Introduction to Stata

Christopher F Baum

Faculty Micro Resource Center
Boston College

August 2011

Overview of the Stata environment

Stata is a full-featured statistical programming language for Windows, Mac OS X, Unix and Linux. It can be considered a “stat package,” like SAS, SPSS, RATS, or eViews.

Stata is available in several versions: Stata/IC (the standard version), Stata/SE (an extended version) and Stata/MP (for multiprocessing). The major difference between the versions is the number of variables allowed in memory, which is limited to 2,047 in standard Stata/IC, but can be much larger in Stata/SE or Stata/MP. The number of observations in any version is limited only by memory.

Stata/SE relaxes the Stata/IC constraint on the number of variables, while Stata/MP is the multiprocessor version, capable of utilizing 2, 4, 8... processors available on a single computer. Stata/IC will meet most users' needs; if you have access to Stata/SE or Stata/MP, you can use that program to create a subset of a large survey dataset with fewer than 2,047 variables. Stata runs on all 64-bit operating systems, and can access larger datasets on a 64-bit OS, which can address a larger memory space.

All versions of Stata provide the full set of features and commands: there are no special add-ons or 'toolboxes'. Each copy of Stata includes a complete set of manuals (over 6,000 pages) in PDF format, hyperlinked to the on-line help.

A Stata license may be used on any machine which supports Stata (Mac OS X, Windows, Linux): there are no machine-specific licenses for Stata versions 11 or 12. You may install Stata on a home and office machine, as long as they are not used concurrently. Licenses can be either annual or perpetual.

Stata works differently than some other packages in requiring that the entire dataset to be analyzed must reside in memory. This brings a considerable speed advantage, but implies that you may need more RAM (memory) on your computer. There are 32-bit and 64-bit versions of Stata, with the major difference being the amount of memory that the operating system can allocate to Stata (or any other application).

In some cases, the memory requirement may be of little concern. Stata is capable of holding data very efficiently, and even a quite sizable dataset (e.g., more than one million observations on 20–30 variables) may only require 500 Mb or so. You should take advantage of the `compress` command, which will check to see whether each variable may be held in fewer bytes than its current allocation.

For instance, indicator (dummy) variables and categorical variables with fewer than 100 levels can be held in a single byte, and integers less than 32,000 can be held in two bytes: see `help datatypes` for details. By default, floating-point numbers are held in four bytes, providing about seven digits of accuracy. Some other statistical programs routinely use eight bytes to store all numeric variables.

The memory available to Stata may be considerably less than the amount of RAM installed on your computer. If you have a 32-bit operating system, it does not matter that you might have 4 Gb or more of RAM installed; Stata will only be able to access about 1 Gb, depending on other processes' demands.

To make most effective use of Stata with large datasets, use a computer with a 64-bit operating system. Stata will automatically install a 64-bit version of the program if it is supported by the operating system. All Linux, Unix and Mac OS X computers today come with 64-bit operating systems.

Stata is eminently portable, and its developers are committed to cross-platform compatibility. Stata runs the same way on Windows, Mac OS X, Unix, and Linux systems. The only platform-specific aspects of using Stata are those related to native operating system commands: e.g. is the file to be accessed

```
C:\Stata\StataData\myfile.dta
or
/users/baum/statadata/myfile.dta
```

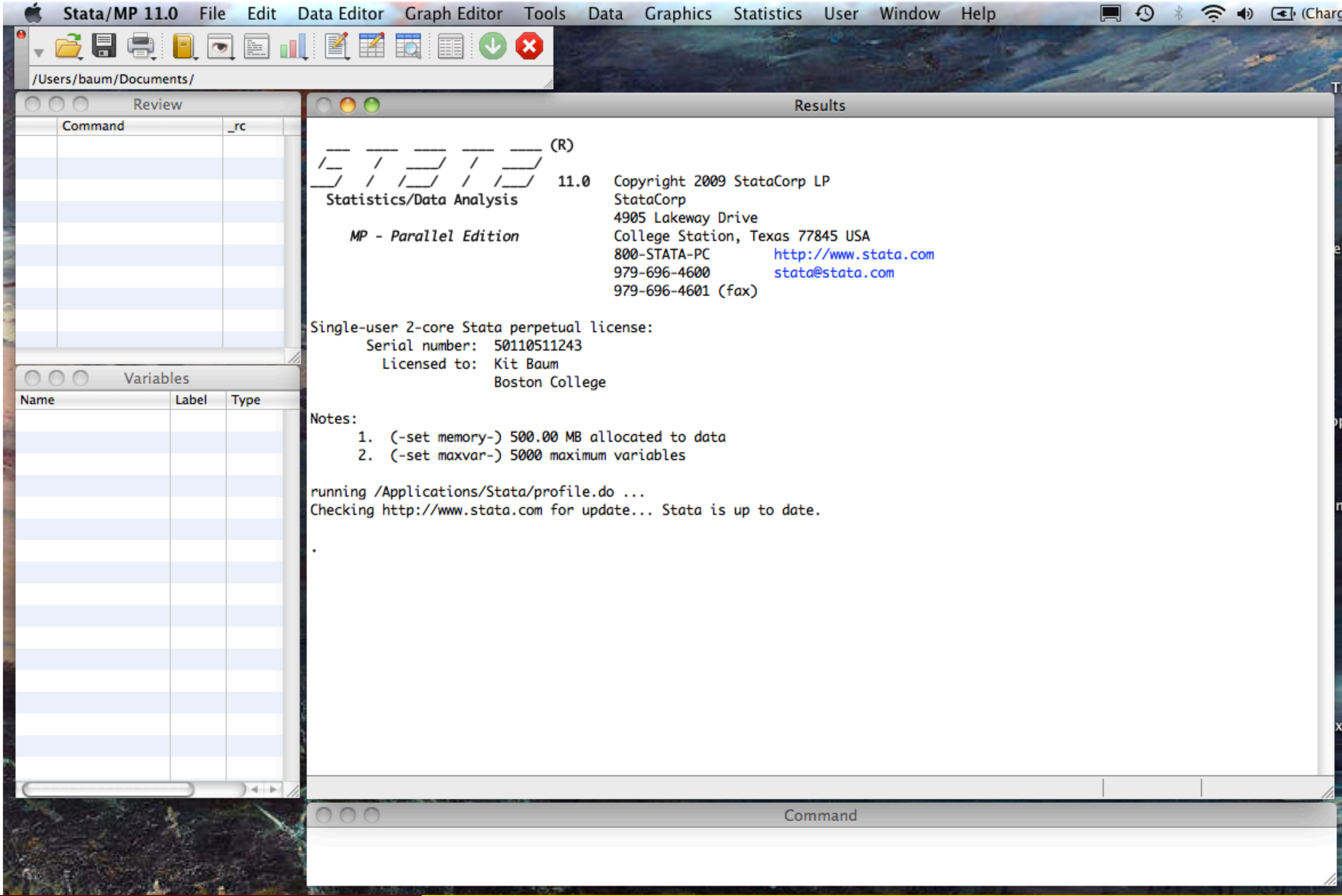
Perhaps unique among statistical packages, Stata's binary data files may be freely copied from one platform to any other, or even accessed over the Internet from any machine that runs Stata. You may store Stata's binary datafiles on a webserver (HTTP server) and open them on any machine with access to that server.

Stata's user interface

Stata has traditionally been a command-line-driven package that operates in a graphical (windowed) environment. Stata version 11 (released June 2009) and version 12 (released July 2011) contains a graphical user interface (GUI) for command entry via menus and dialogs. Stata may also be used in a command-line environment on a shared system (e.g., a Unix server) if you do not have a graphical interface to that system.

A major advantage of Stata's GUI system is that you always have the option of reviewing the command that has been entered in Stata's Review window. Thus, you may examine the syntax, revise it in the Command window and resubmit it. You may find that this is a more efficient way of using the program than relying wholly on dialogs.

Stata (version 11): default screen appearance:



```
  ___  ____  ____  ____  ____ (R)
 /__    /   ____/   /   ____/
___/   /   /___/   /   /___/   11.0   Copyright 2009 StataCorp LP
  Statistics/Data Analysis            StataCorp
                                      4905 Lakeway Drive
     MP - Parallel Edition            College Station, Texas 77845 USA
                                      800-STATA-PC        http://www.stata.com
                                      979-696-4600        stata@stata.com
                                      979-696-4601 (fax)

Single-user 2-core Stata perpetual license:
       Serial number:  50110511243
         Licensed to:  Kit Baum
                       Boston College

Notes:
      1.  (-set memory-) 500.00 MB allocated to data
      2.  (-set maxvar-) 5000 maximum variables

running /Applications/Stata/profile.do ...
Checking http://www.stata.com for update... Stata is up to date.

.
```

The Toolbar contains icons that allow you to Open and Save files, Print results, control Logs, and manipulate windows. Some very important tools allow you to open the Do-File Editor, the Data Editor and the Data Browser.

The Data Editor and Data Browser present you with a spreadsheet-like view of the data, no matter how large your dataset may be. The Do-File editor, as we will discuss, allows you to construct a file of Stata commands, or “do-file”, and execute it in whole or in part from the editor.

The Toolbar also contains an important piece of information: the Current Working Directory, or *cwd*. In the screenshot, it is listed as `/Users/Baum/Documents/` as I am working on a Mac OS X (Unix) laptop. The *cwd* is the directory to which any files created in your Stata session will be saved. Likewise, if you try to open a file and give its name alone, it is assumed to reside in the *cwd*. If it is in another location, you must change the *cwd* [File$-$ $>$Change Working Directory] or qualify its name with the directory in which it resides.

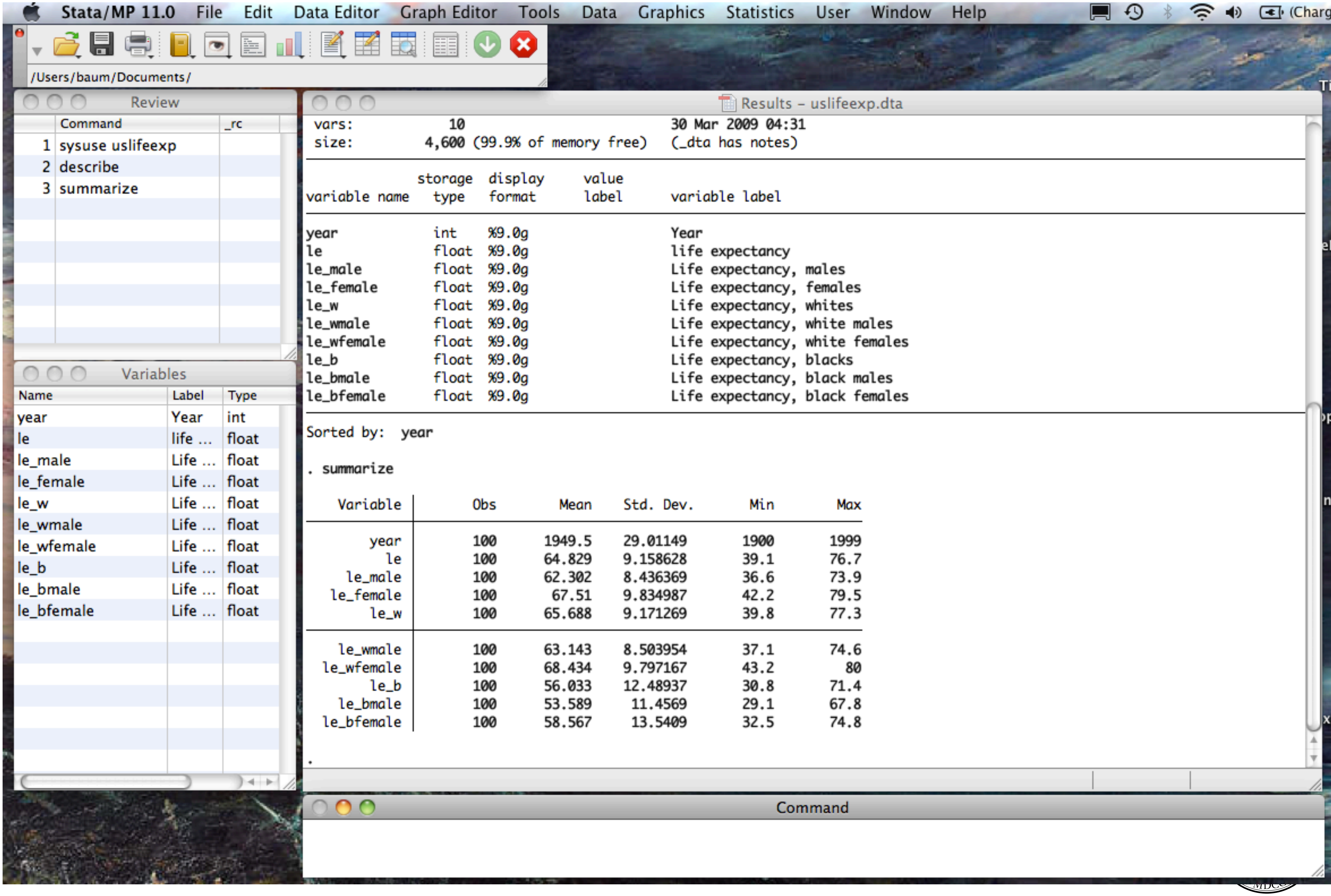
You generally will not want to locate or save files in the default *cwd*. A common strategy is to set up a directory for each project or task in a convenient location in the filesystem and change the *cwd* to that directory when working on that task. This can be automated in a do-file with the `cd` command.

There are four windows in the default interface: the Review, Results, Command and Variables window. You may alter the appearance of any window in the GUI using the Preferences$->$General dialog, and make those changes on a temporary or permanent basis.

As you might expect, you may type commands in the Command window. You may only enter one command in that window, so you should not try pasting a list of several commands. When a command is executed—with or without error—it appears in the Review window, and the results of the command (or an error message) appears in the Results window. You may click on any command in the Review window and it will reappear in the Command window, where it may be edited and resubmitted.

Once you have loaded data into the program, the Variables window will be populated with information on each variable. That information includes the variable name, its label (if any), its type and its format. This is a subset of information available from the `describe` command.

Let's look at the interface after I have loaded one of the datasets provided with Stata, `uslifeexp`, with the `sysuse` command and given the `describe` and `summarize` commands:



Review

	Command	_rc
1	sysuse uslifeexp	
2	describe	
3	summarize	

Variables

Name	Label	Type
year	Year	int
le	life ...	float
le_male	Life ...	float
le_female	Life ...	float
le_w	Life ...	float
le_wmale	Life ...	float
le_wfemale	Life ...	float
le_b	Life ...	float
le_bmale	Life ...	float
le_bfemale	Life ...	float

Results – uslifeexp.dta

```
  vars:            10                         30 Mar 2009 04:31
  size:         4,600 (99.9% of memory free)  (_dta has notes)
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
year            int    %9.0g                  Year
le              float  %9.0g                  life expectancy
le_male         float  %9.0g                  Life expectancy, males
le_female       float  %9.0g                  Life expectancy, females
le_w            float  %9.0g                  Life expectancy, whites
le_wmale        float  %9.0g                  Life expectancy, white males
le_wfemale      float  %9.0g                  Life expectancy, white females
le_b            float  %9.0g                  Life expectancy, blacks
le_bmale        float  %9.0g                  Life expectancy, black males
le_bfemale      float  %9.0g                  Life expectancy, black females
-------------------------------------------------------------------------------
Sorted by:  year

. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        year |       100      1949.5    29.01149       1900       1999
          le |       100      64.829    9.158628       39.1       76.7
     le_male |       100      62.302    8.436369       36.6       73.9
   le_female |       100       67.51    9.834987       42.2       79.5
        le_w |       100      65.688    9.171269       39.8       77.3
-------------+--------------------------------------------------------
    le_wmale |       100      63.143    8.503954       37.1       74.6
  le_wfemale |       100      68.434    9.797167       43.2         80
        le_b |       100      56.033    12.48937       30.8       71.4
     le_bmale |       100      53.589     11.4569       29.1       67.8
  le_bfemale |       100      58.567     13.5409       32.5       74.8

.
```

Notice that the three commands are listed in the Review window. If any had failed, the `_rc` column would contain a nonzero number, in red, indicating the error code. The Variables window contains the list of variables and their labels. The Results window shows the effects of `summarize`: for each variable, the number of observations, their mean, standard deviation, minimum and maximum. If there were any string variables in the dataset, they would be listed as having zero observations.

Try it out: type the commands

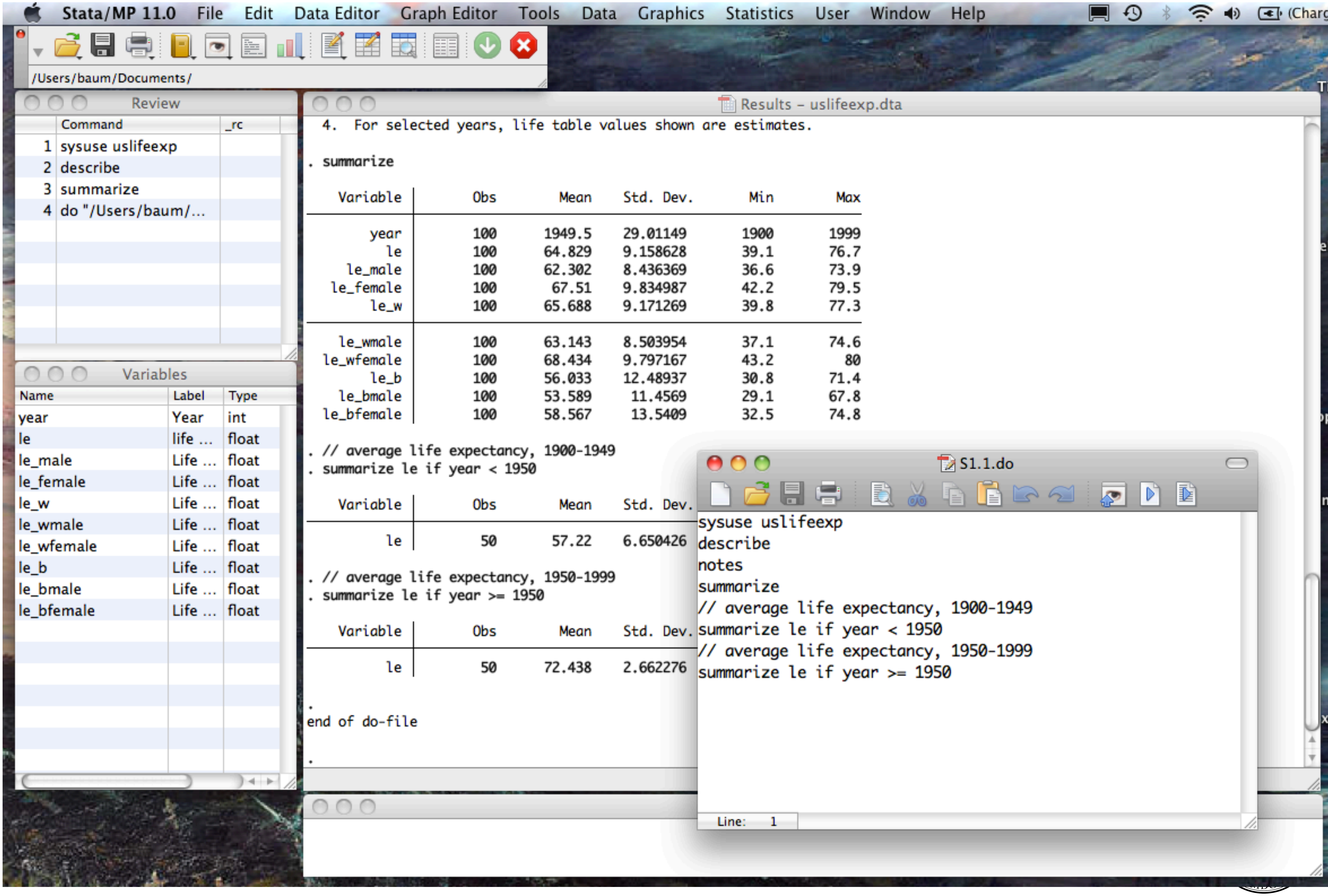
```
sysuse uslifeexp
describe
summarize
```

Take note of an important design feature of Stata. If you do not say what to `describe` or `summarize`, Stata assumes you want to perform those commands for every variable in memory, as shown here. As we shall see, this design principle holds throughout the program.

We may also write a do-file in the do-file editor and execute it. The Do-File Editor icon on the Toolbar brings up a window in which we may type those same three commands, as well as a few more:

```
sysuse uslifeexp
describe
summarize
notes
summarize le if year < 1950
summarize le if year >= 1950
```

After typing those commands into the window, the rightmost icon, with tooltip `Do`, may be used to execute them.



	Command	_rc
1	sysuse uslifeexp	
2	describe	
3	summarize	
4	do "/Users/baum/...	

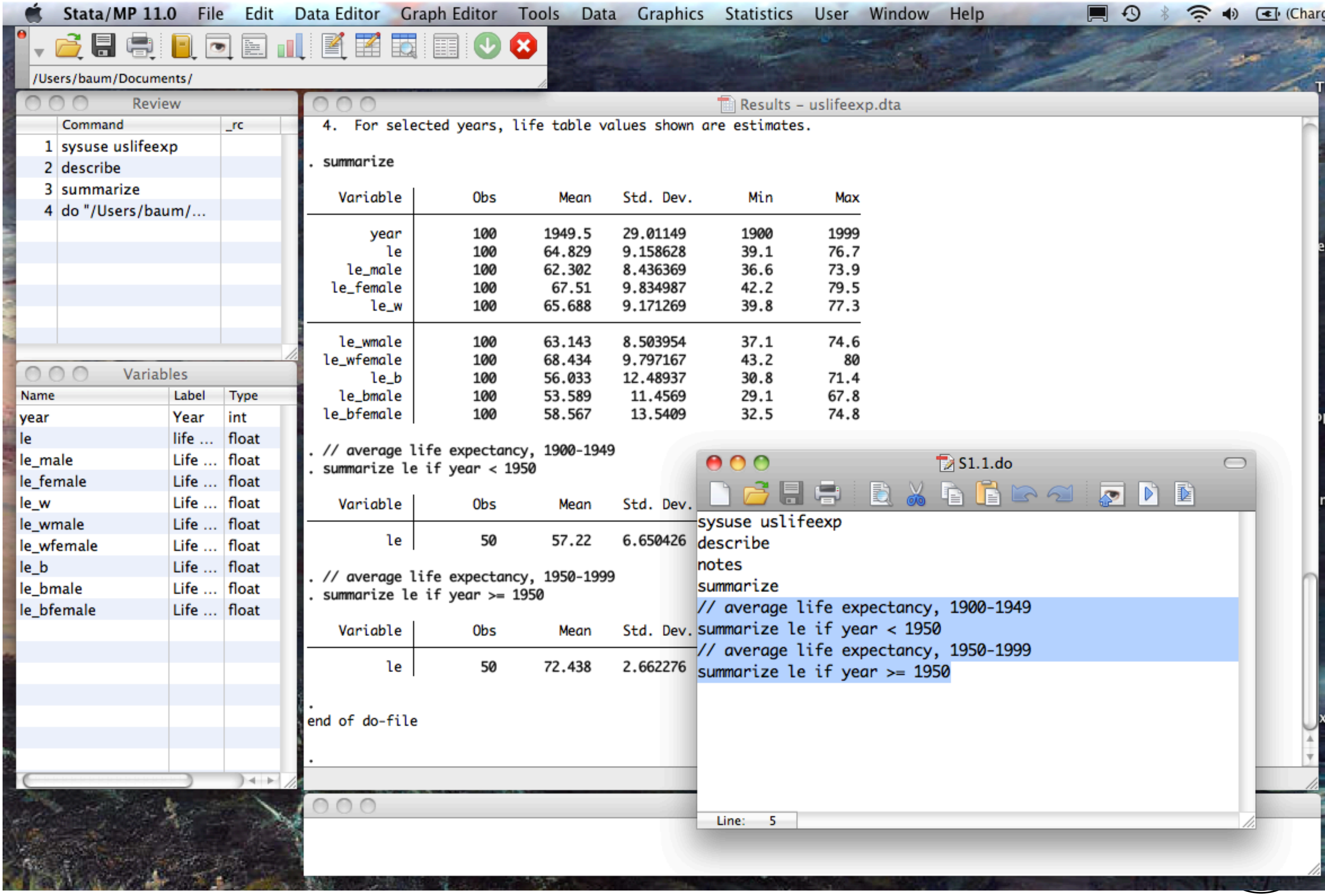
Name	Label	Type
year	Year	int
le	life ...	float
le_male	Life ...	float
le_female	Life ...	float
le_w	Life ...	float
le_wmale	Life ...	float
le_wfemale	Life ...	float
le_b	Life ...	float
le_bmale	Life ...	float
le_bfemale	Life ...	float

```
  4.  For selected years, life table values shown are estimates.

. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        year |       100      1949.5    29.01149       1900       1999
          le |       100      64.829    9.158628       39.1       76.7
     le_male |       100      62.302    8.436369       36.6       73.9
   le_female |       100       67.51    9.834987       42.2       79.5
        le_w |       100      65.688    9.171269       39.8       77.3
-------------+--------------------------------------------------------
    le_wmale |       100      63.143    8.503954       37.1       74.6
  le_wfemale |       100      68.434    9.797167       43.2         80
        le_b |       100      56.033    12.48937       30.8       71.4
     le_bmale |       100      53.589     11.4569       29.1       67.8
  le_bfemale |       100      58.567     13.5409       32.5       74.8

. // average life expectancy, 1900-1949
. summarize le if year < 1950

    Variable |       Obs        Mean    Std. Dev.
-------------+-------------------------------------
          le |        50       57.22    6.650426

. // average life expectancy, 1950-1999
. summarize le if year >= 1950

    Variable |       Obs        Mean    Std. Dev.
-------------+-------------------------------------
          le |        50      72.438    2.662276

.
end of do-file

.
```

```
sysuse uslifeexp
describe
notes
summarize
// average life expectancy, 1900-1949
summarize le if year < 1950
// average life expectancy, 1950-1999
summarize le if year >= 1950
```

In this do-file, I have included the `notes` command to display the notes saved with the dataset, and included two comment lines. There are several styles of comments available. In this style, anything on a line following a double slash (//) is ignored.

You may use the other icons in the Do-File Editor window to save your do-file (to the *cwd* or elsewhere), print it, or edit its contents. You may also select a portion of the file with the mouse and execute only those commands. Note that the tooltip changes to `Do Selected Lines`.



	Command	_rc
1	sysuse uslifeexp	
2	describe	
3	summarize	
4	do "/Users/baum/...	

Name	Label	Type
year	Year	int
le	life ...	float
le_male	Life ...	float
le_female	Life ...	float
le_w	Life ...	float
le_wmale	Life ...	float
le_wfemale	Life ...	float
le_b	Life ...	float
le_bmale	Life ...	float
le_bfemale	Life ...	float

```
  4.  For selected years, life table values shown are estimates.

. summarize

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        year |       100      1949.5    29.01149       1900       1999
          le |       100      64.829    9.158628       39.1       76.7
     le_male |       100      62.302    8.436369       36.6       73.9
   le_female |       100       67.51    9.834987       42.2       79.5
        le_w |       100      65.688    9.171269       39.8       77.3
-------------+--------------------------------------------------------
    le_wmale |       100      63.143    8.503954       37.1       74.6
  le_wfemale |       100      68.434    9.797167       43.2         80
        le_b |       100      56.033    12.48937       30.8       71.4
    le_bmale |       100      53.589     11.4569       29.1       67.8
  le_bfemale |       100      58.567     13.5409       32.5       74.8

. // average life expectancy, 1900-1949
. summarize le if year < 1950

    Variable |       Obs        Mean    Std. Dev.
-------------+-----------------------------------
          le |        50       57.22    6.650426

. // average life expectancy, 1950-1999
. summarize le if year >= 1950

    Variable |       Obs        Mean    Std. Dev.
-------------+-----------------------------------
          le |        50      72.438    2.662276

.
end of do-file

.
```

```
sysuse uslifeexp
describe
notes
summarize
// average life expectancy, 1900-1949
summarize le if year < 1950
// average life expectancy, 1950-1999
summarize le if year >= 1950
```

Try it out: use the Do-File Editor to open the do-file `S1.1.do`, and run the file.

Try selecting only those last four lines and run those commands.

The rightmost menu on the menu bar is labeled Help. From that menu, you can search for help on any command or feature. The Help Browser, which opens in a Viewer window, provides hyperlinks, in blue, to additional help pages. At the foot of each help screen, there are hyperlinks to the full manuals, which are accessible in PDF format. The links will take you directly to the appropriate page of the manual.

You may also search for help at the command line with `help` *command*. But what if you don't know the exact command name? Then you may use `search` or its expanded version, `findit`, each of which may be followed by one or several words.

Results from `search` are presented in the Results window, while `findit` results will appear in a Viewer window. Those commands will present results from a keyword database and from the Internet: for instance, FAQs from the Stata website, articles in the *Stata Journal* and *Stata Technical Bulletin*, and downloadable routines from the SSC Archive (about which more later) and user sites.

Try it out: when you are connected to the Internet, type the command

```
search baum, au
```

and then try

```
findit baum
```

Note the hyperlinks that appear on URLs for the books and journal articles, and on the individual software packages (e.g., `st0030_3`, `archlm`).

Stata is advertised as having three major strengths:

- data manipulation
- statistics
- graphics

Stata is an excellent tool for **data manipulation**: moving data from external sources into the program, cleaning it up, generating new variables, generating summary data sets, merging data sets and checking for merge errors, collapsing cross–section time-series data on either of its dimensions, reshaping data sets from “long” to “wide”, and so on. In this context, Stata is an excellent program for answering ad hoc questions about any aspect of the data.

In terms of **statistics**, Stata provides all of the standard univariate, bivariate and multivariate statistical tools, from descriptive statistics and t-tests through one-, two- and N-way ANOVA, regression, principal components, and the like. Stata's regression capabilities are full-featured, including regression diagnostics, prediction, robust estimation of standard errors, instrumental variables and two-stage least squares, seemingly unrelated regressions, vector autoregressions and error correction models, etc. It has a very powerful set of techniques for the analysis of limited dependent variables: logit, probit, ordered logit and probit, multinomial logit, and the like.

Stata's breadth and depth really shines in terms of its specialized statistical capabilities. These include environments for time-series econometrics (ARCH, ARIMA, ARFIMA, VAR, VEC), model simulation and bootstrapping, maximum likelihood estimation, GMM, and nonlinear least squares. Families of commands provide the leading techniques utilized in each of several categories:

- "`xt`" commands for cross-section/time-series or panel (longitudinal) data
- "`sem`" commands for structural equation modeling
- "`svy`" commands for the handling of survey data with complex sampling designs
- "`st`" commands for the handling of survival-time data with duration models

Stata **graphics** are excellent tools for exploratory data analysis, and can produce high-quality 2-D publication-quality graphics in several dozen different forms. Every aspect of graphics may be programmed and customized, and new graph types and graph “schemes” are being continuously developed. The programmability of graphics implies that a number of similar graphs may be generated without any “pointing and clicking” to alter aspects of the graphs. Stata 12 provides support for contour plots and ‘heatmaps’.

Stata’s update facility

One of Stata’s great strengths is that it can be updated over the Internet. Stata is actually a web browser, so it may contact Stata’s web server and enquire whether there are more recent versions of either Stata’s executable (the kernel) or the ado-files. This enables Stata’s developers to distribute bug fixes, enhancements to existing commands, and even entirely new commands during the lifetime of a given major release (including ‘dot-releases’ such as Stata 11.1).

Updates during the life of the version you own are free. You need only have a licensed copy of Stata and access to the Internet (which may be by proxy server) to check for and, if desired, download the updates.

Extensibility of official Stata

Another advantage of the command-line driven environment involves *extensibility*: the continual expansion of Stata's capabilities. A *command*, to Stata, is a verb instructing the program to perform some action.

Commands may be "built in" commands—those elements so frequently used that they have been coded into the "Stata kernel." A relatively small fraction of the total number of official Stata commands are built in, but they are used very heavily.

The vast majority of Stata commands are written in Stata's own programming language–the "ado-file" language. If a command is not built in to the Stata kernel, Stata searches for it along the `adopath`. Like the `PATH` in Unix, Linux or DOS, the `adopath` indicates the several directories in which an ado-file might be located. This implies that the "official" Stata commands are not limited to those coded into the kernel. *Try it out*: give the `adopath` command in Stata.

If Stata's developers tomorrow wrote a new command named "foobar", they would make two files available on their web site: `foobar.ado` (the ado-file code) and `foobar.sthlp` (the associated help file). Both are ordinary, readable ASCII text files. These files should be produced in a text editor, not a word processing program.

The importance of this program design goes far beyond the limits of official Stata. Since the adopath includes both Stata directories and other directories on your hard disk (or on a server's filesystem), you may acquire new Stata commands from a number of web sites. The *Stata Journal (SJ)*, a quarterly refereed journal, is the primary method for distributing user contributions. Between 1991 and 2001, the *Stata Technical Bulletin* played this role, and a complete set of issues of the *STB* are available on line at the Stata website.

The *SJ* is a subscription publication (articles more than three years old freely downloadable), but the `ado`- and `sthlp`-files may be freely downloaded from Stata's web site. The Stata `help` command accesses help on all installed commands; the Stata command `findit` will locate commands that have been documented in the STB and the SJ, and with one click you may install them in your version of Stata. Help for these commands will then be available in your own copy.

User extensibility: the SSC archive

But this is only the beginning. Stata users worldwide participate in the *StataList* listserv, and when a user has written and documented a new general-purpose command to extend Stata functionality, they announce it on the *StataList* listserv (to which you may freely subscribe: see Stata's web site).

Since September 1997, all items posted to `StataList` (over 1,300) have been placed in the Boston College Statistical Software Components (SSC) Archive in *RePEc* (Research Papers in Economics), available from IDEAS (`http://ideas.repec.org`) and EconPapers (`http://econpapers.repec.org`).

Any component in the SSC archive may be readily inspected with a web browser, using IDEAS' or EconPapers' search functions, and if desired you may install it with one command from the archive from within Stata. For instance, if you know there is a module in the archive named `mvsumm`, you could use `ssc describe mvsumm` to learn more about it, and `ssc install mvsumm` to install it if you wish. Anything in the archive can be accessed via Stata's `ssc` command: thus `ssc describe mvsumm` will locate this module, and make it possible to install it with one click.

Windows users should not attempt to download the materials from a web browser; it won't work.

Try it out: when you are connected to the Internet, type

```
ssc describe mvsumm
ssc install mvsumm
help mvsumm
```

The command `ssc new` lists, in the Stata Viewer, all SSC packages that have been added or modified in the last month. You may click on their names for full details. The command `ssc hot` reports on the most popular packages on the SSC Archive.

The Stata command `adoupdate` checks to see whether all packages you have downloaded and installed from the SSC archive, the *Stata Journal*, or other user-maintained `net from...` sites are up to date. `adoupdate` alone will provide a list of packages that have been updated. You may then use `adoupdate, update` to refresh your copies of those packages, or specify which packages are to be updated.

The importance of all this is that Stata is *infinitely extensible*. Any ado-file on your `adopath` is a full-fledged Stata command. Stata's capabilities thus extend far beyond the official, supported features described in the Stata manual to a vast array of additional tools.

Since the current directory is on the `adopath`, if you create an ado-file **hello.ado**:

```
program define hello
display "Stata says hello!"
end
exit
```

Stata will now respond to the command `hello`. It's that easy. *Try it out!*

For members of the Boston College community, Stata is available through ITS' applications server, `http://apps.bc.edu`. After downloading client software from this site, you may connect to the apps server from any BC-activated computer and run Stata in a window on your computer. It is actually running the Windows version of Stata/SE 11.2, but the interface and commands is almost identical to Stata for Mac OS X or Stata for Linux. Up to 50 users may access Stata on the apps server simultaneously. Results from your analysis may be stored on MyFiles, as the `m:` disk is automatically mapped to your account on `appstorage.bc.edu`, accessible from any web browser with authentication. If you are working from off campus, you must use set up VPN on your computer; see `http://www.bc.edu/help` for details.

If you would like your own copy of Stata, it is quite inexpensive. The vendor's GradPlan program makes the full version of Stata version 12 software available to BC faculty and students for $98.00 (one-year license) or $179.00 (perpetual license). This includes the full set of manuals in PDF format, hyperlinked to Stata's help system.

The "Small Stata" version is available to students for $49.00 for a one-year license. It contains all of Stata's commands, but can only handle a limited number of observations and variables (thus not recommended for Ph.D. students or Senior Honors Thesis students). GradPlan orders are made direct to Stata, with delivery from on-campus inventory.

Stata is very well supported by telephone and email technical support, as well as the more informal support provided by other users on **StataList**, the Stata listserv. The manuals are useful—particularly the User's Guide—but full details of the command syntax are available online, and in hypertext form in the GUI environment, with hyperlinks to the appropriate pages of the full documentation set of over a dozen manuals. The command `findit` *keyword* can also be used to locate Stata materials, including descriptions of built-in commands, Stata FAQs, and hundreds of user-written routines.

But why should I type commands?

But before we discuss the specifics to back up these claims, let's consider a meta-issue: why would you want to learn how to use a command-line-driven package? Isn't that ever so 20th century?

Stata may be used in an interactive mode, and those learning the package may wish to make use of the menu system. But when you execute a command from a pull-down menu, it records the command that you could have typed in the Review window, and thus you may learn that with experience you could type that command (or modify it and resubmit it) more quickly than by use of the menus.

Let us consider a couple of reasons why a command-line-driven package makes for an effective and efficient research strategy.

Reproducibility

First, the important issue of reproducibility. If you are conducting scientific research, you must be able to reproduce your results. Ideally, anyone with your programs and data should be able to do so without your assistance. If you cannot produce such reproducible research findings, it can be argued that you are not following the scientific method, nor is your work conforming to ethical standards of research.

A thorough discussion of this issue is covered in the webpage, `http://fmwww.bc.edu/GStat/docs/pointclick.html`.

In a computer program where all actions are point and click, such as a spreadsheet, who can say how you arrived at a certain set of results? Unless every step of your transformations of the data can be retraced, how can you find exactly how the sample you are employing differs from the raw data? A command-driven program is capable of this level of reproducibility, we should all instill this level of rigor in our research practices.

Reproducibility also makes it very easy to perform an alternate analysis of a particular model. What would happen if we added this interaction, or introduced this additional variable, or decided to handle zero values as missing? Even if many steps have been taken since the basic model was specified, it is easy to go back and produce a variation on the analysis if all the work is represented by a series of programs.

Transportability

Stata binary files may be easily transformed into SPSS or SAS files with the third-party application Stat/Transfer. Stat/Transfer is available for Windows and Mac OS X systems as well as on various Unix systems on campus. Personal copies of Stat/Transfer version 11 (which handles Stata versions 6, 7, 8, 9, 10, 11 and 12 datafiles) are available at a discounted academic rate of $69.00 through the Stata GradPlan.

Stat/Transfer can also transfer SAS, SPSS and many other file formats into Stata format, without loss of variable labels, value labels, and the like. It can also be used to create a manageable subset of a very large Stata file (such as those produced from survey data) by selecting only the variables you need. It is a very useful tool.

Programmability of tasks

Stata may be used in an interactive mode, and those learning the package may wish to make use of the menu system. But when you execute a command from a pull-down menu, it records the command that you could have typed in the Review window, and thus you may learn that with experience you could type that command (or modify it and resubmit it) more quickly than by use of the menus.

Stata makes reproducibility very easy through a log facility, the ability to generate a command log (containing only the commands you have entered), and the do-file editor which allows you to easily enter, execute and save sequences of commands, or program fragments.

Going one step further, if you use the do-file editor to create a sequence of commands, you may save that do-file and reuse it tomorrow, or use it as the starting point for a similar set of data management or statistical operations. Working in this way promotes reproducibility, which makes it very easy to perform an alternate analysis of a particular model. Even if many steps have been taken since the basic model was specified, it is easy to go back and produce a variation on the analysis if all the work is represented by a series of programs.

One of the implications of the concern for reproducible work: avoid altering data in a non-auditable environment such as a spreadsheet. Rather, you should transfer external data into the Stata environment as early as possible in the process of analysis, and only make permanent changes to the data with do-files that can give you an audit trail of every change made to the data.

Programmable tasks are supported by *prefix commands*, as we will soon discuss, that provide implicit loops, as well as explicit looping constructs such as the `forvalues` and `foreach` commands.

To use these commands you must understand Stata's concepts of local and global *macros*. Note that the term macro in Stata bears no resemblance to the concept of an Excel macro. A macro, in Stata, is an alias to an object, which may be a number or string.

Local macros and scalars

In programming terms, *local macros* and *scalars* are the “variables” of Stata programs (not to be confused with the variables of the data set). The distinction: a local macro can contain a string, while a scalar can contain a single number (at maximum precision). You should use these constructs whenever possible to avoid creating variables with constant values merely for the storage of those constants. This is particularly important when working with large data sets.

When you want to work with a scalar object—such as a counter in a `foreach` or `forvalues` command—it will involve defining and accessing a local macro. As we will see, all Stata commands that compute results or estimates generate one or more objects to hold those items, which are saved as numeric scalars, local macros (strings or numbers) or numeric matrices.

The local macro

The *local macro* is an invaluable tool for do-file authors. A local macro is created with the `local` statement, which serves to name the macro and provide its content. When you next refer to the macro, you extract its value by *dereferencing* it, using the backtick (‘) and apostrophe (’) on its left and right:

```
local george 2
local paul = ‘george’ + 2
```

In this case, I use an equals sign in the second local statement as I want to *evaluate* the right-hand side, as an arithmetic expression, and store it in the macro `paul`. If I did not use the equals sign in this context, the macro `paul` would contain the string `2 + 2`.

forvalues and foreach

In other cases, you want to *redefine* the macro, not evaluate it, and you should not use an equals sign. You merely want to take the contents of the macro (a character string) and alter that string. The two key programming constructs for repetition, `forvalues` and `foreach`, make use of local macros as their “counter”. For instance:

```
forvalues i=1/10 {
     summarize PRweek`i'
}
```

Note that the value of the local macro `i` is used within the body of the loop when that counter is to be referenced. Any Stata *numlist* may appear in the `forvalues` statement. Note also the curly braces, which must appear at the end of their lines.

In many cases, the `forvalues` command will allow you to substitute explicit statements with a single loop construct. By modifying the range and body of the loop, you can easily rewrite your do-file to handle a different case.

The `foreach` command is even more useful. It defines an iteration over any one of a number of lists:

- the contents of a varlist (list of existing variables)
- the contents of a newlist (list of new variables)
- the contents of a numlist (list of integers)
- the separate words of a macro
- the elements of an arbitrary list

For example, we might want to `summarize` each of these variables' detailed statistics from this World Bank data set:

```
sysuse lifeexp
foreach v of varlist popgrowth lexp gnppc {
      summarize ‘v’, detail
}
```

Or, run a regression on variables for each region, and graph the data and fitted line:

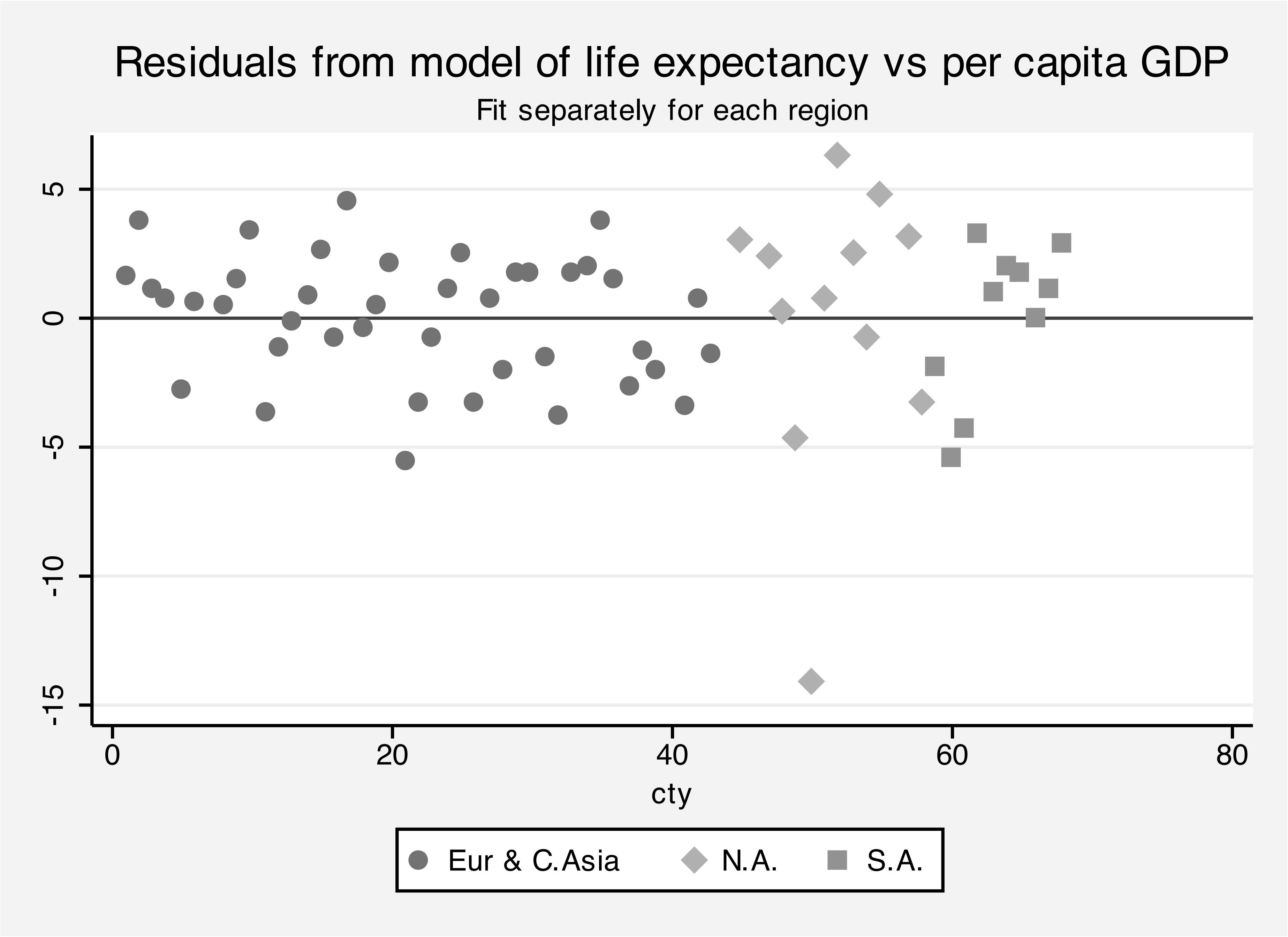
```
levelsof region, local(regid)
foreach c of local regid {
local rr : label region ‘c’
    regress lexp gnppc if region ==‘c’
    twoway (scatter lexp  gnppc if region ==‘c’) ///
           (lfit lexp gnppc if region ==‘c’, ///
           ti(Region: ‘rr’) name(fig‘c’, replace))
 }
```

A local macro can be built up by redefinition:

```
local alleps
foreach c of local regid {
regress lexp gnppc if region ==‘c’
predict double eps‘c’ if e(sample), residual
local alleps "‘alleps’  eps‘c’"
}
```

Within the loop we redefine the macro `alleps` (as a double-quoted string) to contain itself and the name of the residuals from that region’s regression. We could then use the macro `alleps` to generate a graph of all three regions’ residuals:

```
gen cty = _n
scatter `alleps´ cty, yline(0) scheme(s2mono) legend(rows(1)) ///
  ti("Residuals from model of life expectancy vs per capita GDP") ///
  t2("Fit separately for each region")
```


Residuals from model of life expectancy vs per capita GDP
Fit separately for each region
5
0
-5
-10
-15
0
20
40
60
80
cty
● Eur & C.Asia
◆ N.A.
■ S.A.

Global macros

Stata also supports *global macros*, which are referenced by a different syntax (`$country` rather than `` `country' ``). Global macros are useful when particular definitions (e.g., the default working directory for a particular project) are to be referenced in several do-files that are to be executed. However, the creation of persistent objects of global scope can be dangerous, as global macro definitions are retained for the entire Stata session. One of the advantages of local macros is that they disappear when the do-file or ado-file in which they are defined finishes execution.

Stata's command syntax

We now consider the form of Stata commands. One of Stata's great strengths, compared with many statistical packages, is that its command syntax follows strict rules: in grammatical terms, there are no irregular verbs. This implies that when you have learned the way a few key commands work, you will be able to use many more without extensive study of the manual or even on-line help. The `search` command will allow you to find the command you need by entering one or more keywords, even if you do not know the command's name.

The fundamental syntax of all Stata commands follows a *template*. Not all elements of the template are used by all commands, and some elements are only valid for certain commands. But where an element appears, it will appear in the same place, following the same grammar. Like Unix or Linux, Stata is case sensitive. Commands must be given in lower case. For best results, keep all variable names in lower case to avoid confusion.

The general syntax of a Stata command is:

```
[prefix_cmd:] cmdname [varlist] [=exp]
                      [if exp] [in range]
                      [weight] [using...] [,options]
```

where elements in square brackets are optional for some commands.

In some cases, only the `cmdname` itself is required. `describe` without arguments gives a description of the current contents of memory (including the identifier and timestamp of the current dataset), while `summarize` without arguments provides summary statistics for all (numeric) variables. Both may be given with a `varlist` specifying the variables to be considered.

What are the other elements?

The varlist

varlist is a list of one or more variables on which the command is to operate: the subject(s) of the verb. Stata works on the concept of a single set of variables currently defined and contained in memory, each of which has a name. As `desc` will show you, each variable has a data type (various sorts of integers and reals, and string variables of a specified maximum length). The varlist specifies which of the defined variables are to be used in the command.

The order of variables in the dataset matters, since you can use hyphenated lists to include all variables between first and last. (The `order` and `move` commands can alter the order of variables.) You can also use “wildcards” to refer to all variables with a certain prefix. If you have variables pop60, pop70, pop80, pop90, you can refer to them in a varlist as `pop*` or `pop?0`.

The exp clause

The *exp* clause is used in commands such as `generate` and `replace` where an algebraic expression is used to produce a new (or updated) variable. In algebraic expressions, the operators ==, &, | and ! are used as equal, AND, OR and NOT, respectively. The $\wedge$ operator is used to denote exponentiation. The + operator is overloaded to denote concatenation of character strings.

The if and in clauses

Stata differs from several common programs in that Stata commands will automatically apply to all observations currently defined. You need not write explicit loops over the observations. You can, but it is usually bad programming practice to do so. Of course you may want not to refer to all observations, but to pick out those that satisfy some criterion. This is the purpose of the *if exp* and *in range* clauses. For instance, we might:

```
sort price
list make price in 1/5
```

to determine the five cheapest cars in auto.dta. The 1/5 is a *numlist*: in this case, a list of observation numbers. ℓ is the last observation, thus *list make price in -5/*ℓ will list the five most expensive cars in auto.dta.

Even more commonly, you may employ the *if exp* clause. This restricts the set of observations to those for which the “exp”, a Boolean expression, evaluates to true. Stata’s missing value codes are greater than the largest positive number, so that the last command would avoid listing cars for which the price is missing.

```
list make price if foreign==1
```

lists only foreign cars, and

```
list make price if price > 10000 & price <.
```

lists only expensive cars (in 1978 prices!) Note the double equal in the *exp*. A single equal sign, as in the C language, is used for assignment; double equal for comparison.

The using clause

Some commands access files: reading data from external files, or writing to files. These commands contain a *using* clause, in which the filename appears. If a file is being written, you must specify the "replace" option to overwrite an existing file of that name.

Stata's own binary file format, the **.dta** file, is cross-platform compatible, even between machines with different byte orderings (low-endian and high-endian). A .dta file may be moved from one computer to another using **ftp** (in binary transfer mode).

To bring the contents of an existing Stata file into memory, the command:

`use` *file* [,`clear`]

is employed (`clear` will empty the current contents of memory). You must make sufficient memory available to Stata to load the entire file, since Stata's speed is largely derived from holding the entire data set in memory. Consult *Getting Started...* for details on adjusting the memory allocation on your computer, since it differs by operating system.

Reading and writing binary (.dta) files is much faster than dealing with text (ASCII) files (with the `insheet` or `infile` commands), and permits variable labels, value labels, and other characteristics of the file to be saved along with the file. To write a Stata binary file, the command

`save` *file* [,`replace`]

is employed. The `compress` command can be used to economize on the disk space (and memory) required to store variables.

Stata's version 10 and 11 datasets cannot be read by version 8 or 9; to create a compatible dataset, use `saveold`.

The amazing thing about “use filename” is that it is by no means limited to the files on your hard disk. Since Stata is a web browser,

```
webuse klein
```

or

```
use http://fmwww.bc.edu/ec-p/data/Wooldridge/crime1.dta
```

will read these datasets into Stata’s memory over the web.

The `type` command can display any text file, whether on your hard disk or over the Web; thus

```
type http://fmwww.bc.edu/ec-p/data/Wooldridge/crime1.des
```

will display the codebook for this file, and

```
copy http://fmwww.bc.edu/ec-p/data/Wooldridge/crime1.des crime.codebook
```

will make a copy of the codebook on your own hard disk.

When you have `use`d a dataset over the Web, you have loaded it into memory in your desktop Stata. You cannot save it to the Web, but can save the data to your own hard disk. The advantages of this feature for instructional and collaborative research should be clear. Students may be given a URL from which their assigned data are to be accessed; it matters not whether they are using Stata for Windows, Macintosh, Linux, or Unix.

The options clause

Many commands make use of options (such as `clear` on `use`, or `replace` on `save`). All options are given following a single comma, and may be given in any order. Options, like commands, may generally be abbreviated (with the notable exception of `replace`).

Prefix commands

A number of Stata commands can be used as *prefix commands*, preceding a Stata command and modifying its behavior. The most commonly employed is the *by prefix*, which repeats a command over a set of categories. The *statsby:* prefix repeats the command, but collects statistics from each category. The *rolling:* prefix runs the command on moving subsets of the data (usually time series).

Several other command prefixes: *simulate:*, which simulates a statistical model; *bootstrap:*, allowing the computation of bootstrap statistics from resampled data; and *jackknife:*, which runs a command over jackknife subsets of the data. The *svy:* prefix can be used with many statistical commands to allow for survey sample design. See my separate slideshow on *Monte Carlo Simulation in Stata*.

The by prefix

You can often save time and effort by using the *by* prefix. When a command is prefixed with a *bylist*, it is performed repeatedly for each element of the variable or variables in that list, each of which must be categorical. For instance,

```
by foreign:  summ price
```

will provide descriptive statistics for both foreign and domestic cars. If the data are not already sorted by the bylist variables, the prefix `bysort` should be used. The option `,total` will add the overall summary.

What about a classification with several levels, or a combination of values?

```
bysort rep78: summ price
```

```
bysort rep78 foreign: summ price
```

This is a very handy tool, which often replaces explicit loops that must be used in other programs to achieve the same end.

The by prefix should not be confused with the by *option* available on some commands, which allows for specification of a grouping variable: for instance

```
ttest price, by(foreign)
```

will run a t-test for the difference of sample means across domestic and foreign cars.

Another useful aspect of *by* is the way in which it modifies the meanings of the observation number symbol. Usually *_n* refers to the current observation number, which varies from 1 to *_N*, the maximum defined observation. Under a bylist, *_n* refers to the observation within the bylist, and *_N* to the total number of observations for that category. This is often useful in creating new variables.

For instance, if you have individual data with a family identifier, these commands might be useful:

```
sort famid age
by famid: gen famsize = _N
by famid: gen birthorder = _N - _n +1
```

Here the `famsize` variable is set to _N, the total number of records for that family, while the `birthorder` variable is generated by sorting the family members' ages within each family.

Missing values

Missing value codes in Stata appear as the dot (.) in printed output (and a string missing value code as well: “”, the null string). It takes on the largest possible positive value, so in the presence of missing data you do not want to say

`generate hiprice = (price > 10000)`, but rather

`generate hiprice = (price > 10000 & price <.)`

which then generates a “dummy variable” for high-priced cars (for which price data are complete, with prices “less than missing”).

As of version 8, Stata allows for multiple missing value codes (`.a, .b, .c, ..., .z`).

Display formats

Each variable may have its own default display format. This does not alter the contents of the variable, but affects how it is displayed. For instance, `%9.2f` would display a two-decimal-place real number. The command

```
format varname %9.2f
```

will save that format as the default format of the variable, and

```
format date %tm
```

will format a Stata date variable into a monthly format (e.g., `1998m10`).

Variable labels

Each variable may have its own variable label. The variable label is a character string (maximum 80 characters) which describes the variable, associated with the variable via

```
label variable varname "text"
```

Variable labels, where defined, will be used to identify the variable in printed output, space permitting.

Value labels

Value labels associate numeric values with character strings. They exist separately from variables, so that the same mapping of numerics to their definitions can be defined once and applied to a set of variables (e.g. 1=very satisfied...5=not satisfied may be applied to all responses to questions about consumer satisfaction). Value labels are saved in the dataset. For example:

```
label define sexlbl 0 male 1 female
label values sex sexlbl
```

If value labels are defined, they will be displayed in printed output instead of the numeric values.

Generating new variables

The command `generate` is used to produce new variables in the dataset, whereas `replace` must be used to revise an existing variable (and `replace` must be spelled out). The syntax just demonstrated is often useful if you are trying to generate indicator variables, or dummies, since it combines a `generate` and `replace` in a single command.

A full set of functions are available for use in the `generate` command, including the standard mathematical functions, recode functions, string functions, date and time functions, and specialized functions (`help functions` for details). Note that `generate`'s `sum()` function is a running sum.

The egen command

Stata is not limited to using the set of defined functions. The `egen` (*e*xtended *gen*erate) command makes use of functions written in the Stata ado-file language, so that *_gzap.ado* would define the extended generate function `zap()`. This would then be invoked as

```
egen newvar = zap(oldvar)
```

which would do whatever `zap` does on the contents of oldvar, creating the new variable newvar.

A number of `egen` functions provide row-wise operations similar to those available in a spreadsheet: row sum, row average, row standard deviation, etc.

Time series operators

The `D.`, `L.`, and `F.` operators may be used under a timeseries calendar (including in the context of panel data) to specify first differences, lags, and leads, respectively. These operators understand missing data, and numlists: e.g. `L(1/4).x` is the first through fourth lags of `x`, while `L2D.x` is the second lag of the first difference of the `x` variable.

It is important to use the time series operators to refer to lagged or led values, rather than referring to the observation number (e.g., `_n-1`). The time series operators respect the time series calendar, and will not mistakenly compute a lag or difference from a prior period if it is missing. This is particularly important when working with panel data to ensure that references to one individual do not reach back into the prior individual's data.

In Stata 12, you may define a custom business-daily calendar that takes account of weekends, holidays, etc.

Mata: Matrix programming language

As of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with all of the capabilities of MATLAB, Ox or GAUSS. Mata can be used interactively, or Mata functions can be developed to be called from Stata. A large library of mathematical and matrix functions is provided in Mata, including equation solvers, decompositions, eigensystem routines and probability density functions. Mata functions can access Stata's variables and can work with virtual matrices ("views") of a subset of the data in memory. Mata also supports file input/output.

Mata code is automatically compiled into bytecode, like Java, and can be stored in object form or included in-line in a Stata do-file or ado-file. Mata code runs many times faster than the interpreted ado-file language, providing significant speed enhancements to many computationally burdensome tasks. See my separate slideshow *Mata in Stata*.

Estimation commands

All estimation commands share the same syntax. Multiple equation estimation commands use a list of equations, rather than a *varlist*, where equations are defined in parenthesized *varlist*s. Most estimation commands allow the use of various kinds of weights.

Estimation commands display confidence intervals for the coefficients, and tests of the most common hypotheses. More complex hypotheses may be analyzed with the `test` and `lincom` commands; for nonlinear hypothesis, `testnl` and `nlcom` may be applied, making use of the delta method.

Robust (Huber/White) estimates of the covariance matrix are available for almost all estimation commands by employing the `robust` option.

Predicted values and residuals may be obtained after any estimation command with the `predict` command. For nonlinear estimators, `predict` will produce other statistics as well (e.g. the log of the odds ratio from logistic regression). The `mfx` command may be used to generate marginal effects, including elasticities and semi–elasticities, for any estimation command.

All estimation commands “leave behind” results of estimation in the `e()` array, where they may be inspected with `ereturn list`. Any item here, including scalars such as R^2 and *RMSE*, the coefficient vector, and the estimated variance-covariance matrix, may be saved for use in later calculations.

The `estimates` suite of commands allow you to store the results of a particular estimation for later use in a Stata session. For instance, after the commands

```
regress price mpg length turn
estimates store model1
regress price weight length displacement
estimates store model2
regress price weight length gear_ratio foreign
estimates store model3
```

the command

```
estimates table model1 model2 model3
```

will produce a nicely-formatted table of results. Options on `estimates table` allow you to control precision, whether standard errors or t-values are given, significance stars, summary statistics, etc.

For example:

```
estimates table model1 model2 model3, b(%10.3f)
se(%7.2f) stats(r2 rmse N) title(Some models of auto
price)
```

Although `estimates table` can produce a summary table quite useful for evaluating a number of specifications, we often want to produce a publication-quality table for inclusion in a word processing document. Ben Jann's `estout` command processes stored `estimates` and provides a great deal of flexibility in generating such a table.

Programs in the `estout` suite can produce tab-delimited tables for MS Word, HTML tables for the web, and—my favorite—LaTeX tables for professional papers. In the LaTeX output format, `estout` can generate Greek letters, sub- and superscripts, and the like. `estout` is available from SSC, with extensive on-line help, and was described in the *Stata Journal*, 5(3), 2005 and 7(2), 2007. It has its own website at `http://repec.org/bocode/e/estout`.

From the example above, rather than using `estimates save` and `estimates table` we use Jann's `eststo` (store) and `esttab` (table) commands:

```
eststo clear
eststo: reg price mpg length turn
eststo: reg price weight length displacement
eststo: reg price weight length gear_ratio foreign
esttab using auto1.tex, stats(r2 bic N)  ///
subst(r2 \$R^2$) title(Models of auto price) ///
replace
```

Table 1: Models of auto price

	(1) price	(2) price	(3) price
mpg	-186.7*		
	(-2.13)		
length	52.58	-97.63*	-88.03*
	(1.67)	(-2.47)	(-2.65)
turn	-199.0		
	(-1.44)		
weight		4.613**	5.479***
		(3.30)	(5.24)
displacement		0.727	
		(0.10)	
gear_ratio			-669.1
			(-0.72)
foreign			3837.9***
			(5.19)
_cons	8148.0	10440.6*	7041.5
	(1.35)	(2.39)	(1.46)
R^2	0.251	0.348	0.552
bic	1387.2	1377.0	1353.5
N	74	74	74

t statistics in parentheses

* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

File handling

File extensions usually employed (but not required) include:

`.ado`	`automatic do-file (defines a Stata command)`
`.dct`	`data dictionary, optionally used with infile`
`.do`	`do-file (user program)`
`.dta`	`Stata binary dataset`
`.gph`	`graphics output file (binary)`
`.log`	`text log file`
`.smcl`	`SMCL (markup) log file, for use with Viewer`
`.raw`	`ASCII data file`
`.sthlp`	`Stata help file`

These extensions need not be given (except for `.ado`). If you use other extensions, they must be explicitly specified.

Comma-separated (CSV) files or tab-delimited data files may be read very easily with the `insheet` command—which despite its name does not read spreadsheet files. If your file has variable names in the first row that are valid for Stata, they will be automatically used (rather than default variable names). You usually need not specify whether the data are tab- or comma-delimited. Note that `insheet` cannot read space-delimited data (or character strings with embedded spaces, unless they are quoted).

If the file extension is `.raw`, you may just use

```
insheet using filename
```

to read it. If other file extensions are used, they must be given:

```
insheet using filename.csv
insheet using filename.txt
```

In Stata version 12, Excel spreadsheets (either `.xls` or `.xlsx` can be imported into Stata directly, either as entire worksheets or as cell ranges. As with `insheet`, if valid Stata variable names appear in the first row of a worksheet, you may specify that they should be used when the worksheet is imported.

A free-format ASCII text file with space-, tab-, or comma-delimited data may be read with the `infile` command. The missing-data indicator (.) may be used to specify that values are missing.
The command must specify the variable names. Assuming `auto.raw` contains numeric data,

```
infile price mpg displacement using auto
```

will read it. If a file contains a combination of string and numeric values in a variable, it should be read as string, and `encode` used to convert it to numeric with string value labels.

If some of the data are string variables without embedded spaces, they must be specified in the command:

```
infile str3 country price mpg displacement using auto2
```

would read a three-letter country of origin code, followed by the numeric variables. The number of observations will be determined from the available data.

The `infile` command may also be used with fixed-format data, including data containing undelimited string variables, by creating a dictionary file which describes the format of each variable and specifies where the data are to be found. The dictionary may also specify that more than one record in the input file corresponds to a single observation in the data set.

If data fields are not delimited—for instance, if the sequence ‘102’ should actually be considered as three integer variables. A `dictionary` must be used to define the variables’ locations. The `byvariable()` option allows a variable-wise dataset to be read, where one specifies the number of observations available for each series.

An alternative to infile with a dictionary is the `infix` command, which presents a syntax similar to that used by SAS for the definition of variables' data types and locations in a fixed-format ASCII data set: that is, a data file in which certain columns contain certain variables. The `_column()` directive allow contents of a fixed-format data file to be retrieved selectively.

`infix` may also be used for more complex record layouts where one individual's data are contained on several records in an ASCII file.

A logical condition may be used on the `infile` or `infix` commands to read only those records for which certain conditions are satisfied: i.e.

```
infix using employee if sex=="M"
infile price mpg using auto in 1/20
```

where the latter will read only the first 20 observations from the external file. This might be very useful when reading a large data set, where one can check to see that the formats are being properly specified on a subset of the file.

If your data are already in the internal format of SAS, SPSS, Excel, GAUSS, MATLAB, or a number of other packages, the best way to get it into Stata is by using the third-party product Stat/Transfer. Stat/Transfer will preserve variable labels, value labels, and other aspects of the data, and can be used to convert a Stata binary file into other packages' formats. It can also produce subsets of the data (selecting variables, cases or both) so as to generate an extract file that is more manageable. This is particularly important when the 2,047-variable limit on standard Stata data sets is encountered. Stat/Transfer is well documented, with on-line help available in both Windows, Mac OS X and Unix versions, and an extensive manual.

Combining data sets

In many empirical research projects, the raw data to be utilized are stored in a number of separate files: separate “waves” of panel data, timeseries data extracted from different databases, and the like. Stata only permits a single data set to be accessed at one time. How, then, do you work with multiple data sets? Several commands are available, including `append`, `merge`, and `joinby`.

The `append` command combines two Stata-format data sets that possess variables in common, adding observations to the existing variables. The same variables need not be present in both files, as long as a subset of the variables are common to the “master” and “using” data sets. It is important to note that “`PRICE`" and “`price`” are different variables, and one will not be appended to the other.

The merge command

We now describe the `merge` command, which is Stata's basic tool for working with more than one dataset. Its syntax changed considerably in Stata version 11.

The merge command takes a first argument indicating whether you are performing a *one-to-one*, *many-to-one*, *one-to-many* or *many-to-many* merge using specified key variables. It can also perform a one-to-one merge by observation.

Like the `append` command, the `merge` works on a “master” dataset—the current contents of memory—and a single “using” dataset (prior to Stata 11, you could specify multiple using datasets). One or more key variables are specified, and you need not sort either dataset prior to merging.

The distinction between “master” and “using” is important. When the same variable is present in each of the files, Stata’s default behavior is to hold the master data inviolate and discard the using dataset’s copy of that variable. This may be modified by the `update` option, which specifies that non-missing values in the using dataset should replace missing values in the master, and the even stronger `update replace`, which specifies that non-missing values in the using dataset should take precedence.

A “*one-to-one*” merge (written `merge 1:1`) specifies that each record in the using data set is to be combined with one record in the master data set. This would be appropriate if you acquired additional variables for the same observations.

In any use of `merge`, a new variable, `_merge`, takes on integer values indicating whether an observation appears in the master only, the using only, or appears in both. This may be used to determine whether the merge has been successful, or to remove those observations which remain unmatched (e.g. merging a set of households from different cities with a comprehensive list of postal codes; one would then discard all the unused postal code records). The `_merge` variable must be dropped before another `merge` is performed on this data set.

Consider these two stylized datasets:

$$\text{dataset1}:\begin{pmatrix} \mathit{id} & \mathit{var}1 & \mathit{var}2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \end{pmatrix}$$

$$\text{dataset3}:\begin{pmatrix} \mathit{id} & \mathit{var}22 & \mathit{var}44 & \mathit{var}46 \\ 112 & \vdots & \vdots & \vdots \\ 216 & \vdots & \vdots & \vdots \\ 449 & \vdots & \vdots & \vdots \end{pmatrix}$$

We may `merge` these datasets on the common *merge key:* in this case, the `id` variable.

$$\text{combined} : \begin{pmatrix} id & var1 & var2 & var22 & var44 & var46 \\ 112 & \vdots & \vdots & \vdots & \vdots & \vdots \\ 216 & \vdots & \vdots & \vdots & \vdots & \vdots \\ 449 & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

The rule for `merge`, then, is that if datasets are to be combined on one or more *merge keys*, they each must have one or more variables with a common name and datatype (string vs. numeric). In the example above, each dataset must have a variable named `id`. That variable can be numeric or string, but that characteristic of the merge key variables must match across the datasets to be merged. Of course, we need not have exactly the same observations in each dataset: if `dataset3` contained observations with additional `id` values, those observations would be merged with missing values for `var1` and `var2`.

This is the simplest kind of merge: the *one-to-one merge*. Stata supports several other types of merges. But the key concept should be clear: the `merge` command combines datasets “horizontally”, adding variables’ values to existing observations.

The `merge` command can also do a “many-to-one"’ or “one-to-many” merge. For instance, you might have a dataset named `hospitals` and a dataset named `discharges`, both of which contain a hospital ID variable `hospid`. If you had the `hospitals` dataset in memory, you could `merge 1:m hospid using discharges` to match each hospital with its prior patients. If you had the `discharges` dataset in memory, you could `merge m:1 hospid using hospitals` to add the hospital characteristics to each discharge record. This is a very useful technique to combine aggregate data with disaggregate data without dealing with the details.

Although “many-to-one"’ or “one-to-many” merges are commonplace and very useful, you should rarely want to do a “many-to-many” (`m:m`) merge, which will yield seemingly random results.

The long-form dataset is very useful if you want to add aggregate-level information to individual records. For instance, we may have panel data for a number of companies for several years. We may want to attach various macro indicators (interest rate, GDP growth rate, etc.) that vary by year but not by company. We would place those macro variables into a dataset, indexed by year, and sort it by year.

We could then `use` the firm-level panel dataset and sort it by `year`. A `merge` command can then add the appropriate macro variables to each instance of `year`. This use of `merge` is known as a *one-to-many* match merge, where the `year` variable is the *merge key*.

Note that the merge key may contain several variables: we might have information specific to industry and year that should be merged onto each firm's observations.

Writing external data

If you want to transfer data to another package, Stat/Transfer is very useful. But if you just want to create an ASCII file from Stata, the `outfile` command may be used. It takes a *varlist*, and the `if` or `in` clauses may be used to control the observations to be exported. Applying `sort` prior to outfile will control the order of observations in the external file. You may specify that the data are to be written in comma-separated format.

The `outsheet` command can write a comma-delimited or tab-delimited ASCII file, optionally placing the variable names in the first row. Such a file can be easily read by a spreadsheet program such as Excel. Note that `outsheet` does *not* write spreadsheet files.

For customized output, the `file` command can write out information (including scalars, matrices and macros, text strings, etc.) in any ASCII or binary format of your choosing.

The `export excel` command may be used to create an Excel spreadsheet from the contents of memory. You may specify the variables and observations to be exported, and can actually modify an existing Excel worksheet or create a new worksheet.

A very useful capability is provided by the `postfile` and `post` commands, which permit a Stata data set to be created in the course of a program. For instance, you may be simulating the distribution of a statistic, fitting a model over separate samples, or bootstrapping standard errors. Within the looping structure, you may `post` certain numeric values to the `postfile`. This will create a separate Stata binary data set, which may then be opened in a later Stata run and analysed. Note, however, that only numeric expressions may be written to the `postfile`, and the parens () given in the documentation, surrounding each *exp*, are required.

Reconfiguring data

Data are often provided in a different orientation than that required for statistical analysis. The most common example of this occurs with panel, or longitudinal, data, in which each observation conceptually has both cross-section (*i*) and time-series (*t*) subscripts. Often one will want to work with a "pure" cross-section or "pure" time-series. If the microdata themselves are the objects of analysis, this can be handled with sorting and a loop structure. If you have data for N firms for T periods per firm, and want to fit the same model to each firm, one could use the `statsby` command, or if more complex processing of each model's results was required, a `foreach` block could be used. If analysis of a cross-section was desired, a `bysort` would do the job.

But what if you want to use average values for each time period, averaged over firms? The resulting dataset of T observations can be easily created by the `collapse` command, which permits you to generate a new data set comprised of summary statistics of specified variables. More than one summary statistic can be generated per input variable, so that both the number of firms per period and the average return on assets could be generated. `collapse` can produce counts, means, medians, percentiles, extrema, and standard deviations.

Different models applied to longitudinal data require different orientations of those data. For instance, seemingly unrelated regressions (`sureg`) require the data to have T observations (“wide”), with separate variables for each cross–sectional unit. Fixed–effects or random-effects regression models `xtreg`, on the other hand, require that the data be stacked or “vec”’d in the “long” format. It is usually much easier to generate transformations of the data in stacked format, where a single variable is involved.

The `reshape` command allows you to transfer the data from the former (“wide”) format to the latter (“long”) format or vice versa. It is a complicated command, because of the many variations on this process one might encounter, but it is very powerful.

As an example, a dataset from the World Bank, provided as a spreadsheet, has rows labelled by both country (`ccode`) and variable (`vcode`), and columns labelled by years. Two applications of `reshape` were needed to transfer the data to the desired `long` format, where the observations have both country and year subscripts, and the columns are variables:

```
reshape long d, i(ccode vcode) j(year)
reshape wide d, i(ccode year) j(vcode) string
```

The resulting data set is in the appropriate format for `xtreg` modelling. If it were to be used in `sureg`–type models, a further `reshape wide` could be applied to transform it into that format.

See Stata Tip 45, Baum and Cox, *Stata Journal* 7:2, 2007.

Returned results

Stata commands are either *r-class* commands like `summarize`, that return results, or *e-class* commands, that return estimates. You may examine the set of results from a r-class command with the command `return list`. For an e-class command, use `ereturn list`. An e–class command will return `e()` scalars, macros and matrices: for instance, after `regress`, the local macro `e(N)` will contain the number of observations, `e(r2)` the R^2 value, `e(depvar)` will contain the name of the dependent variable, and so on.

Commands may also return matrices. For instance, `regress` (like all estimation commands) will return the matrix `e(b)`, a row vector of point estimates, and the matrix `e(V)`, the estimated variance–covariance matrix of the estimated parameters.

Use `display` to examine the contents of a scalar or local macro. For the latter, you must use the backtick and apostrophe to indicate that you want to access the contents of the macro: contrast `display r(mean)` with ``display "The mean is ` mu′ "``. The contents of matrices may be displayed with the `matrix list` command.

Since items are accessible in local macros, it is very easy to write a program that makes use of results in directing program flow. Local macros can be created by the `local` statement, and used as counters (e.g. in `foreach`).

For more information, see my separate slideshow *Why should you become a Stata programmer?*

Some useful Stata commands

help : online help on a specific command
findit : online references on a keyword or topic
ssc : access routines from the SSC Archive
log : log output to an external file
tsset : define the time indicator for timeseries or panel data
compress : economize on space used by variables
pwd : print the working directory
cd : change the working directory
clear : clear memory
quietly : do not show the results of a command
update query : see if Stata is up to date
adoupdate : see if user-written commands are up to date
exit : exit the program (,clear if dataset is not saved)

Data manipulation commands

generate : create a new variable
replace : modify an existing variable
rename : rename variable
renvars : rename a set of variables
sort : change the sort order of the dataset
drop : drop certain variables and/or observations
keep : keep only certain variables and/or observations
append : combine datasets by stacking
merge : merge datasets (one-to-one or match merge)
encode : generate numeric variable from categorical variable
recode : recode categorical variable
destring : convert string variables to numeric
foreach : loop over elements of a list, performing a block of code
forvalues : loop over a numlist, performing a block of code
local : define or modify a local macro (scalar variable)

describe : describe a data set or current contents of memory
use : load a Stata data set
save : write the contents of memory to a Stata data set
insheet : load a text file in tab- or comma-delimited format
infile : load a text file in space-delimited format or as defined in a dictionary
outfile : write a text file in space- or comma-delimited format
outsheet : write a text file in tab- or comma-delimited format
contract : make a dataset of frequencies
collapse : make a dataset of summary statistics
tab : abbreviation for tabulate: 1- and 2-way tables
table : tables of summary statistics

Statistical commands

summarize : descriptive statistics
correlate : correlation matrices
ttest : perform 1-, 2-sample and paired t-tests
anova : 1-, 2-, n-way analysis of variance
regress : least squares regression
predict : generate fitted values, residuals, etc.
test : test linear hypotheses on parameters
lincom : linear combinations of parameters
cnsreg : regression with linear constraints
testnl : test nonlinear hypothesis on parameters
margins : marginal effects (elasticities, etc.)
ivregress : instrumental variables regression
prais : regression with AR(1) errors
sureg : seemingly unrelated regressions
reg3 : three-stage least squares
qreg : quantile regression
sem : structural equation modeling

Limited dependent variable estimation commands

logit, logistic : logit model, logistic regression
probit : binomial probit model
tobit : one- and two-limit Tobit model
cnsreg : Censored normal regression (generalized Tobit)
ologit, oprobit : ordered logit and probit models
mlogit : multinomial logit model
poisson : Poisson regression
heckman : selection model

Time series estimation commands

arima : Box–Jenkins models, regressions with ARMA errors
arfima : Box–Jenkins models with long memory errors
arch : models of autoregressive conditional heteroskedasticity
dfgls : unit root tests
corrgram : correlogram estimation
var : vector autoregressions (basic and structural)
irf : impulse response functions, variance decompositions
vec : vector error–correction models (cointegration)
sspace : state-space models
dfactor : dynamic factor models
ucm : unobserved-components models

rolling: prefix permitting rolling or recursive estimation over subsets

Panel data estimation commands

xtreg,fe : fixed effects estimator
xtreg,re : random effects estimator
xtgls : panel-data models using generalized least squares
xtivreg : instrumental variables panel data estimator
xtlogit : panel-data logit models
xtprobit : panel-data probit models
xtpois : panel-data Poisson regression
xtgee : panel-data models using generalized estimating equations
xtmixed : linear mixed (multi-level) models
xtabond : Arellano-Bond dynamic panel data estimator

Nonlinear estimation commands

The `nl` command may be used to estimate a nonlinear model, while `ml` supports maximum likelihood estimation with a user-specified likelihood function. See my separate slideshow on *Maximum Likelihood Estimation and Nonlinear Least Squares in Stata*.

Mata now contains a full-featured set of optimization commands as `optimize( )`. These commands are now the preferred method to implement optimization in Stata.

Graphics commands:

`twoway` produces a variety of graphs, depending on options listed

`histogram rep78` histogram of this categorical variable

`twoway scatter price mpg` a Y vs X scatterplot

`twoway line price mpg` a Y vs X line plot

`tsline GDP` a Y vs time time-series plot

`twoway area price mpg` an Y vs X area plot

`twoway rline price mpg` a Y vs X range plot (hi-lo) with lines

The command `twoway` may be omitted in most cases.

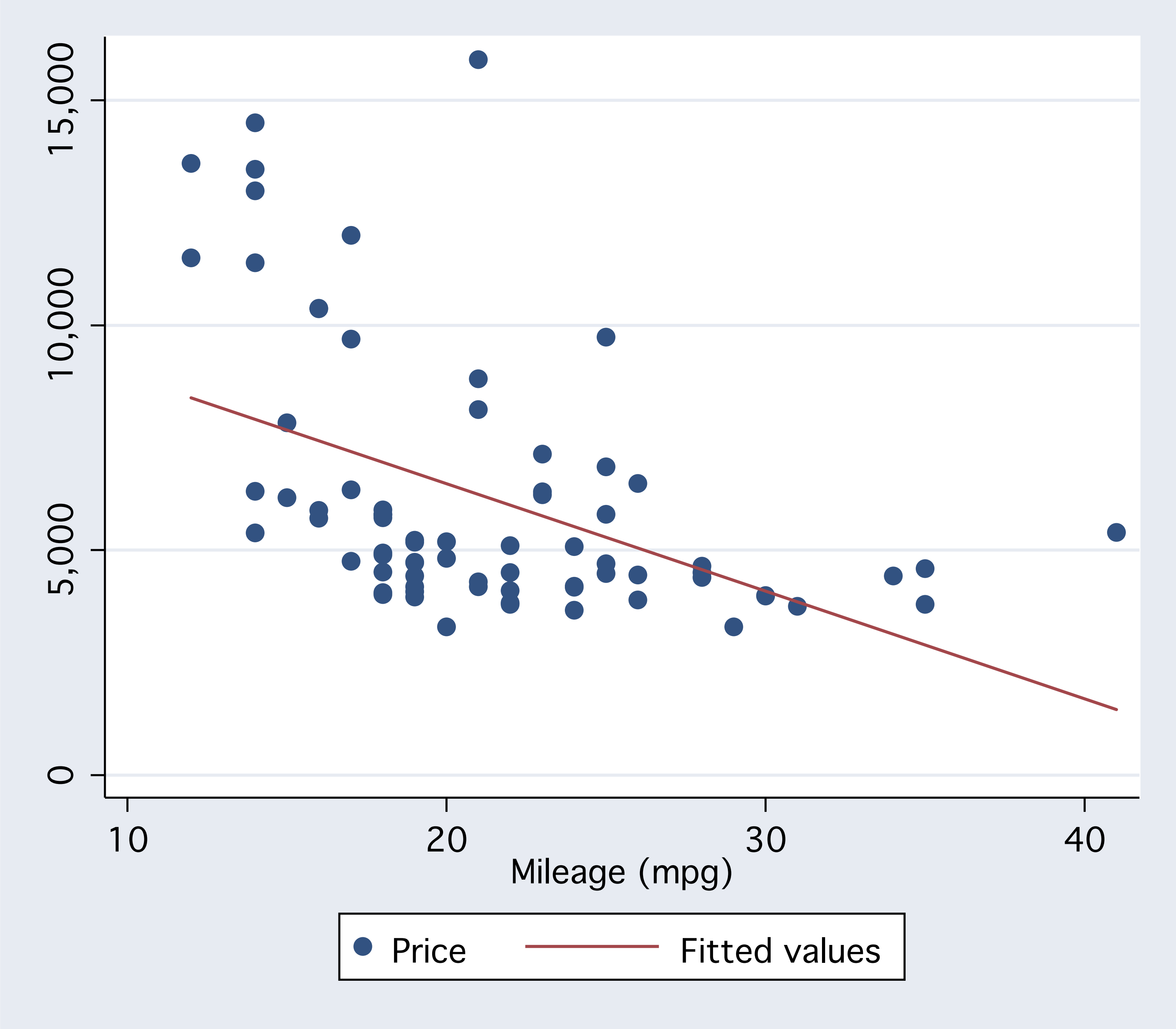
The flexibility of Stata graphics allows any of these plot types (including many more that are available) to be easily combined on the same graph. For instance, using the auto.dta dataset,

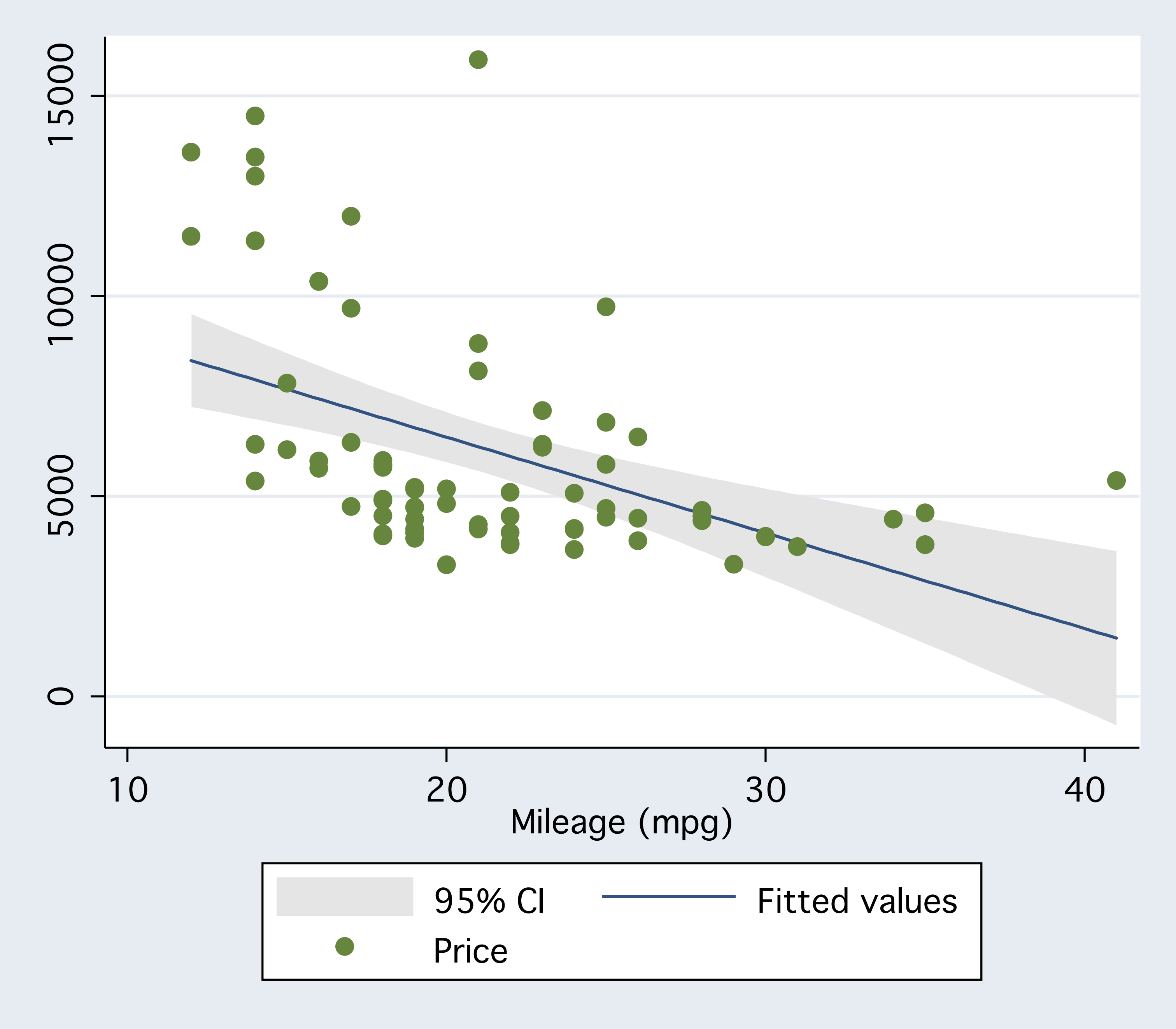
```
twoway (scatter price mpg) (lfit price mpg)
```

will generate a scatterplot, overlaid with the linear regression fit, and

```
twoway (lfitci price mpg) (scatter price mpg)
```

will do the same with the confidence interval displayed.


15,000
10,000
5,000
0
10
20
30
40
Mileage (mpg)
● Price
Fitted values

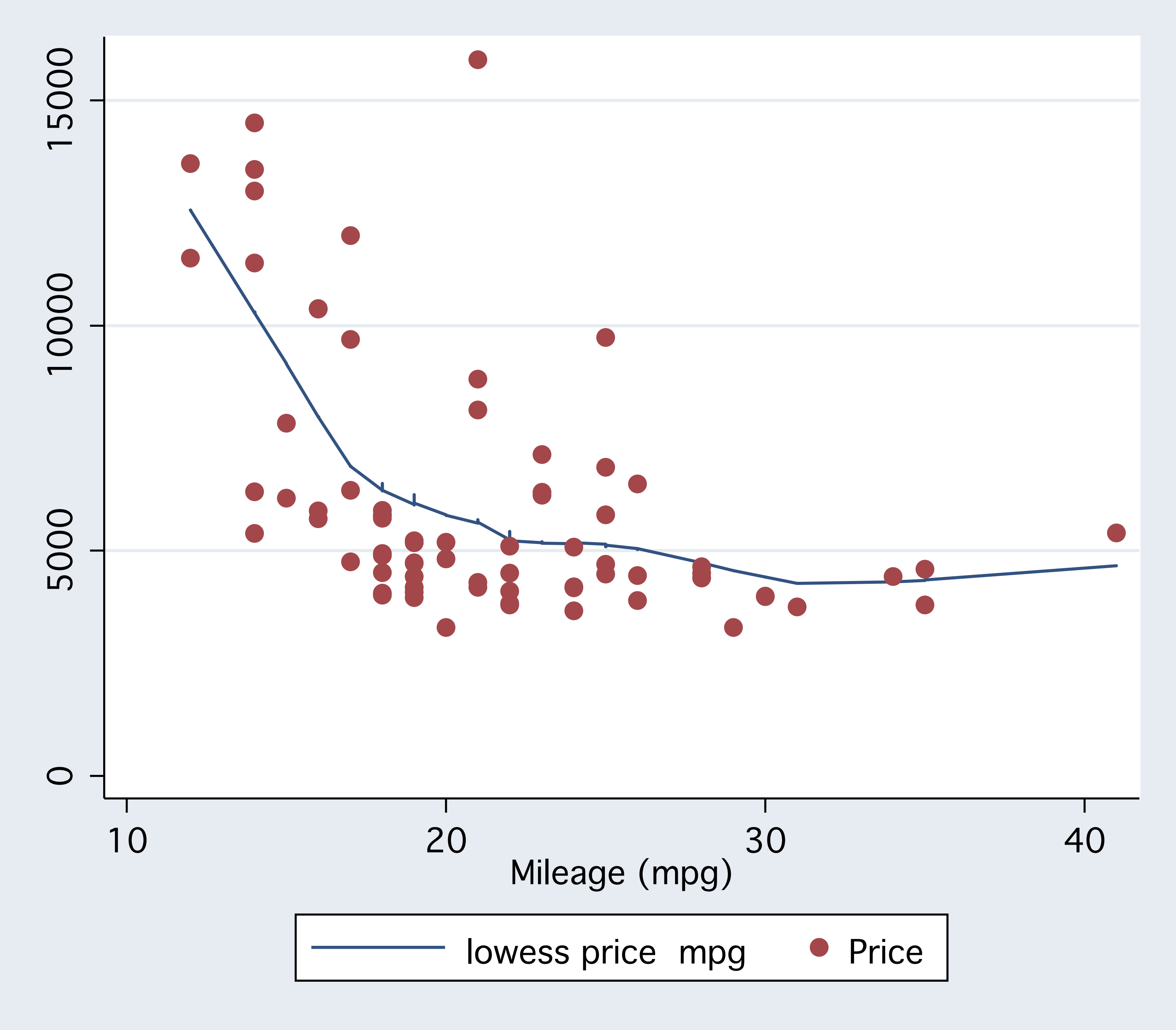

15000
10000
5000
0
10
20
30
40
Mileage (mpg)
95% CI
Fitted values
Price

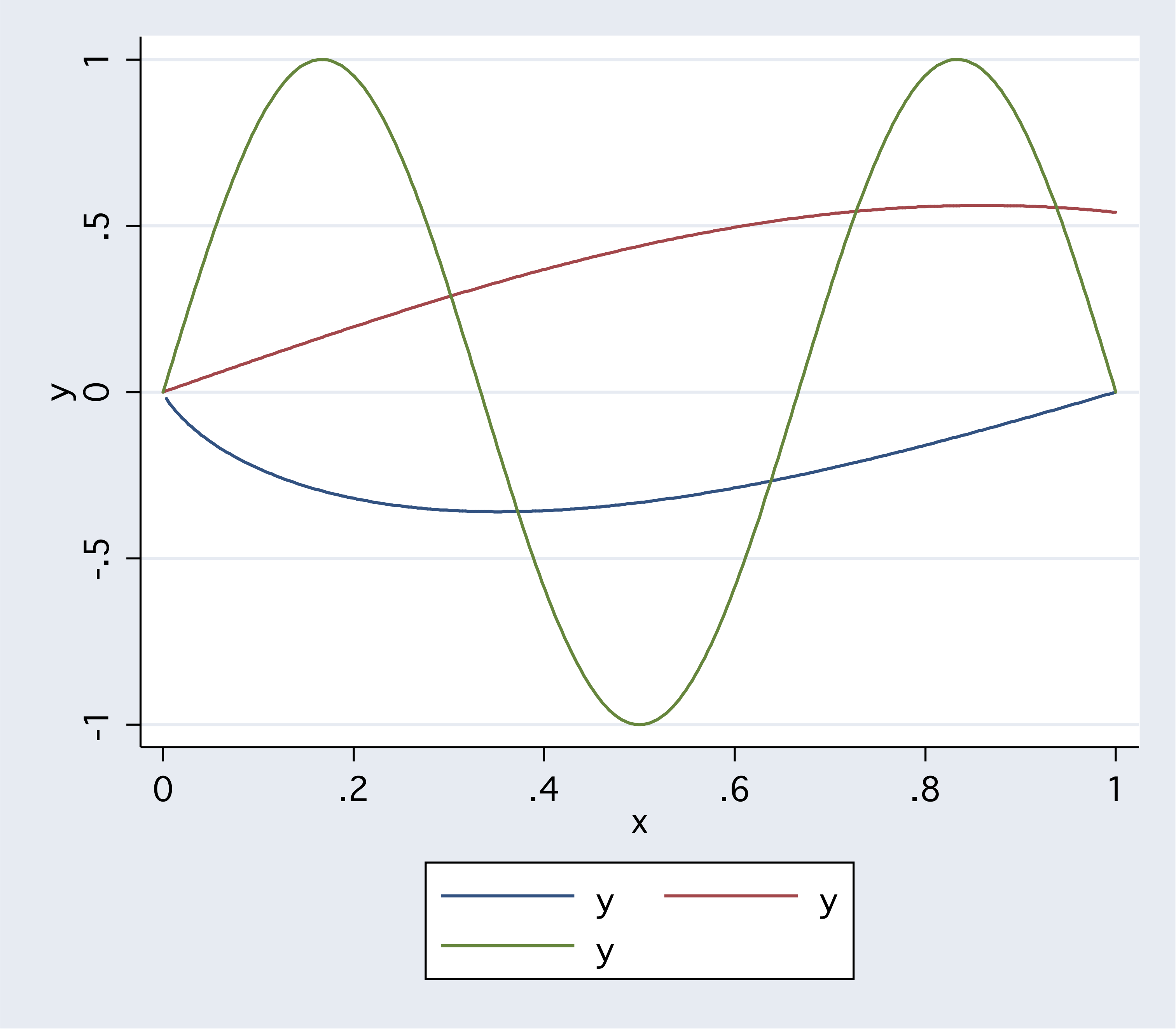
A nonparametric fit of a bivariate relationship can be readily overlaid on a graph via

```
twoway (lowess price mpg) (scatter price mpg)
```

Twoway graphs may also represent mathematical functions, without explicit data:

```
twoway (function y=log(x)*sin(x)) (function y=x*cos(x))
```

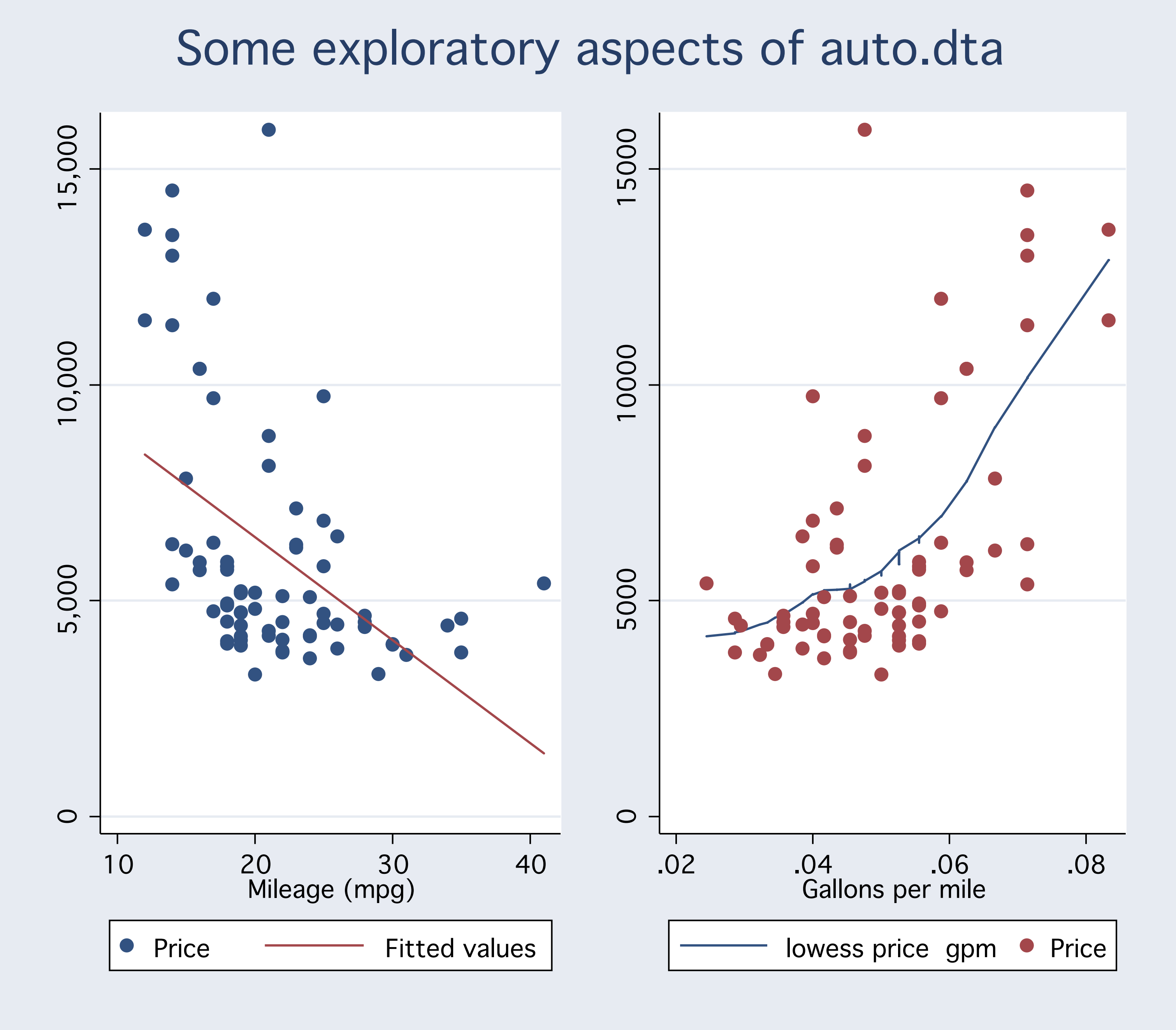

15000
10000
5000
0
10
20
30
40
Mileage (mpg)
lowess price mpg
Price


1
.5
0
-.5
-1
y
0
.2
.4
.6
.8
1
x
y
y
y

Graphs may also be readily combined into a single graphic for presentation. For instance,

```
twoway (scatter price mpg) (lfit price mpg), name(auto1)
gen gpm = 1/mpg
label var gpm "Gallons per mile"
twoway (lowess price gpm) (scatter price gpm),
name(auto2)
graph combine auto1 auto2, saving(myauto, replace) ///
ti("Some exploratory aspects of auto.dta")
```

where the “///” is a continuation of the line.


Some exploratory aspects of auto.dta
0
5,000
10,000
15,000
10
20
30
40
Mileage (mpg)
● Price
Fitted values
0
5000
10000
15000
.02
.04
.06
.08
Gallons per mile
lowess price gpm
● Price

Instructional data sets

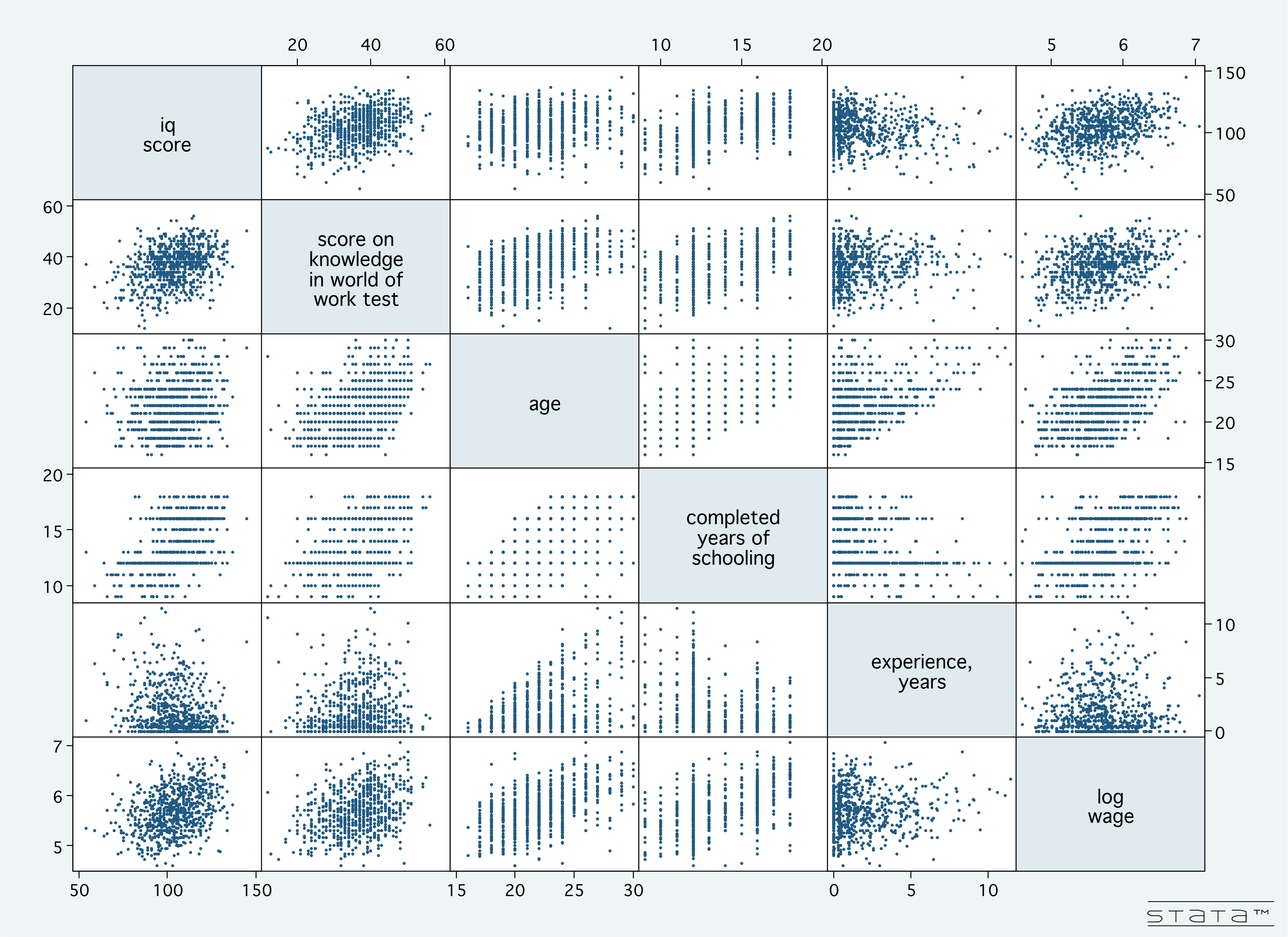
A list of over 100 datasets suitable for instructional use is available on the economics web pages as

```
http://fmwww.bc.edu/ec-p/data/ecfindata.html#teach
```

Sample Stata do-files
Consider the data Zvi Griliches used in his 1976 article on the wages of young men (*Journal of Political Economy*, 84, S69-S85). These are cross-sectional data on 758 individuals collected over several survey years.

```
do http://fmwww.bc.edu/ec-p/software/stata/stataintro1
```

```
* StataIntro: cross-section example
log using intro1, replace
use http://fmwww.bc.edu/ec-p/data/hayashi/griliches76
describe
summarize
label define ur 0 rural 1 urban
label values smsa ur
tab smsa
tab mrt smsa, chi2
ttest med,by(smsa)
anova lw mrt smsa
anova lw mrt smsa mrt*smsa
anova,regress
regress lw tenure kww smsa
predict lweps,resid
scatter lweps kww
bysort year: regress lw tenure kww smsa
graph matrix iq kww age s expr lw, msize(tiny)
gen medrural = med*(smsa==0)
gen medurban = med*(smsa==1)
regress lw tenure kww medurban medrural
test medurban=medrural
log close
```

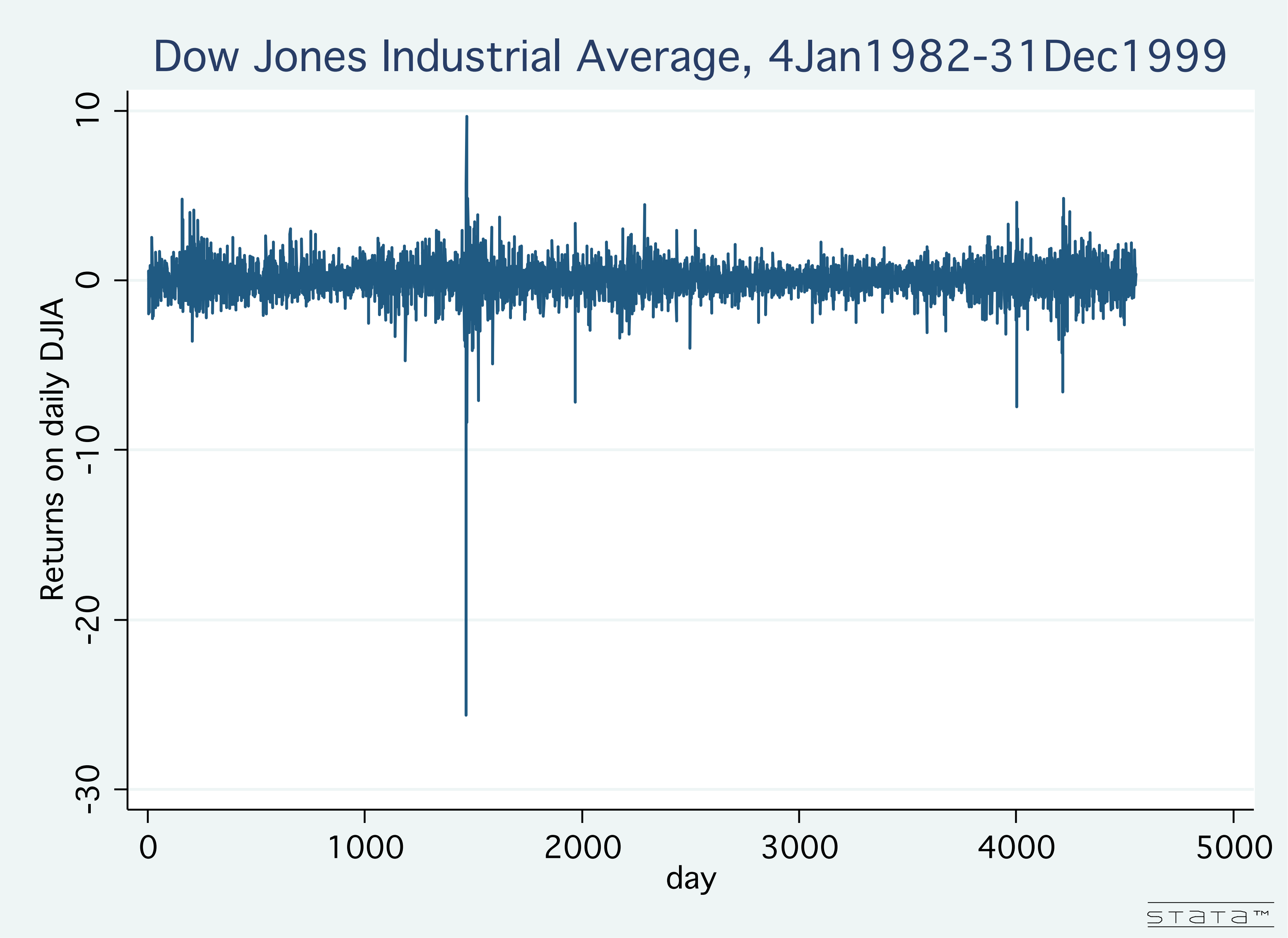

iq score
score on knowledge in world of work test
age
completed years of schooling
experience, years
log wage
20
40
60
10
15
20
5
6
7
150
100
50
60
40
20
30
25
20
15
20
15
10
10
5
0
7
6
5
50
100
150
15
20
25
30
0
5
10
stata™

The following example reads some daily Dow-Jones Averages data, graphs daily returns, then performs Dickey-Fuller tests for unit roots on the DJIA, its log, and its returns (log price relatives), and on their first differences. AR(3) models are then estimated on the series, and the Box–Pierce portmanteau test is then performed on the residuals.

In this example, we make use of "local macros" (with values `` `v' ``), which enable us to perform the same operations on several named variables without having to write out the commands for each variable. This facility may be used with `varlists` of any length, and makes it very easy to generate parallel analyses, produce graphs, etc. for an arbitrary set of variables or time periods.

```
do http://fmwww.bc.edu/ec-p/software/stata/stataintro2
```

```
* StataIntro: time-series example
log using intro2,replace
use http://fmwww.bc.edu/ec-p/data/micro/ddjia.dta
desc
summ
tsset
tsline ret
foreach v of varlist djia ldjia ret {
      dfgls  `v´, maxlag(12)
      dfgls D.`v´, maxlag(12)
      regress `v´  L(1/3).`v´, robust
      predict eps_`v´,resid
      wntestq eps_`v´
      }
log close
```


Dow Jones Industrial Average, 4Jan1982-31Dec1999
Returns on daily DJIA
10
0
-10
-20
-30
0
1000
2000
3000
4000
5000
day
stata™

Examples of Stata programming

Let us form a “rolling forecast” of volatility from a moving-window regression (we had not learned that Baum’s `rollreg` command or Stata’s `rolling:` prefix could do this job for us). Assume that we have 120 time-series observations which have been `tsset`:

```
gen volfc=.
local win 12
forv i=13/120 {
     local first = `i´-`win´+4
     quietly regress y L(1/4).y in `first´/`i´
     quietly replace volfc = e(rmse) in `i´/`i´
}
```

This program will generate the series `volfc` as the RMS error of an AR(4) model fit to a window of 12 observations for the `y` series.

The use of local macros and the appropriate loop constructs make it possible to write a Stata program that is fairly general, and requires little modification to be reused on different series, or with different parameters. This makes your work with Stata very productive, since much of the code is reusable and adaptable to similar tasks. Let us consider how this approach might be pursued in the context of the volatility forecast example.

For more information, see my separate slideshow *Why should you become a Stata programmer?* and my 2009 book *An Introduction to Stata Programming*, available in O'Neill Library.

Writing an ado-file

We show here a complete Stata program, `volfc`, which is stored in the file `volfc.ado` on the `adopath`. Since this is a personally-authored program, it should be placed in the `personal` subdirectory of the `ado` directory (not the Stata directory's `ado` subdirectory!) For more information, see `adopath`.

This program makes use of Stata's syntax parsing capabilities to allow this user-written command to emulate all Stata commands' syntax. It does not make use of many of the features that might be useful in such a command: handling `if` and `in` clauses, providing more specific error messages for inappropriate option values, and so on.

The program generalizes the do-file shown above by allowing the moving–window volatility estimate to be generated from a specified variable, and placed in a new variable specified in the `vol()` option. The window width (option `win()`) and AR length (option `AR()`) take on default values 12 and 4, but may be overridden by the user. The program automatically calculates the first and last observations to be used in the loop from the data and specified options. It could readily be generalized to use a different volatility measure from the rolling regression (e.g. mean absolute error).

To be complete, we should provide a help file for `volfc` in the file `volfc.sthlp`. The help file would specify the syntax of the command, explain its purpose, define each of the options, and provide any references to other Stata commands that might be useful.

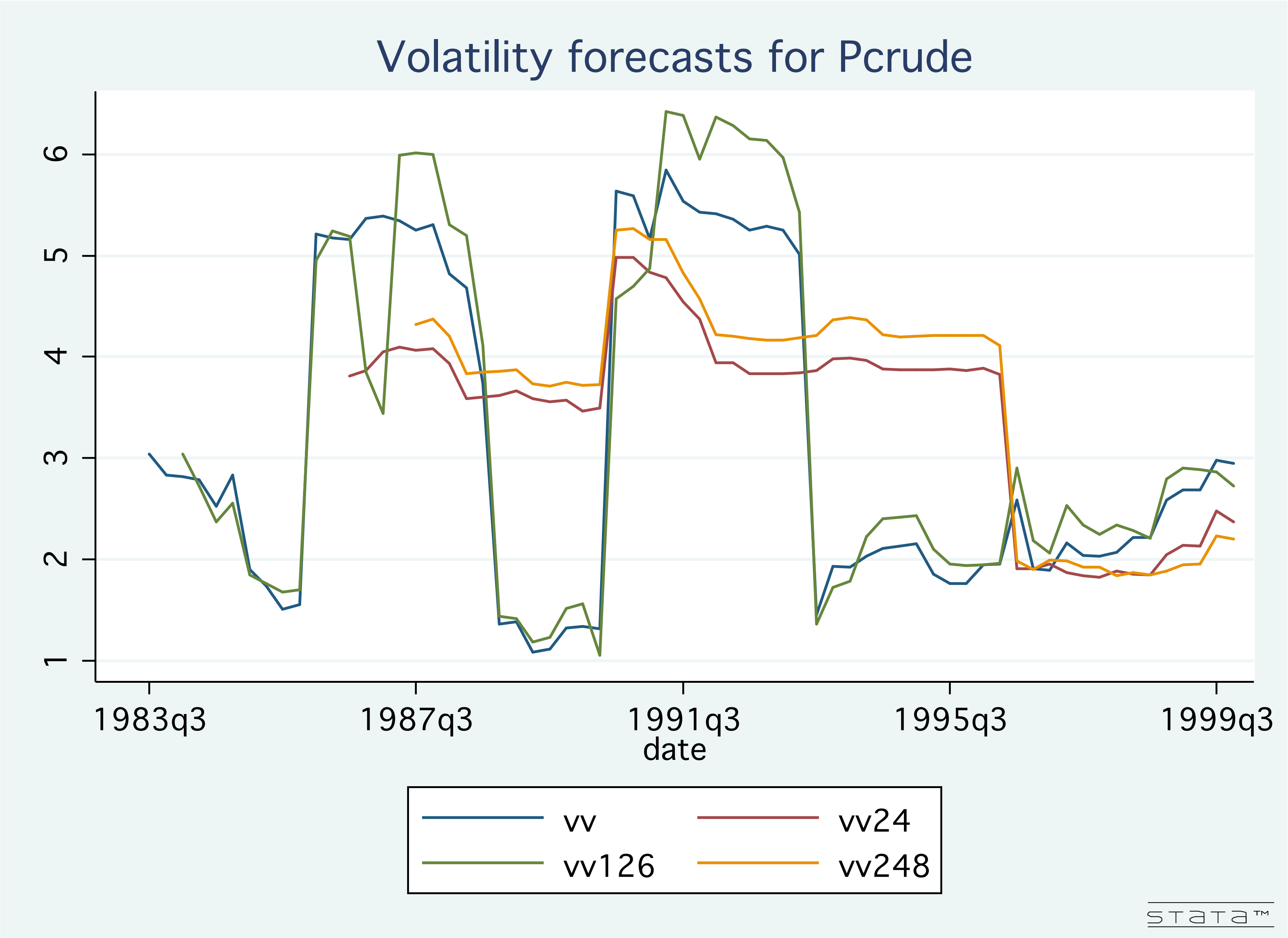
```
program define volfc, rclass
version 10.0
syntax varname(numeric) ,Vol(string) [Win(integer 12) AR(integer 4)]
quietly tsset
if `win´ < `ar´ {
   di "You must have a longer window than AR length!"
   error 198
}
quietly gen `vol´=.
local start = `win´+`ar´
quietly summ `varlist´, meanonly
local last = r(N)
dis _n "`vol´: volatility forecast for `varlist´ with window=`win´, AR(`ar´)"
forv i=`start´/`last´ {
     local first = `i´-`win´+1
     quietly regress `varlist´ L(1/`ar´).`varlist´ ///
                in `first´/`i´
     quietly replace `vol´ = e(rmse) in `i´/`i´
}
exit
end
```

This program defines the volfc command, which will appear like any other Stata command on your machine. It may be executed as

```
use http://fmwww.bc.edu/ec-p/data/macro/bdh, clear
volfc pcrude, vol(vv)
volfc pcrude, vol(vv24) win(24)
volfc pcrude, vol(vv126) ar(6)
volfc pcrude, vol(vv248) win(24) ar(8)
```

The volatility series might then be graphed (presuming a time variable date which is the variable that has been tsset) with

```
 tsline vv vv24 vv126 vv248 if tin(1983q1,), ti(Volatility forecasts for Pcrude)
```


Volatility forecasts for Pcrude
1
2
3
4
5
6
1983q3
1987q3
1991q3
1995q3
1999q3
date
vv
vv24
vv126
vv248
stata™

This illustrates the relative simplicity of developing a quite general tool in Stata's programming language. Although you may use Stata without ever authoring an "ado-file", much of the productivity enhancement that a Stata user may enjoy is likely to be tied to this sort of development. Many research tasks are quite repetitive in some context, and developing a general-purpose tool to implement that repetition is likely to be a very good investment in terms of time and effort.

Many of the modules available from the SSC Archive were first conceived by individuals looking to ease the burden of their own work. Stata's unique extensibility makes it trivial to incorporate user-written additions—including those which you author—into your copy of Stata, and to share it with collaborators or the Stata user community if desired.

Details of program construction

As should be evident from this programming example, the `program define` command is used to declare a program. The program name must match the name of the ado-file in which it is stored. Most user-written programs are r-class. This program could be modified to return its parameters to the calling program with the return statement:

```
return local vol `vol´
return local win `win´
return local ar  `ar´
return local first `start´
return local last `last´
```

With these statements added to the end of the routine, the local macros are defined, and their values stored.

The second element to be noted is the `syntax` statement, which defines the allowable syntax for a user-written command. One may specify that the command allows a single variable, with `varname`; a set of variables, with `varlist`, optionally specifying how many are allowed. For instance, a statistical technique that operates on a pair of variables could specify that exactly two existing variables are to be provided. Likewise, one may specify that a new variable (or set of variables) are the `newvarlist` of the command, and syntax will check that they are indeed new variables.

Although not illustrated above, the syntax command will often specify that `if` and `in` clauses are optional elements. Optional elements of syntax (such as the options `Win` and `AR` above) are placed in brackets ([ ]).

This programming example illustrates a “required option”—the vol option, which must be used on this command to specify the output of the command. The other two options are indeed optional, and take on default values if they are not specified. The argument of the vol option is meant to be a new variable name; that will be trapped when the generate statement attempts to create the variable if it is already in use, or is not a valid variable name.

Most user-written programs could be improved by adding code to trap errors in users’ input. If the program is primarily for your own use, you may eschew extensive development of error trapping: for instance, checking the options for sensibility (although one test is applied here to prevent nonsensical results).

Local macros are exactly that: objects with local scope, defined within the program in which they are used, disappearing when that program terminates. This is generally the desired outcome, preventing a clutter of objects from being retained when a program calls numerous others in the course of execution. At times, though, it is necessary to have objects that can be passed from one subprogram to another. The `return` logic above would not really serve, since although it passes local macros from a program to its caller, they would then have to be passed as arguments to a second program.

To deal with the need for persistent objects, Stata contains *global macros*. These objects, once defined, live for the duration of your Stata session, and may be read or written within any Stata program. They are defined with the `global` command, rather than `local`, and referred to as `$macroname`. Global macros should only be used where they are required.

Example of programming for panel data

We now present an example of a Stata program that operates on panel, or longitudinal data. When you use panel data, you must use the panel data form of `tsset` in which both a unit variable and a time variable are specified.

Assume that you have a panel data set, properly identified as such, containing several time series for each unit in the panel: for instance, investment or population measures for several countries. We would like to generate a new series containing the deviations from a constant growth path (exponential trend) or, alternatively, the constant growth values themselves (the predicted values from the exponential trend line).

This program, pangrodev, performs this task for each unit of a panel, automatically identifying the observations belonging to each unit, taking the logarithm of the specified variable, running the appropriate regression and prediction commands, and assembling the results in the specified new variable.

The program makes use of Stata's tempname and tempvar commands to create non-scalar objects (in this case the matrix VV and variables lvar and pvar which, like local macros, will exist only for the duration of the ado-file). These temporary facilities, like the associated tempfile which allows temporary files to be specified, help reduce clutter and guarantee that objects' names will not conflict with other items in the user's namespace.

```
*! pangrodev 1.1.0  CFBaum 21Jan2006
*  generate deviations from constant growth in panel
* 1.1.0: promote to v9, use levelsof
program define pangrodev, rclass
version 10.0
syntax varname, Gen(string) [xb]
local togens "deviations from constant growth"
if "`xb´" != "" {
local togens "predicted growth"
}
qui tsset
local ivar = r(panelvar)
local timevar = r(timevar)
tempname VV
tempvar lvar pvar
qui gen double `lvar´ = log(`varlist´)
* get list of units
qui levelsof `ivar´, local(vals)
local nvals: word count `vals´
qui gen double `gen´=.
local xc 0
local tbar 0
local rsqr 0
```

(continues...)

```
foreach v of local vals {
summ `lvar´ if `ivar´==`v´,meanonly
if r(N)>2 {
qui regress `lvar´ `timevar´ if `ivar´==`v´
capt drop `pvar´
qui predict double `pvar´ if e(sample),xb
qui replace `gen´ = exp(`pvar´) if e(sample)
if "`xb´" ==""  {
qui replace `gen´ = `varlist´-`gen´ if e(sample)
}
local xc = `xc´ + 1
local tbar = `tbar´ + e(N)
local rsqr = `rsqr´ + e(r2)
}
}
local tbar = int(100*`tbar´ / `xc´)/100.0
local rsqr = int(1000*`rsqr´ / `xc´)/1000.0
di in gr _n "`gen´ : `togens´ for `xc´ of `nvals´ units"
di in gr "tbar = `tbar´    rsq-bar = `rsqr´"
exit
end
```

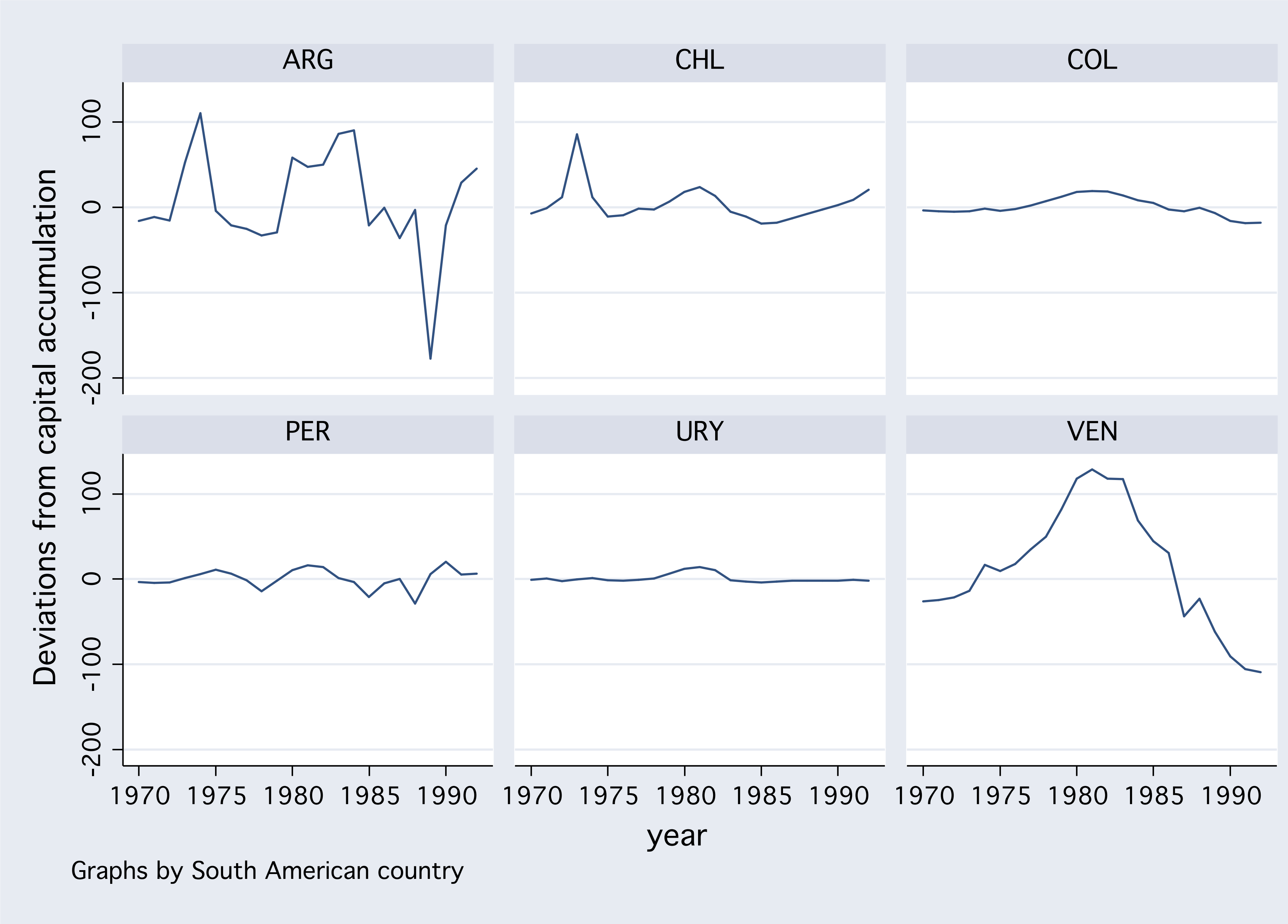
This program defines the `pangrodev` command, which will appear like any other Stata command on your machine. It may be executed as

```
. use http://fmwww.bc.edu/ec-p/data/macro/cap797wa
(World Bank Database for Sectoral Investment, 1948-1992)
. pangrodev TotSECap, g(totcapdev)

totcapdev : deviations from constant growth for
                  57 of 63 units
tbar = 25.94    rsq-bar = .673

. pangrodev TotSECap, g(totcaphat) xb
(output omitted)
```

Selected series computed by `pangrodev` can now be graphed by the `tsline` command, which accepts a `by(`*varlist*`)` option:

```
replace totcapdev = totcapdev/10^9
keep if (ccode==ÄRG¨ | ccode==C̈HL¨ | ccode==C̈OL¨ | ///
ccode==P̈ER¨ | ccode==ÜRY¨ | ccode==V̈EN)̈
label var totcapdev "Deviations from capital accum"
label var ccode "South American country"
tsline totcapdev if year>1969, by(ccode)
```

will demonstrate how many countries followed the same pattern of below-trend growth of the capital stock (curtailed investment) during the 1980s.


ARG
CHL
COL
PER
URY
VEN
Deviations from capital accumulation
100
0
-100
-200
1970
1975
1980
1985
1990
year
Graphs by South American country

Concluding remarks

Whether or not you use Stata's programming facilities to write your own ado-files, a "reading knowledge" of the programming language is very useful in case you want to adapt an existing Stata command (official or user-contributed) in a do-file you are writing.

Since the code for all Stata commands that are implemented as ado-files (as the command `which...` will show) are available on your hard disk, Stata itself is a fertile source of programming techniques that may be adapted to solve any programming problem.

For a thorough treatment of the subject, see my book *An Introduction to Stata Programming* (2009) in O'Neill Library.