Introduction to Stata

Christopher F Baum

Boston College and DIW Berlin

University of York, December 2009

What is Stata? Stata is a full-featured statistical programming language for Windows, Macintosh, Unix and Linux. It can be considered a "stat package," like SAS, SPSS, RATS, or eViews. The number of variables is limited to 2,047 in standard Stata/IC, but can be much larger in Stata/SE or Stata/MP. The number of observations is limited only by memory.

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

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

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

And—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.

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, marginal effects, robust estimation of standard errors, instrumental variables and two-stage least squares, seemingly unrelated regressions, vector autoregressions and error correction models, and so on. 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, VAR, VEC), model simulation and bootstrapping, maximum likelihood estimation, nonlinear least squares and GMM estimation. Families of commands provide the leading techniques utilized in each of several categories:

- "`xt`" commands for cross-section/time-series or panel (longitudinal) data
- "`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. One omission: Stata does not presently support 3-D graphics.

Availability and Support

Stata is available in several versions, all of which provide the full set of features and commands: there are no special add-ons or 'toolboxes'. Beginning with Stata 11, each coy 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 11. 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.

The standard version, Stata/IC, is limited only in being able to handle datasets with no more than 2,047 variables. It can work with any number of observations, depending on your computer's memory. There is a student version, Small Stata, but I do not recommend it for any serious econometric work.

Stata/SE relaxes the 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.

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.

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. The kernel is updated relatively infrequently—once a month at most—but the ado-files may be modified every ten days or so. This enables Stata's developers to distribute bug fixes, enhancements to existing commands, and even entirely new commands during the lifetime of a given release. 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.

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.

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.

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

Stata also provides an elaborate hypertext-based help browser, providing complete access to commands' descriptions and examples of syntax, with links to the appropriate pages of the PDF manuals. Each of these components appears in a separate window on the screen in the GUI version of Stata.

Extensibility

Another clear advantage of the command-line driven environment is its interaction with 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.

If Stata's developers tomorrow wrote a 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 straight ASCII text. 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 `http://ideas.repec.org`.

The SJ is a subscription publication (freely available from IDEAS beyond a three-year moving window), but the ado- and sthlp-files may be freely downloaded from Stata's web site. The Stata command `help` 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 Stata.

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 1,200) have been placed in the Boston College Statistical Software Components (SSC) Archive in **RePEc**, 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 "ivreset," you could use `ssc install ivreset` to install it. Anything in the archive can be accessed via Stata's `ssc` command: thus `ssc describe ivreset` 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.

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.

As the current working directory (cwd) is on the `adopath`, if I create an ado-file **hello.ado**:

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

Stata will now respond to the command `hello`. It's that easy.

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. Personal copies of Stat/Transfer version 10 (which handles Stata versions 6, 7, 8, 9, 10 and 11 datafiles) are available at a discounted rate from the StataCorp website.

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.

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.

Following the examples in the *Getting Started with Stata...* manual, we will make use of `auto.dta`, a dataset of 74 automobiles' characteristics.

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 ⋀ 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`.

Accessing data over the Web

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.

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”).

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.

Data validation

A key element of most research projects is the validation of the data to ensure that it does not contain errors. A number of Stata commands are useful in this context. The `describe` and `summarize` commands may be used to perform 'sanity checks' on the data to ensure that variables have appropriate data types, and that numeric variables have sensible means, minima and maxima.

Data acquired from other sources may code missing values as some numeric value such as -9 or -999, and may use more than one of those codes in a variable to indicate different conditions. The `mvdecode` command may be used to transform those special values into Stata's missing value code so that they are not inadvertently included in statistical analysis.

The assert command

You may use the `assert` command in a do-file to check for invalid values in one or more variables. For instance, if you `assert foo > 0`, the program will stop if any non-positive values of that variable are encountered. If you `assert pct >= 0 & pct <= 100`, values of that variable outside the 0–100 range will be trapped. If you `assert Nbirth = 0 if gender == "M"`, the program will trap the condition of male pregnancies in the sample. Any number of `assert` commands may be included in a do-file to ensure that invalid values of variables, or invalid combinations of variables, are detected.

The duplicates command

The `duplicates` command is also very useful for the detection of invalid characteristics. If a particular variable should take on distinct values, `duplicates list foo` will identify any failures. The command can also be used with a set of variables to identify invalid combinations: e.g., `duplicates list company year` would flag duplicate firm-years in a panel of firm-level data. `duplicates` can also be instructed to drop duplicate observations.

A number of sound data management principles can improve the quality of analysis conducted with Stata.

- Bring the data into Stata for manipulation as early in the process as possible.
- Construct a well-documented do-file to validate the data, ensuring that variables that should be complete are complete, that unique identifiers are such, and that only sensible values are present in every variable.
- The validated and, if necessary, corrected file should not be modified in later analysis. Subsequent data transformations or merges should create new files rather than overwriting the original contents of the validated file.

Strict adherence to these principles, although time-consuming, will ultimately *save* a good deal of your time.

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.

Factor variables

A new feature in Stata version 11 is the *factor variable*. Stata has only one kind of numeric variable (although it supports several different data types, which define the amount of storage needed and possible range of values). However, if a variable is *categorical*, taking on non-negative integer values, it may be used as a factor variable with the `i.` prefix.

The use of factor variables not only avoids explicit generation of indicator (dummy) variables for each level of the categorical variable, but it means that the needed indicator variables are generated 'on the fly', as needed. Thus, to include the variable `rep78`, a categorical variable in `auto.dta` which takes on values 1–5, we need only refer to `i.rep78` in an estimation command.

This in itself merely mimics a preexisting feature of Stata: the `xi:` prefix. But factor variables are much more powerful, in that they can be used to define interactions, both with other factor variables and with continuous variables. Traditionally, you would define interactions by creating new variables representing the product of two indicators, or the product of an indicator with a continuous variable.

There is a great advantage in using factor variables rather than creating new interaction variables. if you define interactions with the factor variable syntax, Stata can then interpret the expression in postestimation commands such as `margins`. For instance, you can say `i.race#i.sex`, or `i.sex#c.bmi`, or `c.bmi#c.bmi`, where `c.` denotes a continuous variable, and `#` specifies an interaction.

With interactions between indicator and continuous variables specified in this syntax, we can evaluate the total effect of a change without further programming. For instance,

```
regress healthscore i.sex#c.bmi c.bmi#c.bmi
margins, dydx(bmi) at(sex = (0 1))
```

which will perform the calculation of $\partial healthscore/\partial bmi$ for each level of categorical variable `sex`, taking into account the squared term in `bmi`. We will discuss `margins` more fully in later talks in this series.

Mata: Matrix programming language

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. We will discuss Mata at greater length in later talks in this series.

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.

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 `margins` 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.

Loading external data: insheet

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
```

Loading external data: infile

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.

Loading external data: infix

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.

Loading external data: Stat/Transfer

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 Stata/IC data sets is encountered.

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

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.

You might have a dataset on the demographic characteristics in 2007 of the largest municipalities in England, `cityEng`. If you were given a second dataset containing the same variables for the largest municipalities in Scotland in 2007, `cityScot`, you might want to combine those datasets with `append`. With the `cityEng` dataset in memory, you would `append using cityScot`, which would add those records as additional observations. You could then save the combined file under a different name. `append` can be used to combine multiple datasets, so if you had the additional files `cityWales` and `cityNIre`, you could list those filenames in the `using` clause as well.

Prior to using `append`, it is a good idea to create an identifier variable in each dataset that takes on a constant value: e.g., `gen region = 1` in the English dataset, `gen region = 2` in the Scottish dataset, etc.

Combining datasets

You may be aware that Stata can only work with one dataset at a time. How, then, do you combine datasets in Stata? First of all, it is important to understand that at least one of the datasets to be combined must already have been saved in Stata format. Second, you should realize that each of Stata's commands for combining datasets provides a certain functionality, which should not confused with that of other commands.

For instance, consider the `append` command with two stylized datasets:

$$\text{dataset1}: \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \end{pmatrix}$$

$$\text{dataset2}: \begin{pmatrix} id & var1 & var2 \\ 126 & \vdots & \vdots \\ 309 & \vdots & \vdots \\ 421 & \vdots & \vdots \\ 604 & \vdots & \vdots \end{pmatrix}$$

These two datasets contain the same variables, as they must for `append` to sensibly combine them. If `dataset2` contained `idcode, Var1, Var2` the two datasets could not sensibly be appended without renaming the variables.[1] Appending these two datasets with common variable names creates a single dataset containing all of the observations:

[1] Recall that in Stata `var1` and `Var1` are two separate variables.

$$\text{combined}: \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \\ 126 & \vdots & \vdots \\ 309 & \vdots & \vdots \\ 421 & \vdots & \vdots \\ 604 & \vdots & \vdots \end{pmatrix}$$

The rule for `append`, then, is that if datasets are to be combined, they should share the same variable names and datatypes (string vs. numeric). In the above example, if `var1` in `dataset1` was a `float` while that variable in `dataset2` was a `string` variable, they could not be appended.

It is permissible to append two datasets with differing variable names in the sense that `dataset2` could also contain an additional variable or variables (for example, `var3, var4`). The values of those variables in the observations coming from `dataset1` would then be set to missing.

The merge command

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

$$\text{dataset1}: \begin{pmatrix} id & var1 & var2 \\ 112 & \vdots & \vdots \\ 216 & \vdots & \vdots \\ 449 & \vdots & \vdots \end{pmatrix}$$

$$\text{dataset3}: \begin{pmatrix} id & var22 & var44 & var46 \\ 112 & \vdots & \vdots & \vdots \\ 216 & \vdots & \vdots & \vdots \\ 449 & \vdots & \vdots & \vdots \end{pmatrix}$$

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.

postfile and post

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 datasets

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.

When data have more than one identifier per record, they may be organized in different ways. For instance, it is common to find on-line displays or downloadable spreadsheets of data for individual units—for instance, U.S. states—with the unit's name labeling the row and the year labeling the column. If these data were brought into Stata in this form, they would be in the *wide form*, wide form with the same measurement (population) for different years denoted as separate Stata variables:

```
. list, noobs
```

state	pop1990	pop1995	pop2000
CT	3291967	3324144	3411750
MA	6022639	6141445	6362076
RI	1005995	1017002	1050664

There are a number of Stata commands—such as `egen` row-wise functions—which work effectively on data stored in the wide form. It may also be a useful form of data organization for producing graphs.

Alternatively, we can imagine stacking each year's population figures from this display into one variable, `pop`. In this format, known in Stata as the *long form*, each datum is identified by two variables: the state name and the year to which it pertains.

We use `reshape` to transform the data, indicating that `state` should be the main row identifier (`i`) with `year` as the secondary identifier (`j`):

```
. reshape long pop, i(state) j(year)

. list, noobs sepby(state)
```

state	year	pop
CT	1990	3291967
CT	1995	3324144
CT	2000	3411750
MA	1990	6022639
MA	1995	6141445
MA	2000	6362076
RI	1990	1005995
RI	1995	1017002
RI	2000	1050664

This data structure is required for many of Stata's statistical commands, such as the `xt` suite of panel data commands. The long form is also very useful for data management using `by`-groups and the computation of statistics at the individual level, often implemented with the `collapse` command.

Inevitably, you will acquire data (either raw data or Stata datasets) that are stored in either the wide or the long form and will find that translation to the other format is necessary to carry out your analysis. In statistical packages lacking a data-reshape feature, common practice entails writing the data to one or more external text files and reading it back in.

With the proper use of `reshape`, writing data out and reading them back in is not necessary in Stata. But `reshape` requires, first of all, that the data to be reshaped are labelled in such a way that they can be handled by the mechanical rules that the command applies. In situations beyond the simple application of `reshape`, it may require some experimentation to construct the appropriate command syntax. This is all the more reason for enshrining that code in a do-file as some day you are likely to come upon a similar application for `reshape`.

Repeating commands

One of Stata's great strengths is the ability to perform repetitive tasks without spelling out the details (e.g. the `by` prefix). However, the `by` prefix can only execute a single command; so that while you may run a regression for each country in your sample, you cannot also save the residuals or predicted values for those country-specific regressions.

Stata provides two commands that allow construction of a true block structure or loop: `foreach` and `forvalues`. These commands permit a delimited block of commands to be repeated over elements of a *varlist* or *numlist*. Indeed, the target of `foreach` may be any list of names, and can be a list of new variables to be created. The `forvalues` *numlist* may include an increment, so that it could for instance count from 10 to 100 in steps of 10: (`10(10)100`), or count down from 10 to 1 (`10(-1)1`).

This code fragment loops over a varlist, calculates (but does not display) the descriptives of each variable, and then summarizes the observations of that variable that exceed its mean. Note the use of *‘var’*, in particular the backtick (‘) on the left of the word. This syntax is mandatory when referring to the placeholder.

```
foreach var of varlist pri-rep t* {
        quietly summarize `var'
        summarize `var' if `var' > r(mean)
}
```

Generally a `forvalues` or `foreach` loop is the best way to solve any programming problem that involves repetition. It is usually much faster, in the long run, to figure out how to place a problem in this context. Nested loops may also be defined with these commands.

A loop structure may also be explicitly defined by the `while` command, which is akin to the "do while" construct in other programming languages. A `while` structure often will make use of an `if` command—not to be confused with the `if` clause on other commands—which will create conditional logic. The `if` command may also use an `else` clause to express conditional logic.

For many purposes, it is more efficient (in terms of your time) to employ `foreach` or `forvalues`, since those commands handle the logic of repetition without explicit detail. Programs written with these commands are easier to maintain and modify.

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. In addition, all Stata commands that compute results or estimates generate one or more objects to hold those items, which are scalars (numbers), local macros (strings) or matrices.

Reusing returned results: return list

This behavior of Stata's computational commands allows you to write a do-file that makes use of these quantities. We saw one example of this above. The command `summarize` generates a number of scalars, such as `r(N)`, the number of observations; `r(mean)`, the mean; `r(Var)`, the variance; etc. The available items are shown by `return list` for a "r-class" command. The contents of these scalars may be used in expressions. In the example above, the mean of a variable was used to govern a following `summarize` command:

```
quietly summarize price
summarize price if price > r(mean)
```

In this example, the scalar `r(mean)` may be used directly. But what if you wanted to issue another command that generated results, which would wipe out all of the `r()` returns? Then you use the `local` statement to preserve the item in a macro of your choosing:

```
local mu r(mean)
```

Later in the program, you could use

```
regress mpg weight length if price > `mu'
```

Note the use of the backtick (‘) on the left of the local macro. This syntax is mandatory, as it makes the *dereference* clear: you are referring to the *value* of the local macro `mu` rather than the contents of the variable `mu`.

Reusing returned results: ereturn list

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`).

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.

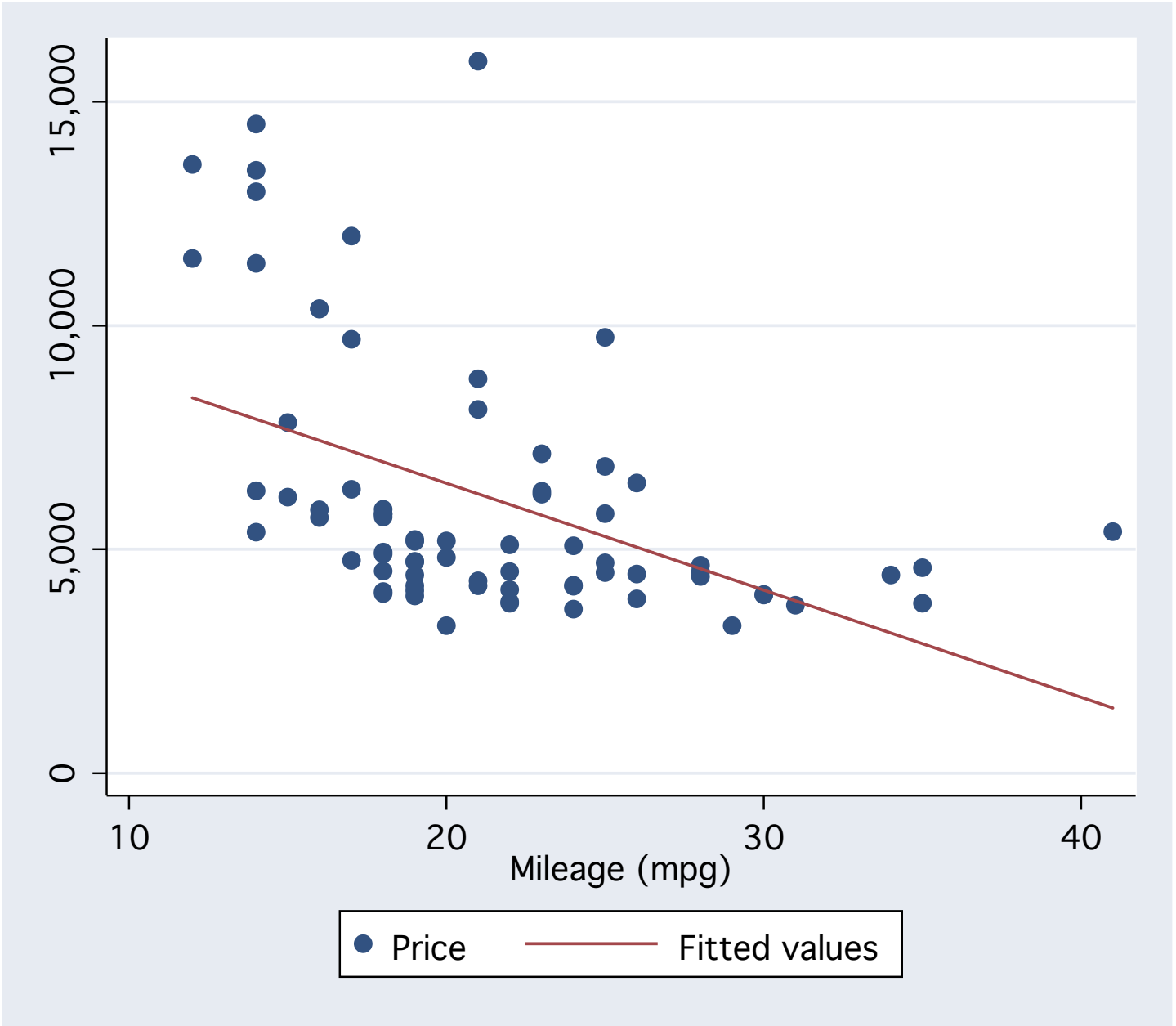
The flexibility of Stata graphics allows any of these plot types (including many more that are available) to be easily combined on the same graph. For instance, using the auto.dta dataset,

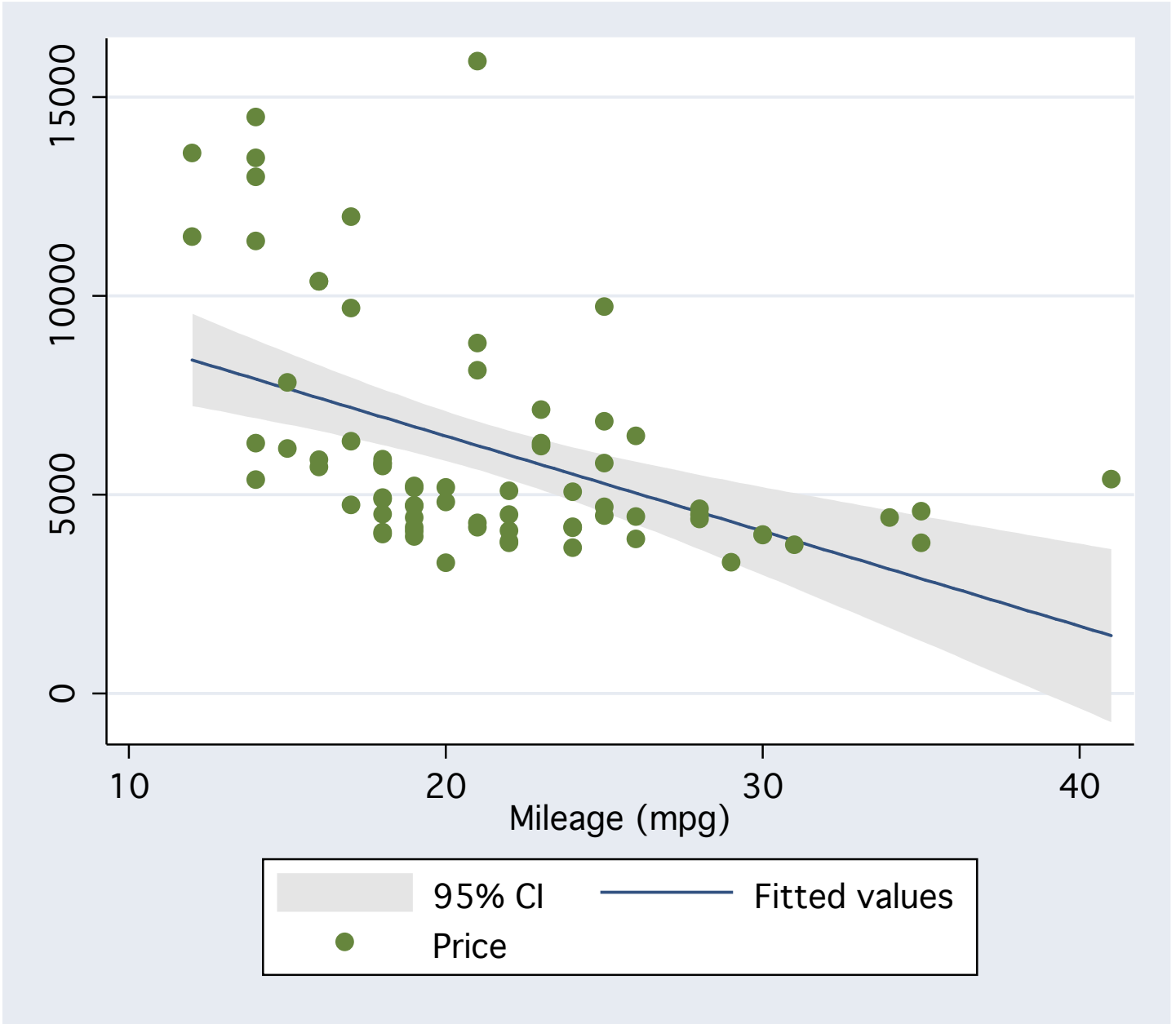
```
twoway (scatter price mpg) (lfit price mpg)
```

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

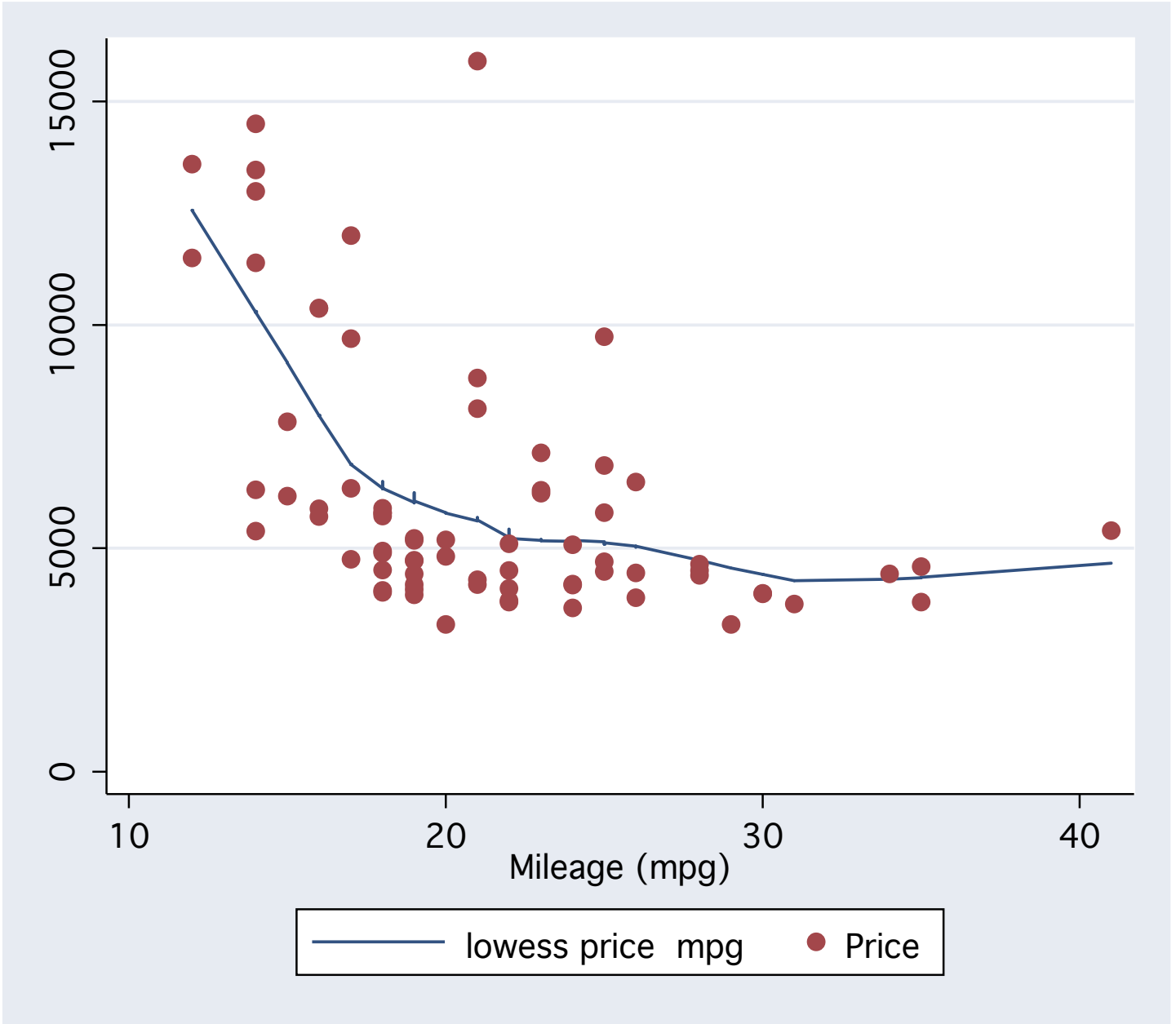

15000
10000
5000
0
10
20
30
40
Mileage (mpg)
95% CI
Fitted values
Price

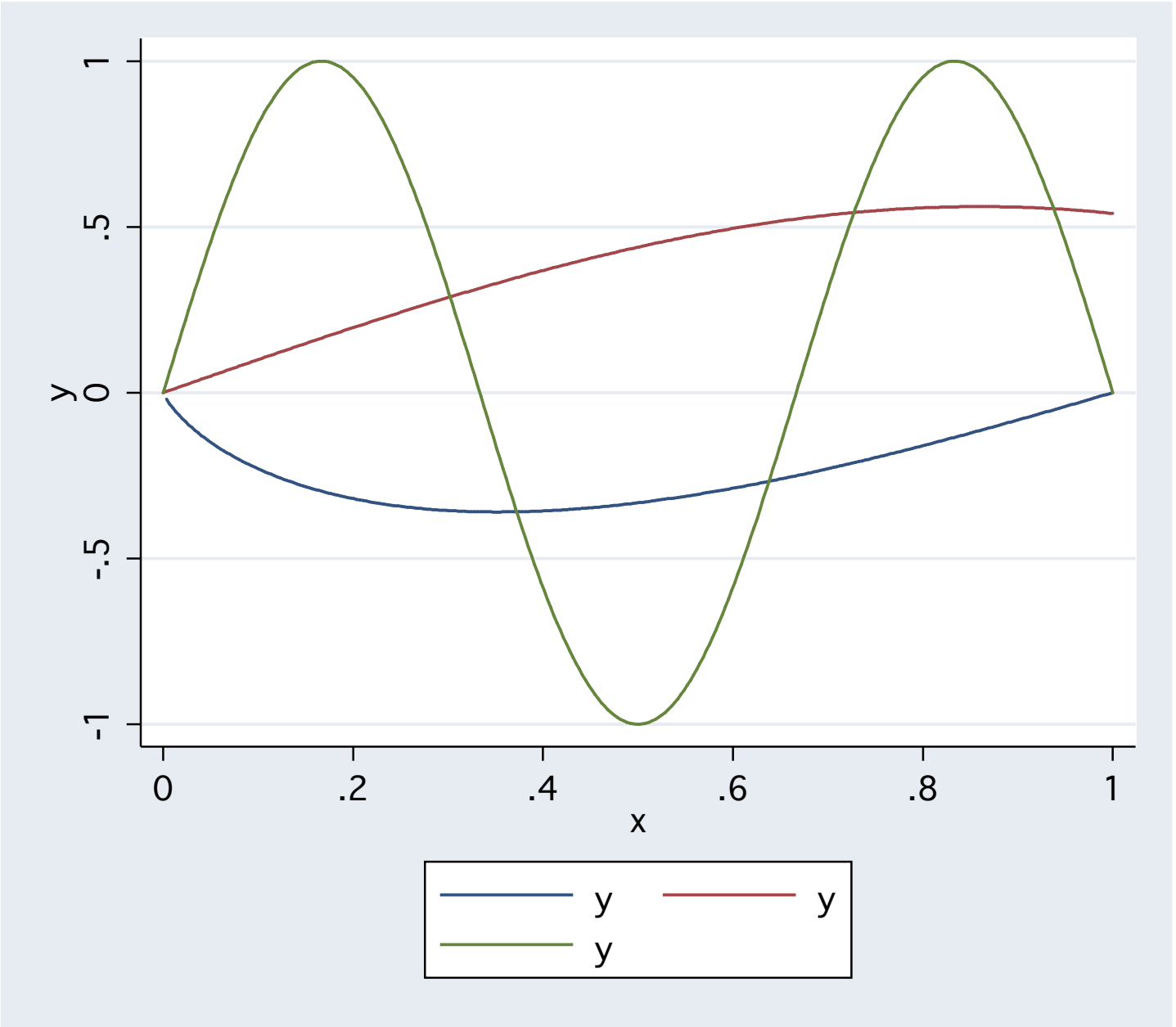
A nonparametric fit of a bivariate relationship can be readily overlaid on a graph via

```
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

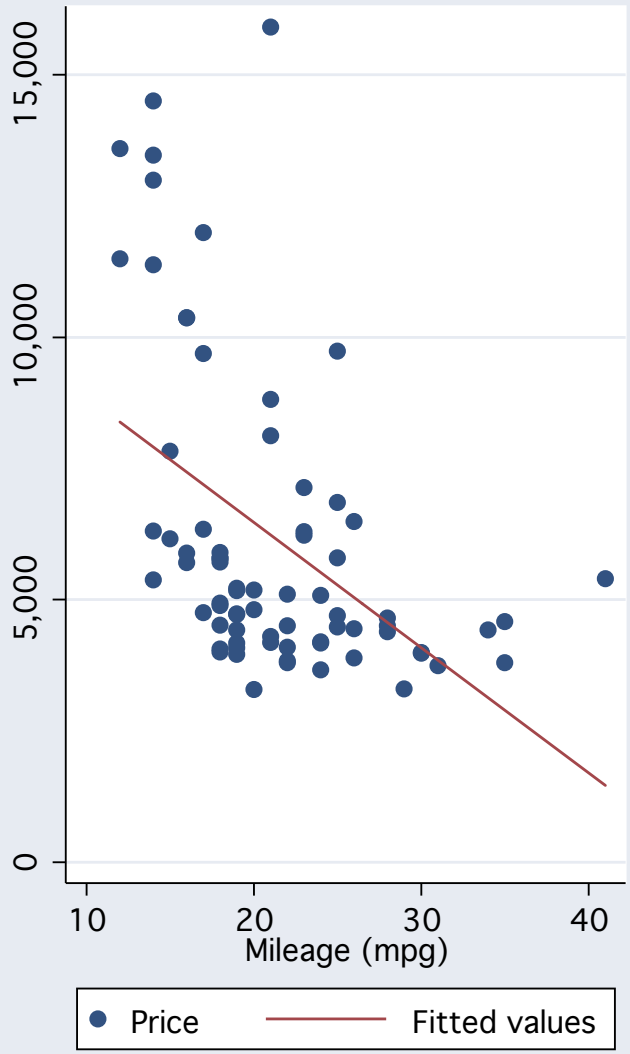

1
.5
0
-.5
-1
y
0
.2
.4
.6
.8
1
x
y
y
y

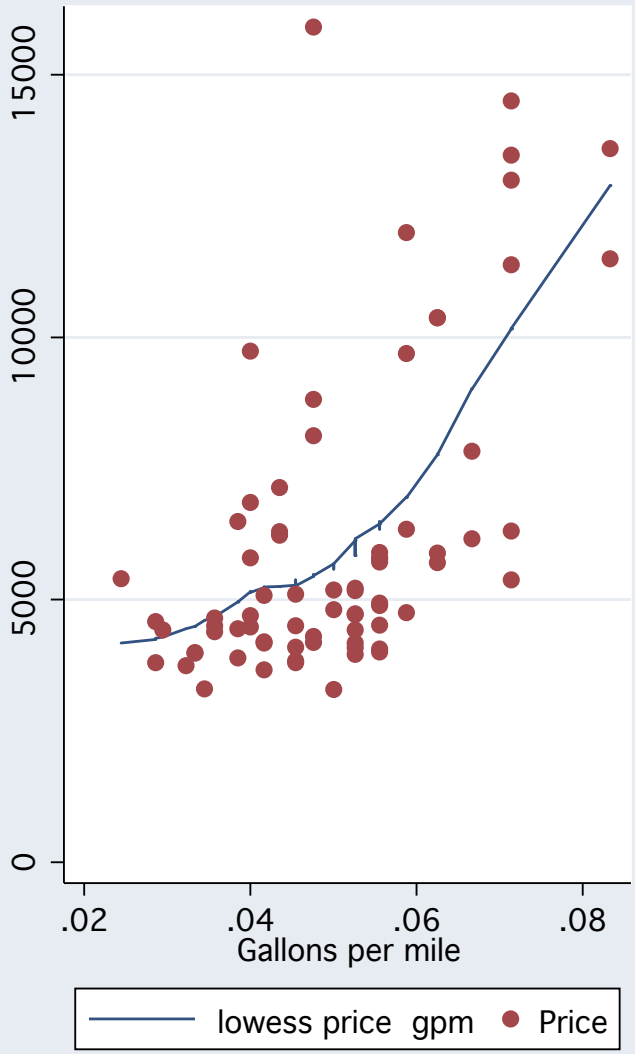
Graphs may also be readily combined into a single graphic for presentation. For instance,

```
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





Report and Graphics Automation

One of Stata's great strengths, relative to using spreadsheet software to produce a research report, is that every aspect of the analysis can be automated without learning any additional 'macro languages'. You may often want to produce a set of tables or a set of figures which are identical in format for different industries, firms, countries, or years.

A very modest investment in Stata's programming capabilities gives you the ability to perform this task: and more importantly, the ability to redo the task when some aspect of the analysis has changed. Likewise, once you have set up the structure to perform this automation, it takes very little effort to clone it to perform a similar task.

As an example, consider the `wagepan` dataset (ITSP p. 181), which contains 4,360 longitudinal observations over eight years, 1980–1987. We want to estimate two wage models for each year, calculate elasticities at the point of means for two key regressors and produce a set of publication-quality tables, one per year. As the dependent variable is already in log terms, we use `margins, dyex(educ hours)`.

We develop a simple do-file that performs these tasks, making use of Ben Jann's `estout` suite of commands (available from the SSC Archive). Our target is a set of LaTeX tables, although we could also produce SMCL, HTML, RTF or tab-delimited output.

```
 . xtset nr year
       panel variable:  nr (strongly balanced)
        time variable:  year, 1980 to 1987
                delta:  1 unit
. replace hours = hours / (50*40)
hours was int now float
(4360 real changes made)
. local reg1 educ hours
. local reg2 educ hours exper expersq
. forvalues y = 1980/1987 {
  2.          eststo clear
  3.          forvalues j = 1/2 {
  4.          qui regress lwage `reg`j'' if year == `y'
  5.          qui margins, dyex(educ hours)
  6.          mat eta = r(b)
  7.          qui eststo, addscalars(etaeduc eta[1,1] etahours eta[1,2])
  8.          }
  9.          qui esttab using lwage_`y'.tex, replace not nomtitles se ///
>         ti("Wage equations for `y'") nodepvars scalar(r2 rmse etaeduc etahour
> s) ///
>         substitute("_cons" "constant" "etaeduc" "$\eta_{\rm educ}$" ///
>         "etahours" "$\eta_{\rm hours}$" "r2" "$ R^2$")
 10. }
```

This approach has the advantage that the tables themselves need not be included in the LaTeX document, so that if we revise the tables (to include a different specification, for instance) we need not copy and paste the tables. To illustrate, we display one of the tables here.

LaTeX output is most convenient as LaTeX is itself a programming language, and Stata code may easily write programs in that language. The example above could be easily adapted to produce a set of tab-delimited files for use in Word or Excel, or alternatively a set of HTML web pages.

TABLE 1. Wage equations for 1984

	(1)	(2)
educ	0.0760***	0.0917***
	(0.0124)	(0.0156)
hours	-0.195*	-0.193*
	(0.0873)	(0.0872)
exper		-0.0811
		(0.0761)
expersq		0.00625
		(0.00459)
constant	1.016***	1.074**
	(0.175)	(0.373)
N	545	545
R^2	0.0714	0.0773
rmse	0.506	0.505
η_{educ}	0.895	1.079
η_{hours}	-0.220	-0.219

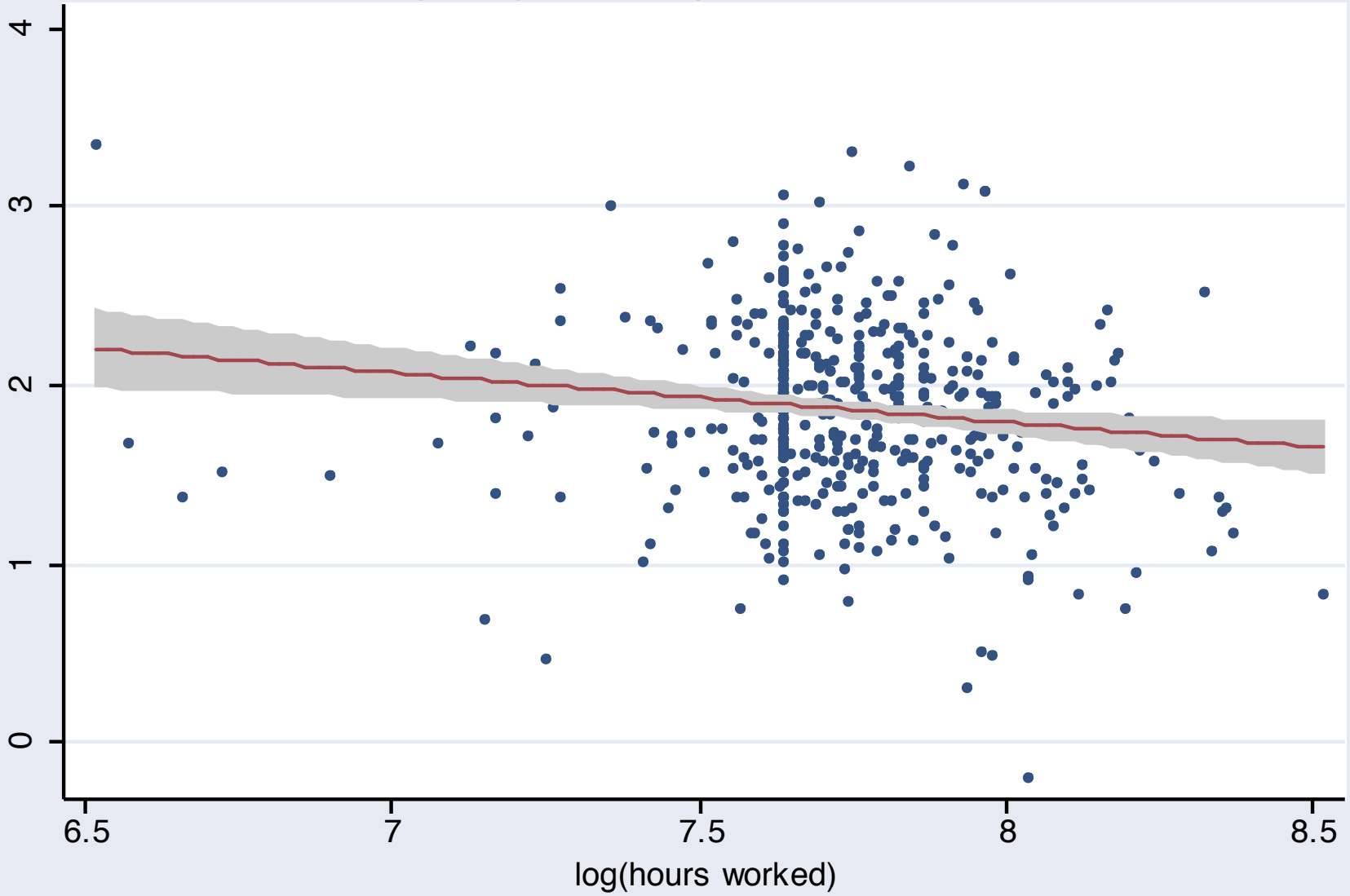
Standard errors in parentheses

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

We might also want to produce a set of graphs. Generation of graphs that meet particular specifications is even more time-consuming in other languages, as a number of manual steps are often necessary to produce the desired graph. But as Stata's graphics are completely programmable, you need only write the code that generates the appropriate graph, and reuse it in a loop. We illustrate with a modified version of our analysis of `wagepan`, using a double-log specification.

```
. forvalues y = 1980/1987 {
  2.          qui regress lwage lhours if year == `y´
  3.          qui twoway (scatter lwage lhours if e(sample), msize(vsmall)) ///
>         (lfitci lwage lhours if e(sample),  legend(off) ///
>         saving(lwage`y´, replace) ti("log(wage) vs log(hours) for `y´"))
  4.          qui graph export lwage_`y´fig.pdf, replace
  5. }
```

In this example, we use `graph export` to produce a PDF of the graph. That feature is only available in Stata for Mac OS X. On a Windows system, you could export the graph as `.eps` for inclusion in LaTeX, or `.wmf` for inclusion in MS Word.


log(wage) vs log(hours) for 1987
4
3
2
1
0
6.5
7
7.5
8
8.5
log(hours worked)