable must be strictly positive.

Menu path: /Model/Poisson

plot

Argument:	*varlist*
Option:	`--one-scale` (force a single scale)

Plots the values for specified variables, for the range of observations currently in effect, using ASCII symbols. Each line stands for an observation and the values are plotted horizontally. By default the variables are scaled appropriately. See also gnuplot.

pooled

Arguments:	*depvar indepvars*
Options:	`--unit-weights` (feasible GLS)
	`--iterate` (iterate to Maximum Likelihood solution)
	`--vcv` (print covariance matrix)

By default, estimates a model via OLS (see ols for details on syntax), and flags it as a pooled or panel model, so that the hausman test item becomes available.

If the `--unit-weights` flag is given, estimation is by feasible GLS, using weights constructed from the specific error variances per cross-sectional unit. This offers a gain in efficiency over OLS if the error variance differs across units.

If, in addition, the `--iterate` flag is given, the GLS estimator is iterated: if this procedure converges it yields Maximum Likelihood estimates.

Menu path: /Model/Pooled OLS

print

Arguments:	*varlist* or *string_literal*
Options:	`--byobs` (by observations)
	`--ten` (use 10 significant digits)
	`--no-dates` (use simple observation numbers)
Examples:	`print x1 x2 --byobs`
	`print "This is a string"`

If *varlist* is given, prints the values of the specified variables; if no list is given, prints the values of all variables in the current data file. If the `--byobs` flag is given the data are printed by observation, otherwise they are printed by variable. If the `--ten` flag is given the data are printed by variable to 10 significant digits.

If the `--byobs` flag is given and the data are printed by observation, the default is to show the date (with time-series data) or the observation marker string (if any) at the start of each line. The `--no-dates` option suppresses the printing of dates or markers; a simple observation number is shown instead.

If the argument to `print` is a literal string (which must start with a double-quote, "), the string is printed as is. See also printf.

Menu path: /Data/Display values

printf

Arguments: *format args*

Prints scalar values under the control of a format string (providing a small subset of the `printf()` statement in the C programming language). Recognized formats are `%g` and `%f`, in each case with the various modifiers available in C. Examples: the format `%.10g` prints a value to 10 significant figures; `%12.6f` prints a value to 6 decimal places, with a width of 12 characters.

The format string itself must be enclosed in double quotes. The values to be printed must follow the format string, separated by commas. These values should take the form of either (a) the names of variables in the dataset, (b) expressions that are valid for the `genr` command, or (c) the special functions `varname()` or `date()`. The following example prints the values of two variables plus that of a calculated expression:

```
ols 1 0 2 3
genr b = coeff(2)
genr se_b = stderr(2)
printf "b = %.8g, standard error %.8g, t = %.4f\n", b, se_b, b/se_b
```

The next lines illustrate the use of the varname and date functions, which respectively print the name of a variable, given its ID number, and a date string, given a 1-based observation number.

```
printf "The name of variable %d is %s\n", i, varname(i)
printf "The date of observation %d is %s\n", j, date(j)
```

The maximum length of a format string is 127 characters. The escape sequences `\n` (newline), `\t` (tab), `\v` (vertical tab) and `\\` (literal backslash) are recognized. To print a literal percent

sign, use %%.

probit

Arguments:	*depvar indepvars*
Options:	`--robust` (robust standard errors)
	`--vcv` (print covariance matrix)

The dependent variable should be a binary variable. Maximum likelihood estimates of the coefficients on *indepvars* are obtained via the "binary response model regression" (BRMR) method outlined by Davidson and MacKinnon (2004). As the model is nonlinear the slopes depend on the values of the independent variables: the reported slopes are evaluated at the means of those variables. The chi-square statistic tests the null hypothesis that all coefficients are zero apart from the constant.

By default, standard errors are computed using the negative inverse of the Hessian. If the `--robust` flag is given, then QML or Huber–White standard errors are calculated instead. In this case the estimated covariance matrix is a "sandwich" of the inverse of the estimated Hessian and the outer product of the gradient. See Davidson and MacKinnon (2004, Chapter 10) for details.

Probit for analysis of proportions is not implemented in `gretl` at this point.

Menu path: /Model/Probit

pvalue

Arguments:	*dist* `[` *params* `]` *xval*
Examples:	`pvalue z zscore`
	`pvalue t 25 3.0`
	`pvalue X 3 5.6`
	`pvalue F 4 58 fval`
	`pvalue G xbar varx x`

Computes the area to the right of *xval* in the specified distribution (`z` for Gaussian, `t` for Student's *t*, `X` for chi-square, `F` for *F* and `G` for gamma). For the *t* and chi-square distributions the degrees of freedom must be given; for *F* numerator and denominator degrees of freedom are required; and for gamma the mean and variance are needed.

Menu path: /Utilities/p-value finder

pwe

Arguments:	*depvar indepvars*
Option:	`--vcv` (print covariance matrix)
Example:	`pwe 1 0 2 4 6 7`

Computes parameter estimates using the Prais–Winsten procedure, an implementation of feasible GLS which is designed to handle first-order autocorrelation of the error term. The procedure is iterated, as with corc; the difference is that while Cochrane–Orcutt discards the first observation, Prais–Winsten makes use of it. See, for example, Chapter 13 of Greene's *Econometric Analysis* (2000) for details.

Menu path: /Model/Time series/Prais-Winsten

quit

Exits from the program, giving you the option of saving the output from the session on the way out.

Menu path: /File/Exit

rename

Arguments: *varnumber newname*

Changes the name of the variable with identification number *varnumber* to *newname*. The *varnumber* must be between 1 and the number of variables in the dataset. The new name must be of 8 characters maximum, must start with a letter, and must be composed of only letters, digits, and the underscore character.

Menu path: /Variable/Edit attributes

Other access: Main window pop-up menu (single selection)

reset

Must follow the estimation of a model via OLS. Carries out Ramsey's RESET test for model specification (non-linearity) by adding the square and the cube of the fitted values to the regression and calculating the *F* statistic for the null hypothesis that the parameters on the two added terms are zero.

Menu path: Model window, /Tests/Ramsey's RESET

restrict

Imposes a set of linear restrictions on either (a) the model last estimated or (b) a system of equations previously defined and named. The syntax and effects of the command differ slightly in the two cases.

In both cases the set of restrictions should be started with the keyword "restrict" and terminated with "end restrict". In the single equation case the restrictions are implicitly to be applied to the last model, and they are evaluated as soon as the `restrict` command is terminated. In the system case the initial "restrict" must be followed by the name of a previously defined system of equations (see system). The restrictions are evaluated when the system is next estimated, using the estimate command.

Each restriction in the set should be expressed as an equation, with a linear combination of parameters on the left and a numeric value to the right of the equals sign. In the single-equation case, parameters are referenced in the form b*N*, where *N* represents the position in the list of regressors, starting at zero. For example, `b1` denotes the second regression parameter. In the system case, parameters are referenced using `b` plus two numbers in square brackets. The leading number represents the position of the equation within the system, starting from 1, and the second number indicates position in the list of regressors, starting at zero. For example `b[2,0]` denotes the first parameter in the second equation, and `b[3,1]` the second parameter in the third equation.

The `b` terms in the equation representing a restriction equation may be prefixed with a numeric multiplier, using `*` to represent multiplication, for example `3.5*b4`.

Here is an example of a set of restrictions for a previously estimated model:

```
restrict
```

```
 b1 = 0
 b2 - b3 = 0
 b4 + 2*b5 = 1
end restrict
```

And here is an example of a set of restrictions to be applied to a named system. (If the name of the system does not contain spaces, the surrounding quotes are not required.)

```
restrict "System 1"
 b[1,1] = 0
 b[1,2] - b[2,2] = 0
 b[3,4] + 2*b[3,5] = 1
end restrict
```

In the single-equation case the restrictions are evaluated via a Wald *F*-test, using the coefficient covariance matrix of the model in question. In the system case, the full results of estimating the system subject to the restrictions are shown; the test statistic depends on the estimator chosen (a Likelihood Ratio test if the system is estimated using a Maximum Likelihood method, or an asymptotic *F*-test otherwise.)

Menu path: Model window, /Tests/linear restrictions

rhodiff

Arguments:	*rholist* ; *varlist*
Examples:	`rhodiff .65 ; 2 3 4`
	`rhodiff r1 r2 ; x1 x2 x3`

Creates rho-differenced counterparts of the variables (given by number or by name) in *varlist* and adds them to the data set, using the suffix # for the new variables. Given variable `v1` in *varlist*, and entries `r1` and `r2` in *rholist*, `v1# = v1(t) - r1*v1(t-1) - r2*v1(t-2)` is created. The *rholist* entries can be given as numerical values or as the names of variables previously defined.

rmplot

Argument:	*varname*

Range–mean plot: this command creates a simple graph to help in deciding whether a time series, y(t), has constant variance or not. We take the full sample t=1,...,T and divide it into small subsamples of arbitrary size k. The first subsample is formed by y(1),...,y(k), the second is y(k+1), ..., y(2k), and so on. For each subsample we calculate the sample mean and range (= maximum minus minimum), and we construct a graph with the means on the horizontal axis and the ranges on the vertical. So each subsample is represented by a point in this plane. If the variance of the series is constant we would expect the subsample range to be independent of the subsample mean; if we see the points approximate an upward-sloping line this suggests the variance of the series is increasing in its mean; and if the points approximate a downward sloping line this suggests the variance is decreasing in the mean.

Besides the graph, gretl displays the means and ranges for each subsample, along with the slope coefficient for an OLS regression of the range on the mean and the p-value for the null hypothesis that this slope is zero. If the slope coefficient is significant at the 10 percent significance level then the fitted line from the regression of range on mean is shown on the graph.

Menu path: /Variable/Range-mean graph

run

Argument: *inputfile*

Execute the commands in *inputfile* then return control to the interactive prompt.

Menu path: Run icon in script window

runs

Argument: *varname*

Carries out the nonparametric "runs" test for randomness of the specified variable. If you want to test for randomness of deviations from the median, for a variable named x1 with a non-zero median, you can do the following:

```
genr signx1 = x1 - median(x1)
runs signx1
```

Menu path: /Variable/Runs test

scatters

Arguments: *yvar* ; *xvarlist* or *yvarlist ; xvar*

Examples:
```
scatters 1 ; 2 3 4 5
scatters 1 2 3 4 5 6 ; 7
```

Plots pairwise scatters of *yvar* against all the variables in *xvarlist*, or of all the variables in *yvarlist* against *xvar*. The first example above puts variable 1 on the *y*-axis and draws four graphs, the first having variable 2 on the *x*-axis, the second variable 3 on the *x*-axis, and so on. The second example plots each of variables 1 through 6 against variable 7 on the *x*-axis. Scanning a set of such plots can be a useful step in exploratory data analysis. The maximum number of plots is six; any extra variable in the list will be ignored.

Menu path: /Data/Multiple scatterplots

seed

Obsolete command. See set.

set

Arguments: *variable value*

Examples:
```
set qr on
set csv_delim tab
set horizon 10
```

Set the values of various program parameters. The given value remains in force for the duration of the gretl session unless it is changed by a further call to set. The parameters that can be set in this way are enumerated below. Note that the settings of hac_lag and hc_version are used when the --robust option is given to the ols command.

`echo` values: `off` or `on` (the default). Suppress or resume the echoing of commands in gretl's output.

`qr` values: `on` or `off` (the default). Use QR rather than Cholesky decomposition in calculating OLS estimates.

`seed` value: an unsigned integer. Sets the seed for the pseudo-random number generator. By default this is set from the system time; if you want to generate repeatable sequences of random numbers you must set the seed manually.

`hac_lag` values: `nw1` (the default) or `nw2`, or an integer. Sets the maximum lag value, p, used when calculating HAC (Heteroskedasticity and Autocorrelation Consistent) standard errors using the Newey-West approach, for time series data. `nw1` and `nw2` represent two variant automatic calculations based on the sample size, T: for nw1, $p = 0.75 \times T^{1/3}$, and for nw2, $p = 4 \times (T/100)^{2/9}$.

`hc_version` values: 0 (the default), 1, 2 or 3. Sets the variant used when calculating Heteroskedasticity Consistent standard errors with cross-sectional data. The options correspond to the HC0, HC1, HC2 and HC3 discussed by Davidson and MacKinnon in *Econometric Theory and Methods*, chapter 5. HC0 produces what are usually called "White's standard errors".

`force_hc` values: off (the default) or on. By default, with time-series data and when the `--robust` option is given with `ols`, the HAC estimator is used. If you set `force_hc` to "on", this forces calculation of the regular Heteroskedasticity Consistent Covariance Matrix (which does not take autocorrelation into account).

`garch_vcv` values: `unset`, `hessian`, `im` (information matrix) , `op` (outer product matrix), `qml` (QML estimator), `bw` (Bollerslev–Wooldridge). Specifies the variant that will be used for estimating the coefficient covariance matrix, for GARCH models. If `unset` is given (the default) then the Hessian is used unless the "robust" option is given for the garch command, in which case QML is used.

`hp_lambda` values: `auto` (the default), or a numerical value. Sets the smoothing parameter for the Hodrick–Prescott filter (see the `hpfilt` function under the `genr` command). The default is to use 100 times the square of the periodicity, which gives 100 for annual data, 1600 for quarterly data, and so on.

`bkbp_limits` values: two integers, the second greater than the first (the defaults are 8 and 32). Sets the frequency bounds for the Baxter–King bandpass filter (see the `bkfilt` function under the `genr` command).

`bkbp_k` value: one integer (the default is 8). Sets the approximation order for the Baxter–King bandpass filter.

`horizon` value: one integer (the default is based on the frequency of the data). Sets the horizon for impulse responses and forecast variance decompositions in the context of vector autoregressions.

`csv_delim` value: either `comma` (the default), `space` or `tab`. Sets the column delimiter used when saving data to file in CSV format.

setobs

Arguments:	*periodicity startobs*
Options:	`--cross-section` (interpret as cross section)
	`--time-series` (interpret as time series)
	`--stacked-cross-section` (interpret as panel data)
	`--stacked-time-series` (interpret as panel data)
Examples:	`setobs 4 1990:1 --time-series`
	`setobs 12 1978:03`
	`setobs 1 1 --cross-section`
	`setobs 20 1:1 --stacked-time-series`

Force the program to interpret the current data set as having a specified structure.

The *periodicity*, which must be an integer, represents frequency in the case of time-series data (1 = annual; 4 = quarterly; 12 = monthly; 52 = weekly; 5, 6, or 7 = daily; 24 = hourly). In the case of panel data the periodicity means the number of lines per data block: this corresponds to the number of cross-sectional units in the case of stacked cross-sections, or the number of time periods in the case of stacked time series. In the case of simple cross-sectional data the periodicity should be set to 1.

The starting observation represents the starting date in the case of time series data. Years may be given with two or four digits; subperiods (for example, quarters or months) should be separated from the year with a colon. In the case of panel data the starting observation should be given as 1:1; and in the case of cross-sectional data, as 1. Starting observations for daily or weekly data should be given in the form YY/MM/DD or YYYY/MM/DD (or simply as 1 for undated data).

If no explicit option flag is given to indicate the structure of the data the program will attempt to guess the structure from the information given.

Menu path: /Sample/Dataset structure

setmiss

Arguments:	*value* [*varlist*]
Examples:	`setmiss -1`
	`setmiss 100 x2`

Get the program to interpret some specific numerical data value (the first parameter to the command) as a code for "missing", in the case of imported data. If this value is the only parameter, as in the first example above, the interpretation will be applied to all series in the data set. If *value* is followed by a list of variables, by name or number, the interpretation is confined to the specified variable(s). Thus in the second example the data value 100 is interpreted as a code for "missing", but only for the variable `x2`.

Menu path: /Sample/Set missing value code

shell

Argument:	*shellcommand*
Examples:	`! ls -al`
	`! notepad`

A ! at the beginning of a command line is interpreted as an escape to the user's shell. Thus arbitrary shell commands can be executed from within gretl.

sim

Arguments: `[ startobs endobs ] varname a0 a1 a2 ...`

Examples:
```
sim 1979.2 1983.1 y 0 0.9
sim 15 25 y 10 0.8 x
```

Simulates values for *varname* for the current sample range, or for the range *startobs* through *endobs* if these optional arguments are given. The variable *y* must have been defined earlier with appropriate initial values. The formula used is

$$y_t = a_{0t} + a_{1t}y_{t-1} + a_{2t}y_{t-2} + \ldots$$

The ai(t) terms may be either numerical constants or variable names previously defined; these terms may be prefixed with a minus sign.

This command is deprecated. You should use genr instead.

smpl

Alternate forms:
```
smpl startobs endobs

smpl +i -j
smpl dumvar --dummy
smpl condition --restrict
smpl --no-missing [ varlist ]
smpl n --random
smpl full
```

Examples:
```
smpl 3 10
smpl 1960:2 1982:4
smpl +1 -1
smpl x > 3000 --restrict
smpl y > 3000 --restrict --replace
smpl 100 --random
```

Resets the sample range. The new range can be defined in several ways. In the first alternate form (and the first two examples) above, *startobs* and *endobs* must be consistent with the periodicity of the data. Either one may be replaced by a semicolon to leave the value unchanged. In the second form, the integers *i* and *j* (which may be positive or negative, and should be signed) are taken as offsets relative to the existing sample range. In the third form *dummyvar* must be an indicator variable with values 0 or 1 at each observation; the sample will be restricted to observations where the value is 1. The fourth form, using --restrict, restricts the sample to observations that satisfy the given Boolean condition (which is specified according to the syntax of the genr command).

With the --no-missing form, if *varlist* is specified observations are selected on condition that all variables in *varlist* have valid values at that observation; otherwise, if no *varlist* is given, observations are selected on condition that *all* variables have valid (non-missing) values.

With the `--random` flag, the specified number of cases are selected from the full dataset at random. If you wish to be able to replicate this selection you should set the seed for the random number generator first (see the set command).

The final form, `smpl full`, restores the full data range.

Note that sample restrictions are, by default, cumulative: the baseline for any `smpl` command is the current sample. If you wish the command to act so as to replace any existing restriction you can add the option flag `--replace` to the end of the command.

The internal variable `obs` may be used with the `--restrict` form of `smpl` to exclude particular observations from the sample. For example

```
smpl obs!=4 --restrict
```

will drop just the fourth observation. If the data points are identified by labels,

```
smpl obs!="USA" --restrict
```

will drop the observation with label "USA".

One point should be noted about the `--dummy`, `--restrict` and `--no-missing` forms of `smpl`: Any "structural" information in the data file (regarding the time series or panel nature of the data) is lost when this command is issued. You may reimpose structure with the setobs command.

Please see Chapter 6 for further details.

Menu path: /Sample

spearman

Arguments:	*x y*
Option:	`--verbose` (print ranked data)

Prints Spearman's rank correlation coefficient for the two variables *x* and *y*. The variables do not have to be ranked manually in advance; the function takes care of this.

The automatic ranking is from largest to smallest (i.e. the largest data value gets rank 1). If you need to invert this ranking, create a new variable which is the negative of the original first. For example:

```
genr altx = -x
spearman altx y
```

Menu path: /Model/Rank correlation

square

Argument:	*varlist*
Option:	`--cross` (generate cross-products as well as squares)

Generates new variables which are squares of the variables in *varlist* (plus cross-products if the `--cross` option is given). For example, `square x y` will generate `sq_x` = `x` squared, `sq_y` = `y` squared and (optionally) `x_y` = `x` times `y`. If a particular variable is a dummy variable it is not squared because we will get the same variable.

Menu path: /Data/Add variables/squares of variables

store

Arguments: *datafile* [*varlist*]
Options: `--csv` (use CSV format)
`--omit-obs` (see below, on CSV format)
`--gnu-octave` (use GNU Octave format)
`--gnu-R` (use GNU R format)
`--traditional` (use traditional ESL format)
`--gzipped` (apply gzip compression)
`--dat` (use PcGive ASCII format)
`--database` (use gretl database format)
`--overwrite` (see below, on database format)

Saves either the entire dataset or, if a *varlist* is supplied, a specified subset of the variables in the current dataset, to the file given by *datafile*.

By default the data are saved in "native" gretl format, but the option flags permit saving in several alternative formats. CSV (Comma-Separated Values) data may be read into spreadsheet programs, and can also be manipulated using a text editor. The formats of `Octave`, `R` and `PcGive` are designed for use with the respective programs. Gzip compression may be useful for large datasets. See Chapter 4 for details on the various formats.

The option flag `--omit-obs` is applicable only when saving data in CSV format. By default, if the data are time series or panel or if the dataset includes specific observation markers, the CSV file includes a first column identifying the observations (e.g. by date). If the `--omit-obs` flag is given this column is omitted; only the actual data are printed.

Note that any scalar variables will not be saved automatically: if you wish to save scalars you must explicitly list them in *varlist*.

The option of saving in gretl database format is intended to help with the construction of large sets of series, possibly having mixed frequencies and ranges of observations. At present this option is available only for annual, quarterly or monthly time-series data. If you save to a file that already exists, the default action is to append the newly saved series to the existing content of the database. In this context it is an error if one or more of the variables to be saved has the same name as a variable that is already present in the database. The `--overwrite` flag has the effect that, if there are variable names in common, the newly saved variable replaces the variable of the same name in the original dataset.

Menu path: /File/Save data; /File/Export data

summary

Argument: [*varlist*]

Print summary statistics for the variables in *varlist*, or for all the variables in the data set if *varlist* is omitted. Output consists of the mean, standard deviation (sd), coefficient of variation (= sd/mean), median, minimum, maximum, skewness coefficient, and excess kurtosis.

Menu path: /Data/Summary statistics

Other access: Main window pop-up menu

system

Alternate forms: `system method=`*estimator*

`system name=`*sysname*

Argument: *savevars*

Examples:
```
system name="Klein Model 1"
system method=sur
system method=sur save=resids
system method=3sls save=resids,fitted
```

Starts a system of equations. Either of two forms of the command may be given, depending on whether you wish to save the system for estimation in more than one way or just estimate the system once.

To save the system you should give it a name, as in the first example (if the name contains spaces it must be surrounded by double quotes). In this case you estimate the system using the estimate command. With a saved system of equations, you are able to impose restrictions (including cross-equation restrictions) using the restrict command.

Alternatively you can specify an estimator for the system using `method=` followed by a string identifying one of the supported estimators: `ols` (Ordinary Least Squares), `tsls` (Two-Stage Least Squares) `sur` (Seemingly Unrelated Regressions), `3sls` (Three-Stage Least Squares), `fiml` (Full Information Maximum Likelihood) or `liml` (Limited Information Maximum Likelihood). In this case the system is estimated once its definition is complete.

An equation system is terminated by the line `end system`. Within the system four sorts of statement may be given, as follows.

§ equation: specify an equation within the system. At least two such statements must be provided.

§ `instr`: for a system to be estimated via Three-Stage Least Squares, a list of instruments (by variable name or number). Alternatively, you can put this information into the `equation` line using the same syntax as in the tsls command.

§ `endog`: for a system of simultaneous equations, a list of endogenous variables. This is primarily intended for use with FIML estimation, but with Three-Stage Least Squares this approach may be used instead of giving an `instr` list; then all the variables not identified as endogenous will be used as instruments.

§ `identity`: for use with FIML, an identity linking two or more of the variables in the system. This sort of statement is ignored when an estimator other than FIML is used.

In the optional `save=` field of the command you can specify whether to save the residuals (`resids`) and/or the fitted values (`fitted`).

For full examples of the specification and estimation of systems of equations, please see the scripts `klein.inp` and `greene14_2.inp` (supplied with the gretl distribution).

tabprint

Argument: `[ -f` *filename* `]`

Option: `--complete` (Create a complete document)

Must follow the estimation of a model. Prints the estimated model in the form of a LaTeX table. If a filename is specified using the `-f` flag output goes to that file, otherwise it goes to a file with a name of the form `model_N.tex`, where `N` is the number of models estimated to date

in the current session. See also eqnprint.

If the `--complete` flag is given the LaTeX file is a complete document, ready for processing; otherwise it must be included in a document.

Menu path: Model window, /LaTeX

testuhat

Must follow a model estimation command. Gives the frequency distribution for the residual from the model along with a chi-square test for normality, based on the procedure suggested by Doornik and Hansen (1984).

Menu path: Model window, /Tests/normality of residual

tobit

Arguments:	*depvar indepvars*
Options:	`--vcv` (print covariance matrix)
	`--verbose` (print details of iterations)

Estimates a Tobit model. This model may be appropriate when the dependent variable is "truncated". For example, positive and zero values of purchases of durable goods on the part of individual households are observed, and no negative values, yet decisions on such purchases may be thought of as outcomes of an underlying, unobserved disposition to purchase that may be negative in some cases. For details see Greene's *Econometric Analysis*, Chapter 20.

Menu path: /Model/Tobit

tsls

Arguments:	*depvar indepvars* ; *instruments*
Options:	`--vcv` (print covariance matrix)
	`--robust` (robust standard errors)
Example:	`tsls y1 0 y2 y3 x1 x2 ; 0 x1 x2 x3 x4 x5 x6`

Computes two-stage least squares (TSLS or IV) estimates: *depvar* is the dependent variable, *indepvars* is the list of independent variables (including right-hand side endogenous variables) in the structural equation for which TSLS estimates are needed; and *instruments* is the combined list of exogenous and predetermined variables in all the equations. If the *instruments* list is not at least as long as *indepvars*, the model is not identified.

In the above example, the `y`s are the endogenous variables and the `x`s are the exogenous and predetermined variables.

Menu path: /Model/Two-Stage least Squares

var

Arguments:	*order varlist* ; *detlist*
Options:	`--nc` (do not include a constant)
	`--seasonals` (include seasonal dummy variables)
	`--robust` (robust standard errors)
	`--impulse-responses` (print impulse responses)

	`--variance-decomp` (print forecast variance decompositions)
Examples:	`var 4 x1 x2 x3 ; time mydum`
	`var 4 x1 x2 x3 --seasonals`

Sets up and estimates (using OLS) a vector autoregression (VAR). The first argument specifies the lag order, then follows the setup for the first equation. Don't include lags among the elements of *varlist* — they will be added automatically. The semi-colon separates the stochastic variables, for which *order* lags will be included, from deterministic or exogenous terms in *detlist*.

In fact, gretl is able to recognize the more common deterministic variables (time trend, dummy variables with no values other than 0 and 1) as such, so these do not have to placed after the semi-colon. More complex deterministic variables (e.g. a time trend interacted with a dummy variable) must be put after the semi-colon. Note that a constant is included automatically unless you give the `--nc` flag, and seaonal dummy variables may be added using the `--seasonals` flag.

A separate regression is run for each variable in *varlist*. Output for each equation includes *F*-tests for zero restrictions on all lags of each of the variables, an *F*-test for the significance of the maximum lag, and, if the `--impulse-responses` flag is given, forecast variance decompositions and impulse responses.

Forecast variance decompositions and impulse responses are based on the Cholesky decomposition of the contemporaneous covariance matrix, and in this context the order in which the (stochastic) variables are given matters. The first variable in the list is assumed to be "most exogenous" within-period. The horizon for variance decompositions and impulse responses can be set using the set command.

Menu path: /Model/Time series/Vector autoregression

varlist

Prints a listing of variables currently available. `list` and `ls` are synonyms.

vartest

Arguments:	*var1 var2*

Calculates the *F* statistic for the null hypothesis that the population variances for the variables *var1* and *var2* are equal, and shows its p-value.

Menu path: /Data/Difference of variances

vecm

Arguments:	*order rank varlist*
Options:	`--nc` (no constant)
	`--rc` (restricted constant)
	`--crt` (constant and restricted trend)
	`--ct` (constant and unrestricted trend)
	`--seasonals` (include centered seasonal dummies)
Examples:	`vecm 4 1 Y1 Y2 Y3`
	`vecm 3 2 Y1 Y2 Y3 --rc`

A VECM is a form of vector autoregression or VAR (see var), applicable where the variables in the model are individually integrated of order 1 (that is, are random walks, with or without drift), but exhibit cointegration. This command is closely related to the Johansen test for cointegration (see coint2).

The *order* parameter to this command represents the lag order of the VAR system. The number of lags in the VECM itself (where the dependent variable is given as a first difference) is one less than *order*.

The *rank* parameter represents the cointegration rank, or in other words the number of cointegrating vectors. This must be greater than zero and less than or equal to (generally, less than) the number of endogenous variables given in *varlist*.

varlist supplies the list of endogenous variables, in levels. The inclusion of deterministic terms in the model is controlled by the option flags. The default if no option is specified is to include an "unrestricted constant", which allows for the presence of a non-zero intercept in the cointegrating relations as well as a trend in the levels of the endogenous variables. In the literature stemming from the work of Johensen (see for example his 1995 book) this is often referred to as "case 3". The first four options given above, which are mutually exclusive, produce cases 1, 2, 4 and 5 respectively. The meaning of these cases and the criteria for selecting a case are explained in Chapter 12.

The `--seasonals` option, which may be combined with any of the other options, specifies the inclusion of a set of centered seasonal dummy variables. This option is available only for quarterly or monthly data.

The first example above specifies a VECM with lag order 4 and a single cointegrating vector. The endogenous variables are `Y1`, `Y2` and `Y3`. The second example uses the same variables but specifies a lag order of 3 and two cointegrating vectors; it also specifies a "restricted constant", which is appropriate if the cointegrating vectors may have a non-zero intercept but the `Y` variables have no trend.

Menu path: /Model/Time series/VECM

vif

Must follow the estimation of a model which includes at least two independent variables. Calculates and displays the Variance Inflation Factors (VIFs) for the regressors. The VIF for regressor j is defined as

$$\frac{1}{1 - R_j^2}$$

where R_j is the coefficient of multiple correlation between regressor j and the other regressors. The factor has a minimum value of 1.0 when the variable in question is orthogonal to the other independent variables. Neter, Wasserman, and Kutner (1990) suggest inspecting the largest VIF as a diagnostic for collinearity; a value greater than 10 is sometimes taken as indicating a problematic degree of collinearity.

Menu path: Model window, /Tests/collinearity

wls

Arguments:	*wtvar depvar indepvars*
Option:	`--vcv` (print covariance matrix)

Computes weighted least squares estimates using *wtvar* as the weight, *depvar* as the dependent variable, and *indepvars* as the list of independent variables. Specifically, an OLS regression is run on *wtvar* * *depvar* against *wtvar* * *indepvars*. If the *wtvar* is a dummy variable,

this is equivalent to eliminating all observations with value zero for *wtvar*.

Menu path: /Model/Weighted Least Squares

Estimators and tests: summary

Table 14-2 shows the estimators available under the Model menu in gretl's main window. The corresponding script command (if there is one available) is shown in parentheses. For details consult the command's entry in Chapter 14.

Table 14-2. Estimators

Estimator	Comment
Ordinary Least Squares (ols)	The workhorse estimator
Weighted Least Squares (wls)	Heteroskedasticity, exclusion of selected observations
HCCM (hccm)	Heteroskedasticity corrected covariance matrix
Heteroskedasticity corrected (hsk)	Weighted Least Squares based on predicted error variance
Cochrane–Orcutt (corc)	First-order autocorrelation
Hildreth–Lu (hilu)	First-order autocorrelation
Prais–Winsten (pwe)	First-order autocorrelation
Autoregressive Estimation (ar)	Higher-order autocorrelation (generalized Cochrane–Orcutt)
ARMAX (arma)	Time-series model with ARMA error
GARCH (garch)	Generalized Autoregressive Conditional Heteroskedasticity
Vector Autoregression (var)	Systems of time-series equations
Cointegration test (coint)	Long-run relationships between series
Two-Stage Least Squares (tsls)	Simultaneous equations
Nonlinear Least Squares (nls)	Nonlinear models
Logit (logit)	Binary dependent variable (logistic distribution)
Probit (probit)	Binary dependent variable (normal distribution)
Tobit (tobit)	Truncated dependent variable
Logistic (logistic)	OLS, with logistic transformation of dependent variable
Least Absolute Deviation (lad)	Alternative to Least Squares
Rank Correlation (spearman)	Correlation with ordinal data
Pooled OLS (pooled)	OLS estimation for pooled cross-section, time series data
Multiple precision OLS (mpols)	OLS estimation using multiple precision arithmetic

Table 14-3 shows the tests that are available under the Tests menu in a model window, after estimation.

Table 14-3. Tests for models

Test	Corresponding command
Omit variables (*F*-test if OLS)	`omit`
Add variables (*F*-test if OLS)	`add`
Sum of coefficients (*t*-test if OLS)	`coeffsum`
Nonlinearity (squares)	`lmtest --squares`
Nonlinearity (logs)	`lmtest --logs`
Nonlinearity (Ramsey's RESET)	`reset`
Heteroskedasticity (White's test)	`lmtest --white`
Influential observations	`leverage`
Autocorrelation up to the data frequency	`lmtest --autocorr`
Chow (structural break)	`chow`
CUSUM (parameter stability)	`cusum`
ARCH (conditional heteroskedasticity)	`arch`
Normality of residual	`testuhat`
Panel diagnostics	`hausman`

Table 14-4 shows the correspondence between the long-form option flags used in the command reference above and the short versions.

Table 14-4. Long- and short-form options

Command	Option	Effect	Short form
`arma`	`--x-12-arima`	estimate with X-12-ARIMA	`-x`
	`--verbose`	print iteration details	`-v`
`coint2`	`--verbose`	print details of VARs	`-v`
`eqnprint`, `tabprint`	`--complete`	complete TeX document	`-o`
`fcasterr`	`--plot`	show graph	`-o`
`gnuplot`	`--with-lines`	plot with lines	`-o`
	`--with-impulses`	plot with impulses	`-m`
	`--suppress-fitted`	don't show fitted line	`-s`
	`--dummy`	factor separation	`-z`
`import`	`--box1`	import BOX1 data	`-o`
`leverage`	`--save`	save values to dataset	`-s`
`lmtest`	`--logs`	non-linearity (logs)	`-l`
	`--squares`	non-linearity (squares)	`-s`
	`--white`	White's test (heteroskedasticity)	`-w`

Command	Option	Effect	Short form
	`--autocorr`	serial correlation	`-m`
`meantest`	`--unequal-vars`	assume unequal variances	`-o`
`leverage`	`--save`	save values to dataset	`-s`
`ols`	`--robust`	robust standard errors	`-r`
	`--vcv`	print covariance matrix	`-o`
	`--quiet`	don't print results	`-q`
`outfile`	`--write`	open file for writing	`-w`
	`--append`	append to existing file	`-a`
	`--close`	close file	`-c`
`panel`	`--time-series`	stacked time series	`-s`
	`--cross-section`	stacked cross-sections	`-c`
`pca`	`--save`	save useful eigenvectors	`-s`
	`--save-all`	save all eigenvectors	`-a`
`pergm`	`--bartlett`	use Bartlett lag window	`-o`
`plot`	`--one-scale`	don't scale variables	`-o`
`print`	`--byobs`	variables in columns	`-o`
	`--ten`	use 10 significant figures	`-t`
`smpl`	`--dummy`	restrict using dummy var	`-o`
	`--no-missing`	only complete observations	`-m`
	`--restrict`	use boolean criterion	`-r`
`spearman`	`--verbose`	print original, ranked data	`-o`
`square`	`--cross`	generate cross-products	`-o`
`store`	`--csv`	comma-separated values	`-c`
	`--traditional`	traditional ESL format	`-t`
	`--gnu-octave`	GNU Octave format	`-m`
	`--gnu-R`	GNU R format	`-r`
	`--gzipped`	gzip compression	`-z`
`var`	`--quiet`	don't print all results	`-q`
Model commands	`--vcv`	print covariance matrix	`-o`

Chapter 15. Troubleshooting gretl

Bug reports

Bug reports are welcome. I believe 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?

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 to produce an informative error message window. 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 (e.g. from an xterm under the X window system, or from a "DOS box" under MS Windows, in which case type `gretlw32.exe`), rather than via a desktop menu entry or icon. 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.

System	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 16. The command line interface

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 the Section called *gretlcli* in Chapter 13 above).

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.

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.

```
esl -b datafile < inputfile > outputfile
```

while for `gretlcli` you type:

```
gretlcli -b inputfile > outputfile
```

1. Actually, the key bindings shown below are only the defaults; they can be customized. See the readline manual.

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

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.

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 7 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.

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. I have abided as far as possible by the ISO/ANSI C Standard (C89), although the graphical user interface and some other components necessarily make use of platform-specific extensions.

gretl was developed under Linux. The shared library and command-line client should compile and run on any platform that (a) supports ISO/ANSI C, (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. I recommend use of GTK 2.0 if it is available, 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).

I'm 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").

In October 2002 I had a useful exchange with Giovanni Baiocchi and Walter Distaso, who were writing a review of gretl for the *Journal of Applied Econometrics*, and James MacKinnon, software review editor for the journal.[1] and I am 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, where I had done my testing. 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 (see genr) then print it using `print --ten` (see Chapter 14). This will show the value to 10 digits.

1. This review has since been published; see Baiocchi and Distaso (2003).

Appendix D. Advanced econometric analysis with free software

As mentioned in the main text, gretl offers a reasonably full selection of least-squares based estimators, plus a few additional estimators such as (binomial) logit and probit and Least Absolute Deviations. Advanced users may, however, find gretl's menu of statistical routines restrictive.

No doubt some advanced users will prefer to write their own statistical code in a fundamental computer language such as C, C++ or Fortran. Another option is to use a relatively high-level language that offers easy matrix manipulation and that already has numerous statistical routines built in, or available as add-on packages. If the latter option sounds attractive, and you are interested in using free, open source software, I would 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.

Also 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 statistical routines (e.g. generalized linear models, maximum likelihood estimation, time series methods). I have therefore supplied gretl with 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 either R --gui=gnome if an installation of the Gnome desktop (gnome.org) was detected at compile time, or xterm -e R if Gnome was not found. 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.

Census Bureau, Data Extraction Service

http://www.census.gov/ftp/pub/DES/www/welcome.html

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://www.gimp.org/~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.

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.

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.

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.