Stata Training

Jamie Fogel
Research Assistant

July 31, 2013

This presentation borrows heavily from Ryan Kessler's Stata training, Summer 2012

Basic Rules for Stata Success

1. All of your work should be in a do-file
2. Use loops and macros to keep your code simple
3. Add comments to your code to keep it organized and make it easy for others (or yourself a year later) to understand
4. Use a log file
5. Set up an organized file structure

All of your work should be in a do-file

- You **ALWAYS** want to have a record of what you are doing; a do-file is that record
- You may be asked to update your results months from now, after new data has been released. With a do-file this is as simple as re-running your code with the updated data.

Boolean

In most programming languages, including Stata, a **Boolean**, or logical, is a data type that takes only two possible values: true or false. A Boolean expression evaluates to 0 if false or 1 if true.

```
di 1==1
> 1
di 1==0
> 0
```

Boolean expressions

In programming "==" tests for equality, while "=" is used for assigning values to variables.

```
set obs 1
gen x = 1
count if x == 1
> 1
gen y == 1
> == invalid name
> r(198);
```

The operators `==`, `~=`, `!=`, `>`, `>=`, `<`, and `<=` are used to test equality or inequality. The operators `&` `|` `~` and `!` are used to indicate "and", "or", and "not".

Boolean arguments

Logical or Boolean arguments, such as the argument to `if` or `while`, may take on any value, not just 0 or 1; 0 is treated as false and any other numeric value as true

```
local a = 35
if `a'    di `a'
> 35
local a = 0
if `a'    di `a'
>
```

Be careful: **Missing evaluates to true!!!** Remember, as far as an `if` or `while` statement is concerned, 0 means false and anything else means true. Stata's missing value indicator, ".", is not 0 and therefore must mean true. This is simple, but it is something you are very likely to encounter.[ii]

[ii] Aside: Stata's missing actually evaluates to infinity. This can be very important if you use `if` statements with missing data

A useful resource

Much of the language on the preceding slides was taken from the below Stata support page. I recommend you visit it if you have any further questions.
`http://www.stata.com/support/faqs/data-management/true-and-false/`

Macros

A Stata macro is what most programming languages call a variable; they can be used to store content, numeric or string, which can then be inserted elsewhere in the code by invoking the macro. Macros can be local or global in scope, with the former existing solely within the program or do-file in which they are defined and the latter remaining available until they are dropped (`macro drop macroname`) or Stata is closed.

Local macros are invoked by placing the macroname between a backtick (‘) and a forward tick (’); global macros are invoked by preceding the macroname with the dollar sign ($). In either case, braces ({}) can be used to clarify meaning and to form nested constructions. For example:

```
* Local macros
local l1 "p"
local l2 "i"
gen ‘l1’‘l2’ = c(pi)
```

```
* Global macros
global pat "T:/census/ACS/"
do \${pat}acs2009use.do
```

We often use macros instead of scalars — a similar construct used to store numbers and strings. But we really shouldn't.

1. Using a scalar will be faster than using a macro, because a macro must be converted from its internal (binary) representation into a printable form.
2. More importantly, these conversions into and out of binary representation can result in a loss of accuracy if the numeric quantity is not an integer.

By storing the result of a computation — for example, a variable's mean or standard deviation — in a scalar Stata need not convert its value and hence the result is held in Stata's full numeric precision.

```
local x = sqrt(2)
scalar x = sqrt(2)
di (   x    == sqrt(2))
> 0
di ( x == sqrt(2))
> 1
```

```
sysuse auto , clear

sum weight , detail
scalar q1 =r(p25)
scalar q3 =r(p75)
di " IQR = " q3 - q1
```

Two things to consider when assigning values to macros:

1. Beware the evaluating equal sign

```
local equal = 2*2
local noequal 2*2
display "`equal'"
> 4
display "`noequal'"
> 2*2
```

2. Compound quotes(`" "')

```
local cities ""Boston MA" "Cambridge MA" "Brookline MA""
display "`cities'"
> Boston not found
> r(111)
display `"`cities'"'
> "Boston MA" "Cambridge MA" "Brookline MA"
```

Macro extended functions — typically of the form `local macroname : ...` — simplify data management tasks (and bypass the 244 character cap on string length, which is being eliminated in Stata 13!).

```
// Assigning to ‘months’ a list of the 12 months
local months ‘c(Months)’
local summer "June July August"

// Remove the summer months
local not_too_hot : list months - summer

// Replace all instances of "January" with "Jan"
local not_too_hot : subinstr local not_too_hot "January" "Jan", all

// Putting list into alphabetical order
local not_too_hot : list sort not_too_hot

// Accessing the length of the macro
local len : length local not_too_hot
display "‘len’"
>64

display "‘not_too_hot’"
> April December February Jan March May November October September
```

The `if` command vs. `if` qualifier

Stata provides two different types of `if` structures — the `if` qualifier and the `if` command or statement.

- `if` qualifier — tells a Stata command to operate on only a subset of the data. The `if` qualifier appears after the command name in the command's syntax and is followed by an expression which it will evaluate. The command will then only operate on observations for which the expression evaluates to "TRUE."
- `if` command — is a programming command and tells Stata to execute certain commands only if the expression following "if" evaluates to "TRUE." The `if` command always appears at the beginning of a line of code.

The `if` qualifier

The `if` qualifier tells a Stata command to operate on only a subset of the data. For example, using the auto data set we could restrict a regression of price on MPG and weight to only foreign cars. This is much more convenient than dropping observations that we want to ignore for now but may use later.

```
sysuse auto, clear
reg price mpg weight
reg price mpg weight if foreign==1
```

Similarly, we might want to drop all observations for which a certain variable is missing.

```
sysuse auto, clear
drop if missing(rep78)
```

The `if` command

The `if` command is a programming command that allows the programmer to specify conditions that must be satisfied in order for certain lines of code to be executed. Whereas the `if` qualifier operates observation-by-observation, the `if` command is only evaluated once. The `if` command is often used within loops or within programs which take arguments. The code to be executed in the `if` command evaluates to "TRUE" may be placed on the same line as the `if` command if it is only a single line; otherwise it must follow the `if` command in braces.

```
forval i=1(1)10{
    if `i'<=5 display `i'
    if `i'>5{
        local j = `i'^2
        di `j'
    }
}
```

Loop Basics

Loops are a programming construct that allow you to repeat the same code for different values of a variable. Stata provides 3[iii] types of loops:

1. `foreach` — Probably the most useful type of loop; used for cycling through lists of values — variable names, numbers, elements in a macro — and repeating commands.
2. `forvalues` — A special case of `foreach`, intended for cycling through certain kinds of numlists.
3. `while` — Rather than repeating commands a specified number of times, as in a `foreach` or `forvalues` loop, a `while` loop repeats commands as long as a specified condition evaluates to true. It is easy to generate an infinite loop using `while` loops if you are not careful.

[iii] For loops exist but are out-of-date and are rarely used

The foreach Loop

```
// Looping over all variables in memory
foreach x of varlist * {
    gen log_‘x’ = log(‘x’)
}
```

```
// Looping over years of PSID
foreach y of numlist 1968/1997 1999(2)2009 {
    gen inc_‘y’ = income if year == ‘y’
}
```

```
// Looping over elements of a macro
local vowels "A E I O U"
foreach v of local vowels {
    di "‘v’"
}
```

The forvalues Loop

```
// Display all integers 1--10
forval i=1(1)10 {
    di `i'
}
```

```
// Alternative syntax for incrementing by 1
forval i=1/10 {
    gen var`i' = `i'
}
```

```
// Counting down from 10, by 2's
forval i=10(-2)0 {
    di `i'
}
```

The `while` Loop

```
// Counting dow from 10, by 2's
local i=10
while `i' >= 0 {
    di `i'
    local `i' = `i'-2
}
```

```
//Read and process text from a text file
capture file close file_to_read_from
file open file_to_read_from using `file', read
file read file_to_read_from line
while r(eof) == 0{
     //do stuff with text which is saved in `line'.
     file read file_to_read_from line
}
file close file_to_read_from
```

What are strings?

In a programming context "strings" are variables composed of text, rather than numeric, characters. In Stata, you will encounter two types of strings.

1. String variables — Some variables will be stored as strings. For example, names or categorical variables like race or religion.
2. Macros — Any macro, local or global, containing text is a string.

While strings are ubiquitous and often necessary, they can be difficult to manipulate and use in data management or statistical analysis. Fortunately, Stata offers a wide variety of string functions.

encode and decode

Sometimes you will encounter a categorical string variable which you would like to use in estimation. Stata cannot use string variables in estimation, however. In this case you will want to convert your string variable to a numeric variable that can be used in estimation using the encode command. encode will create a new numeric variable with integer values corresponding to the original string values and will save the string values as value labels.

```
sysuse auto, clear
encode make, generate(make_numeric)
```

Analogously, decode will take a numeric variable with value labels and generate a string variable where each observation contains the value label for the corresponding numeric variable.

```
sysuse auto, clear
decode foreign, gen(foreign_string)
```

Matching and replacing text

`strmatch` tests whether or not two strings match.

```
local bestpackage "Stata"
di strmatch("`bestpackage'","Excel")
> 0
di strmatch("`bestpackage'","Stata")
> 1
```

Let's replace some text with `subinstr`!

```
local dumb "Excel is great for statistics. I love Excel!"
local smart = subinstr("`dumb'", "Excel", "Stata", .)
di " `smart'"
> Stata is great for statistics. I love Stata!
```

Regular expressions

Regular expressions allow for the matching of complicated patterns of text with minimal effort. For example, regular expressions allow the user to take a list of standardized addresses and split them into street address, city, state, and zipcode.

```
clear

local date  "`c(current_date)'"
if regexm("`date'","^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$") local d=regexs(1)
if regexm("`date'","^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$") local m=regexs(3)
if regexm("`date'","^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$") local y=regexs(5)

macro list

set obs 1
gen date = "`date'"
gen d = regexs(1) if regexm("`date'", "^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$")
gen m = regexs(3) if regexm("`date'", "^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$")
gen y = regexs(5) if regexm("`date'", "^([0-9]+)( *)([a-zA-Z]+)( *)([0-9]+)$")
```

Other string functions

We have only covered a small number of the string functions available in Stata. Others include:

- `length(str)` — returns the number if characters in str
- `trim(str)` — remove all leading and trailing spaces from str
- `lower(str)` — change all uppercase letters to lowercase (`upper()` also exists)
- `substr(str,n1,n2)` — extracts characters n1 through n2 from str

A more thorough (but not exhaustive) list can be found at `http://econweb.tamu.edu/jnighohossian/tools/stata/strings.htm`

file read/write

Stata's `file read` and `file write` commands allow programmers to write ASCII text and binary files. It should NOT be confused with `insheet`, `infile`, and `infix`, which are the usual ways that data are brought into Stata from text files. You will rarely use `file read` and `file write` for any standard tasks like loading or saving data and outputting results; however, if you spend enough time working in Stata you are likely to encounter an unusual situation in which these commands come in handy.

Suppose you want to create a list of every possible value of a variable in a particular data set. In this example, I list every color scheme in Stata's preloaded data set "colorschemes.dta."

```
sysuse colorschemes, clear
//Store all possible values of the variable in the specified macro
levelsof scheme, local(list)

// Open the file to be written to and refer it with the handle ``writefile''
capture file close writefile
file open writefile using                                          ///
    /home/a1jsf01/trainings/stata/do/schemne_list.txt, write replace

foreach m of local list{
    // Write each element of "list", followed by the endline character
    file write writefile `"`m'"' _n
}

file close writefile
```

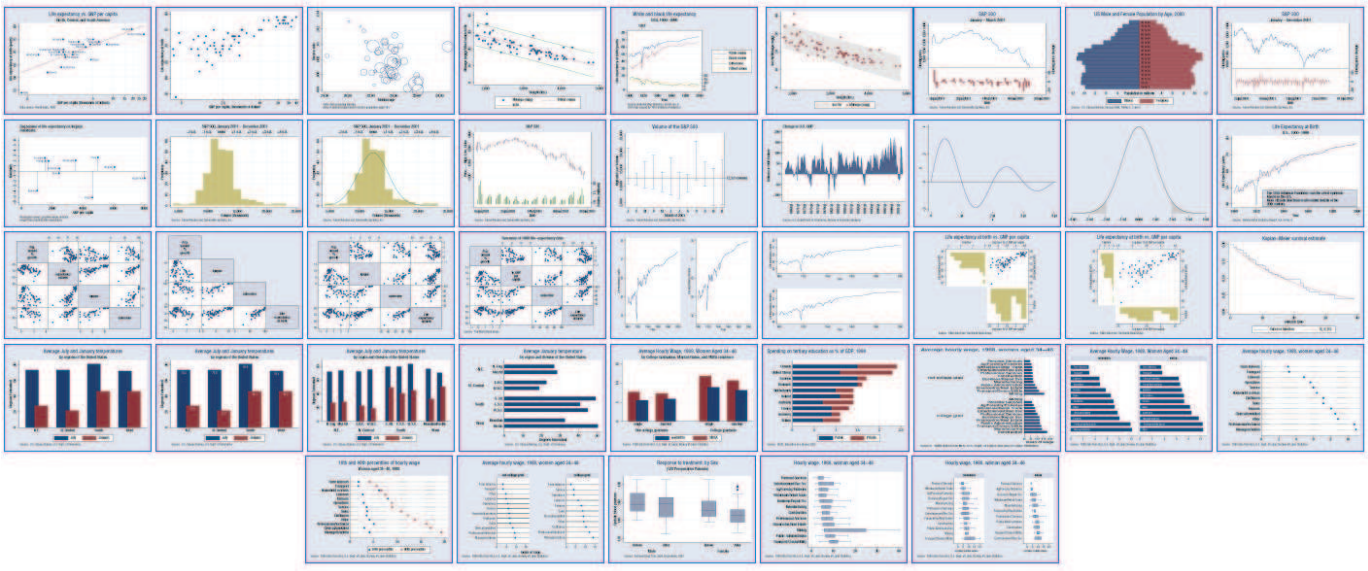
Now we can read the file we just created line-by-line and display each line in the results window using `file read`.

```
// Open the file to be read from and referencing it with the handle "readfile"
capture file close readfile
file open readfile using /home/a1jsf01/trainings/stata/do/schemne_list.txt, rea

// Read a line and save it in the macro "line"
file read readfile line
while r(eof)==0{ // Read until reaching the end of the file ( r(eof)==1 )
    di `"`line'"'
    file read readfile line
}

file close readfile
```

Introduction



Stata graphics galleries

http://www.ats.ucla.edu/stat/Stata/library/GraphExamples/default.htm

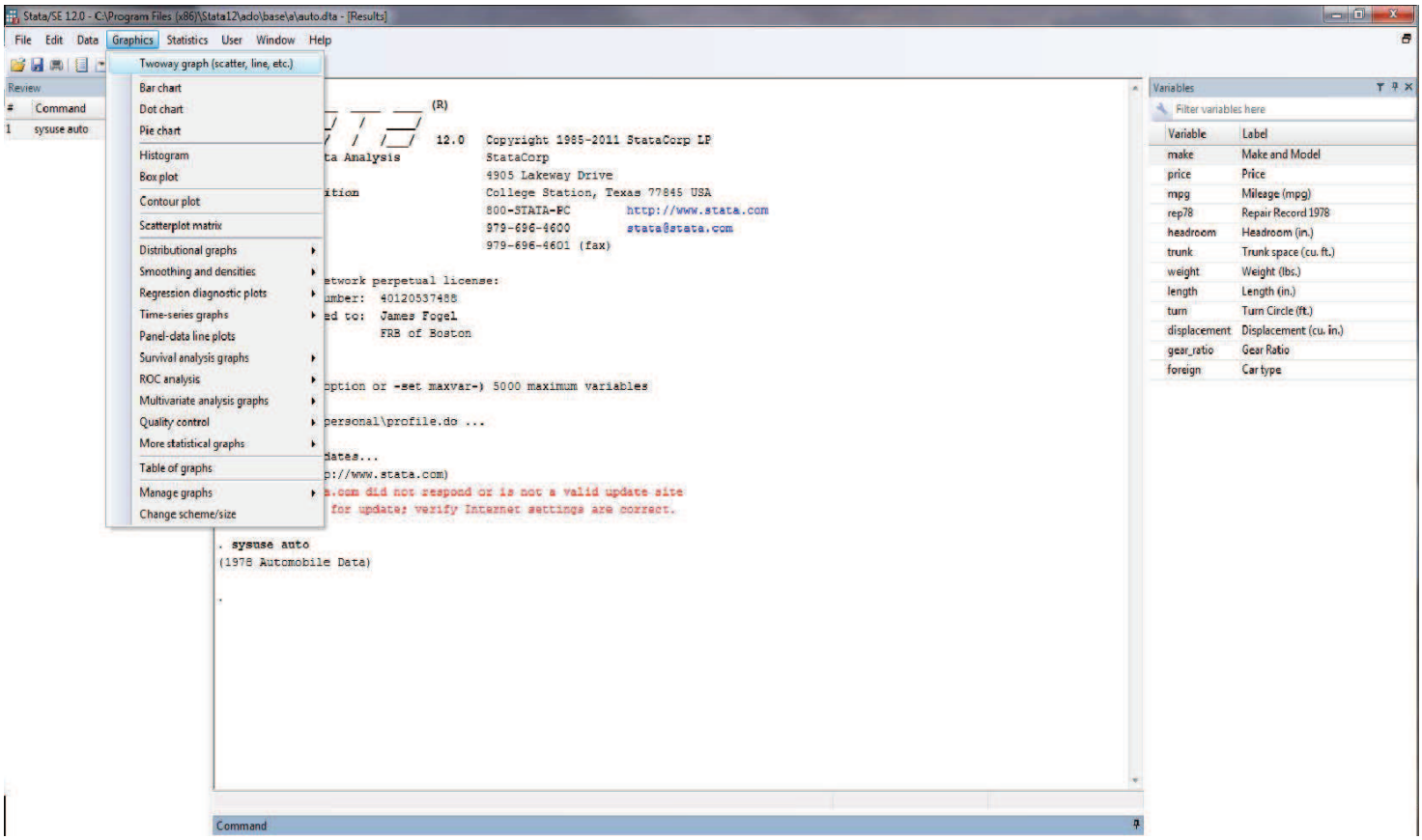
http://www.survey-design.com.au/Usergraphs.html

http://www.stata.com/support/faqs/graphics/gph/statagraphs.html

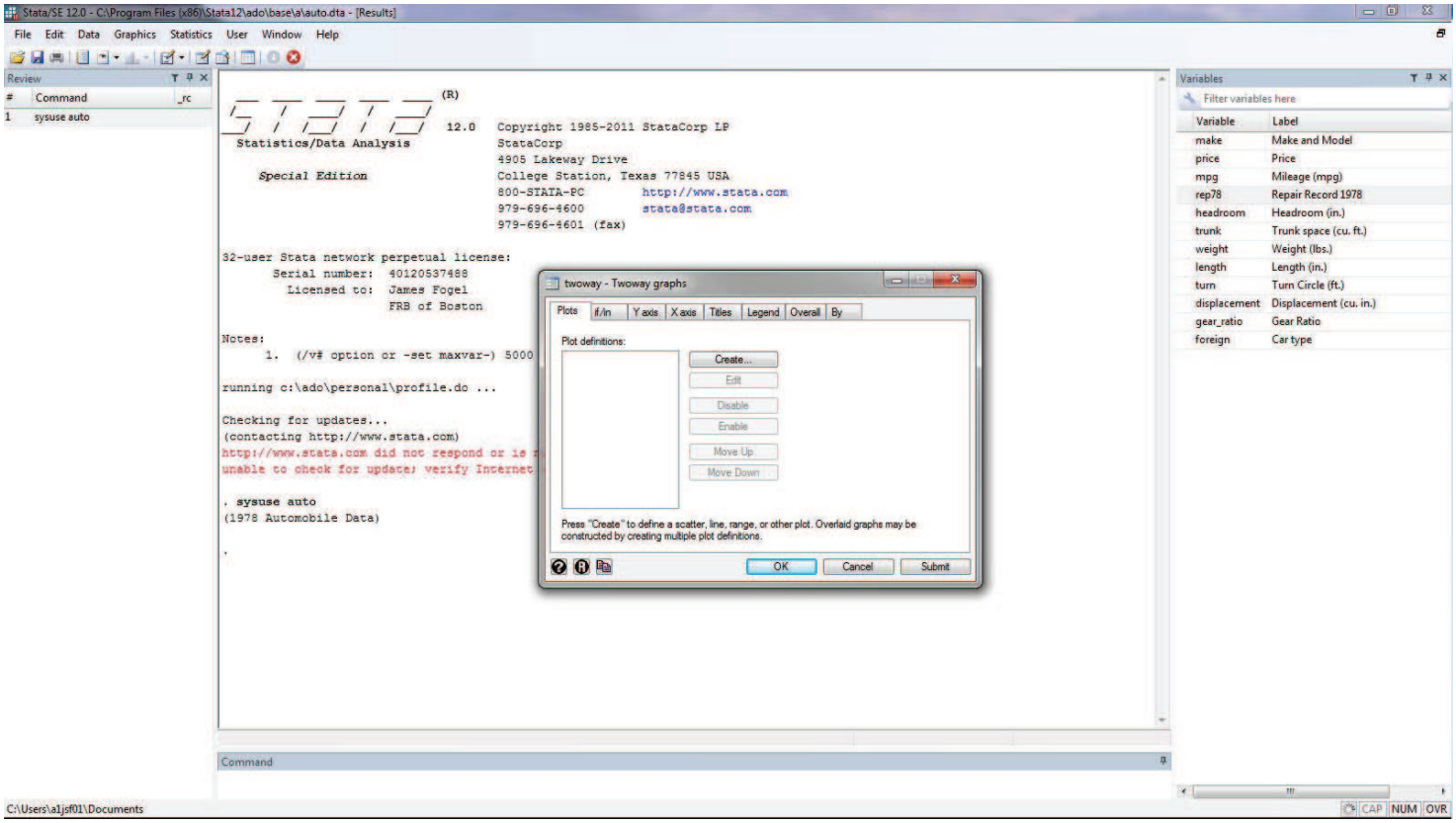
Creating figures in Stata

Stata's graphics capabilities are quite powerful, although the learning curve can be somewhat steep, as there many different types of graphs and each type comes with a vast assortment of options. In general I recommend using do-files rather than the Stata GUI, however you may want to create a graph but not know the specific options you need to specify in order to add titles and axis labels, format the legend, or change colors. In this case simply use the Stata graphics editor to create the graph. It will then give you the code necessary to re-create the graph in both the review window and the results window.

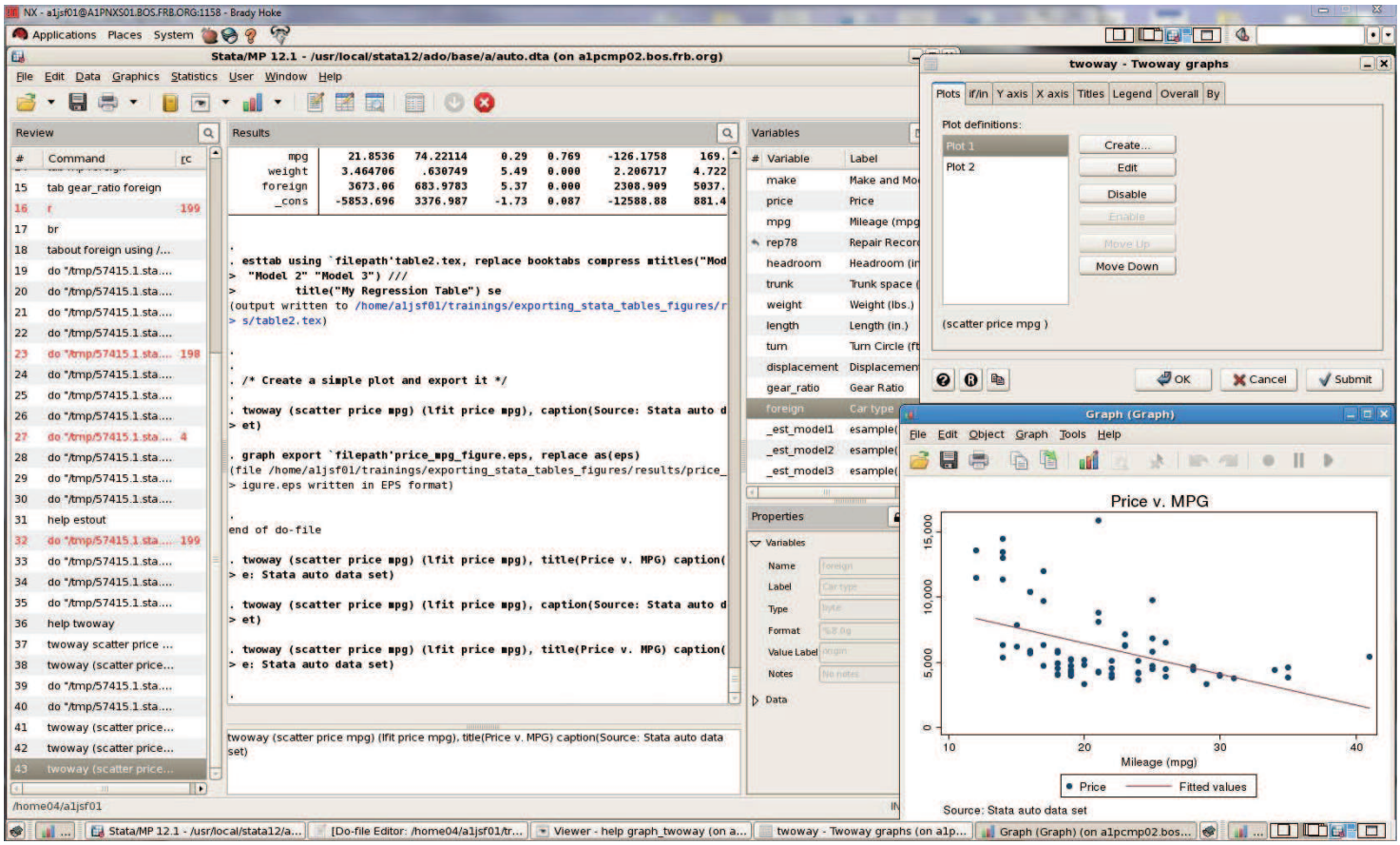
Graphics editor



Graphics editor



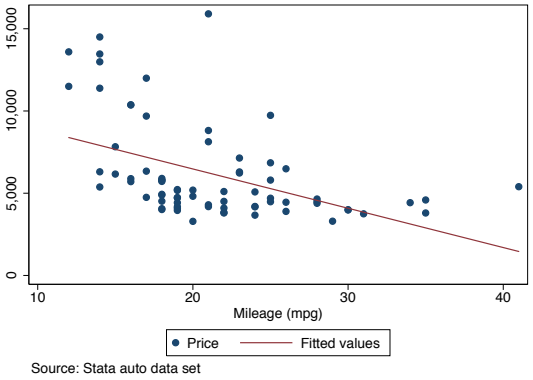
Graphics editor



Example — `twoway`

Stata's `twoway` command can be used to combine multiple (compatible) graph types into the same figure; `twoway` takes multiple graph commands as arguments, each in parentheses. In the below example, I use `twoway` to combine a scatter plot and a linear fit:

```
sysuse auto  //Use Stata's pre-loaded auto data set
twoway (scatter price mpg) (lfit price mpg), title(Price v. MPG) ///
caption(Source: Stata auto data set)
graph export 'filepath'price_mpg_figure.eps, replace as(eps)
```

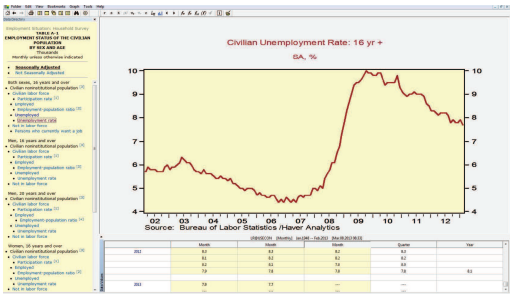


Haver

Stata provides a command, `haver use`, that allows you to load data from the Haver Analytics database, provided you know the appropriate mnemonics.

Unfortunately, `haver use` is compatible with PC Stata only, not Linux Stata. This is an inconvenience, however it is trivially easy to pull and save data using PC Stata and then access it on the Linux Cluster.

Once we identify a data series, all we need to know is the mnemonic and the file in which it's stored, both of which are listed right below the default chart in Haver.



Haver Example

Suppose we want to plot the unemployment rate from January 2000–present, with recession shading. The US unemployment rate is available in Haver under the mnemonic LR in the US Economic Indicators file USECON.DAT, as is the NBER recession indicator (RECESSM2).

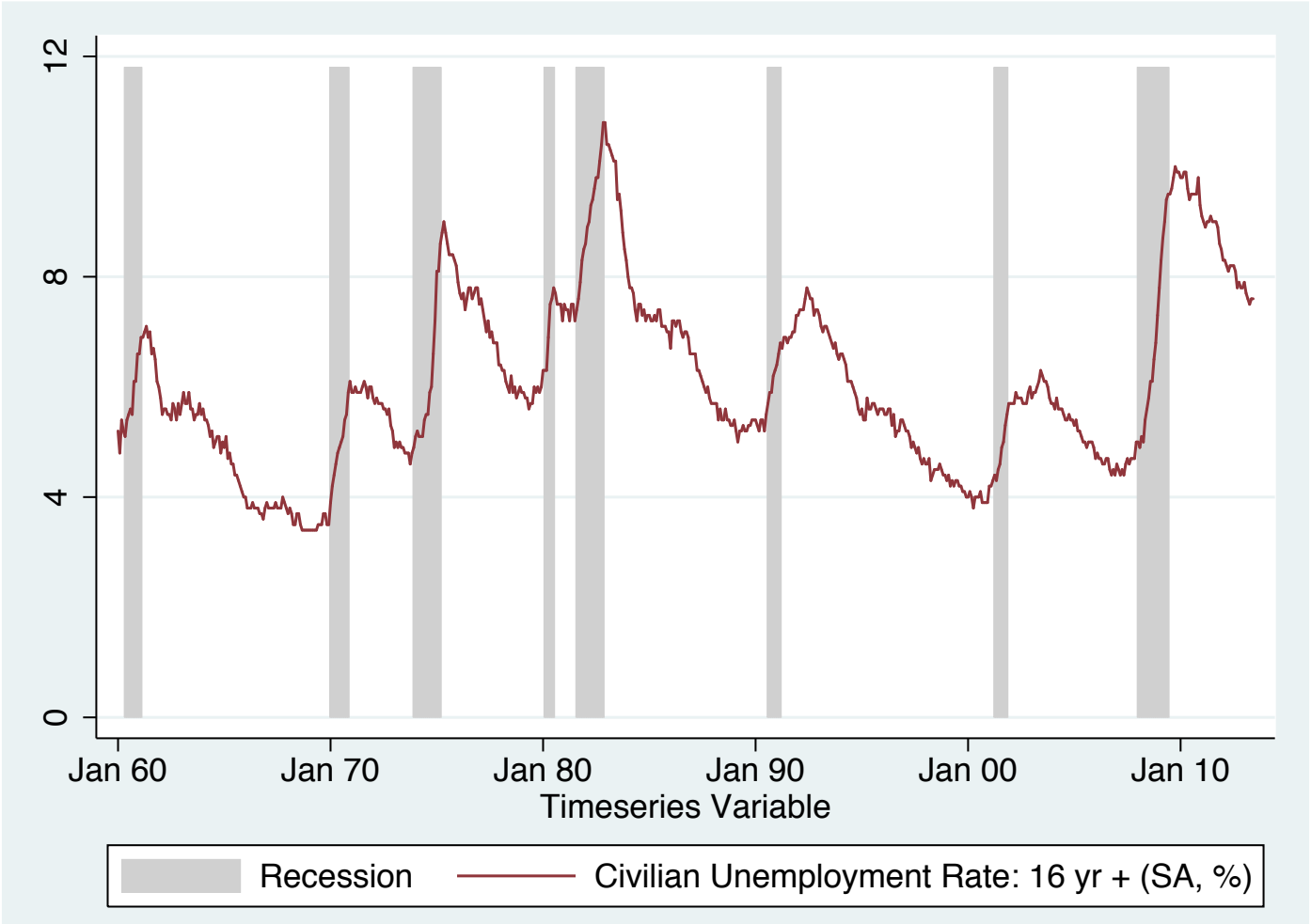
```
/* This code will work in PC Stata only */
haver use LR RECESSM2 using "G:\Deptdata\Research\Haver\USECON.DAT", clear

// Formatting date variable
rename time date
format date %tmMon_YY //Format the var so that it will be labeled as a date

// Adjust recession indicator series for consistency with unemployment rate
sum LR
gen recess = (r(max)+1)*RECESSM2

// Plot and export the series
#delimit ;
twoway
  (bar recess date, fintensity(100) lcolor(gs13) fcolor(gs13))
  (tsline LR date)
```

Haver Example



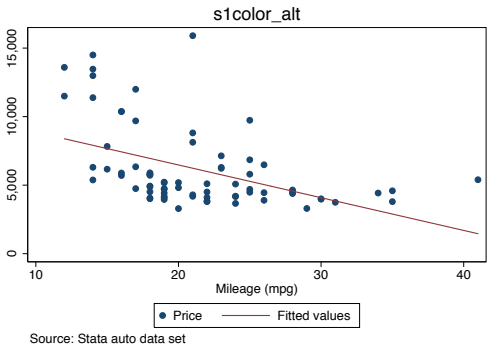
graph export

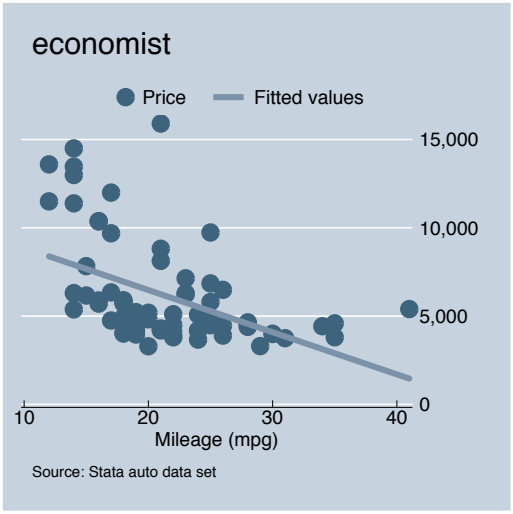
Stata's `graph export` command allows the user to export the graph currently in memory. Therefore `graph export` usually appears immediately after a Stata graph command, such as `twoway tsline varlist.` `graph export` allows you to export graphs as a number of different file types, however I usually export figures as .eps files because they are supported by both Linux and Windows Stata and because they interact well with LaTeX.

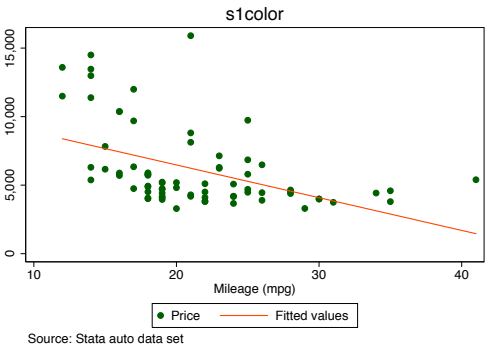
```
sysuse auto, clear
twoway (scatter price mpg) (lfit price mpg), title(Price v. MPG) ///
    caption(Source: Stata auto data set)
graph export `filepath'price_mpg_figure.eps, replace as(eps)
```

Stata provides a number of preset "schemes" that define how figures will look. You may also write your own schemes

The scheme on the left, s1color_alt, is a scheme I created myself by adjusting the default colors used by s1color.







Anyone who performs empirical research is familiar with the tedious task of turning estimation output into tables, with appropriate handling of standard errors or t-statistics, p-values, significance stars and presentation of summary statistics. Not only is the task tedious, but any manual manipulation of estimation output opens the door for mistakes, which can create a great deal of extra work for you or cause serious problems if they make it into the final version of a paper or presentation.

Fortunately, there are a number of user-written packages that greatly improve the process of outputting results. There are a number of useful packages which I will not cover (including a popular one called `outreg2`), however I will focus on two today:

1. `tabout` — Outputting tabulations of variables
2. `esttab/estout` — Outputting regression output and other statistical tables

What is tabout?

tabout allows the user to produce oneway or twoway tables of frequencies and/or percentages, as well as tables of summary statistics. These tables can be exported in a number of different formats, including .csv for use in Excel, but in my opinion the most useful is .tex, which is used by LaTeX. A much more thorough treatment of tabout can be found at
http://www.ianwatson.com.au/stata/tabout_tutorial.pdf.

A simple tabulation

The below command will produce a simple oneway tabulation of the variable *foreign* and save it as a .tex file for use in a LaTeX document:

```
tabout foreign using ‘filepath’tabout_foreign.tex, replace style(tex)
```

Code produced by `tabout`

```
Car type&No. \\
\hline
Domestic&52.0 \\
Foreign&22.0 \\
Total&74.0 \\
```

Which, when compiled, looks like

Car type	No.
Domestic	52.0
Foreign	22.0
Total	74.0

Now let's make the table prettier by using the "booktabs" option and add a column for the column percentage

```
tabout foreign using 'filepath'tabout_foreign2.tex, replace style(tex) ///
    cells(freq col) booktabs
```

```
Car type&No.&\% \\
\midrule
Domestic&52.0&70.3 \\
Foreign&22.0&29.7 \\
Total&74.0&100.0 \\
```

Car type	No.	%
Domestic	52.0	70.3
Foreign	22.0	29.7
Total	74.0	100.0

We could also do a twoway cross-tab

```
tabout foreign over40mpg using ‘filepath’tabout_foreign_over40mpg.tex, ///
replace style(tex) cells(freq col)
```

	over40mpg					
Car type	0		1		Total	
	No.	%	No.	%	No.	%
Domestic	52.0	71.2	0.0	0.0	52.0	70.3
Foreign	21.0	28.8	1.0	100.0	22.0	29.7
Total	73.0	100.0	1.0	100.0	74.0	100.0

Summary statistics for MPG and weight, broken down by foreign or domestic car

```
tabout foreign using ‘filepath’tabout_sumstats.tex, replace sum ///
cells(mean mpg median mpg mean weight median weight) style(tex)}
```

Car type	Mean mpg	Median mpg	Mean weight	Median weight
Domestic	19.8	19.0	3,317.1	3,360.0
Foreign	24.8	24.5	2,315.9	2,180.0
Total	21.3	20.0	3,019.5	3,190.0

What are esttab and estout?

The `estout` package is a user-written package which provides tools for producing publication-quality tables in Stata. It contains the following programs:

- esttab: Produces publication-style regression tables that display nicely in Stata's results window or, optionally, are exported to formats such as CSV, RTF, HTML, or LaTeX. esttab is a user-friendly wrapper for the estout command.
- estout: Generic program to compile a table of coefficients, "significance stars", summary statistics, standard errors, t- or z-statistics, p-values, confidence intervals, or other statistics for one or more models previously fitted and stored. The table is displayed in the results window or written to a text file.
- eststo: Utility to store estimation results for later tabulation. eststo is an alternative to official Stata's estimates store. Main advantages of eststo over estimates store are that the user does not have to provide a name for the stored estimation set and that eststo may be used as a prefix command.
- estadd: Program to add extra results to the returns of an estimation command. This is useful to make the the results available for tabulation.
- estpost: Program to prepare results from commands such as summarize, tabulate, or correlate for tabulation by esttab or estout.

For further details see `http://repec.org/bocode/e/estout/index.html` or `http://repec.org/bocode/e/estout/esttab.html`

esttab

For the most part you will want to use `esttab` rather than `estout` because it produces nicely-formatted tables suitable for export as LaTeX, CSV, RTF, or HTML formats. All of `estout`'s options are also available in `esttab`, however the reverse is not true. Additionally, `esttab` produces all of the code for opening the table and tabular environment that we had to provide ourselves when using tabout.

Let's produce a simple regression table using the auto data set.

```
sysuse auto, clear
eststo clear
eststo: reg price mpg weight
eststo: reg price mpg foreign
eststo: reg price mpg weight foreign
esttab using 'filepath'table1.tex, ///
    replace booktabs compress    ///
    addnote("Your footnote here")
```

	(1) price	(2) price	(3) price
mpg	-49.51	-294.2***	21.85
	(-0.57)	(-5.28)	(0.29)
weight	1.747**		3.465***
	(2.72)		(5.49)
foreign		1767.3*	3673.1***
		(2.52)	(5.37)
_cons	1946.1	11905.4***	-5853.7
	(0.54)	(10.28)	(-1.73)
N	74	74	74

t statistics in parentheses
Your footnote here
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

Alternatively, we could explicitly name each set of regression estimates and esttab any combination of the stored estimation sets.

```
sysuse auto, clear
eststo clear
eststo e1: reg price mpg
eststo e2: reg price mpg weight
eststo e3: reg price mpg foreign
eststo e4: reg price mpg weight foreign
esttab e2 e3 e4    ///
    using 'filepath'table1.tex,    ///
    replace booktabs compress    ///
    addnote("Your footnote here")
```

	(1) price	(2) price	(3) price
mpg	-49.51	-294.2***	21.85
	(-0.57)	(-5.28)	(0.29)
weight	1.747**		3.465***
	(2.72)		(5.49)
foreign		1767.3*	3673.1***
		(2.52)	(5.37)
_cons	1946.1	11905.4***	-5853.7
	(0.54)	(10.28)	(-1.73)
N	74	74	74

t statistics in parentheses
Your footnote here
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

esttab flexibility

esttab allows the user a great deal of flexibility in formatting tables. For example, we can add model titles, a caption, and replace the t-statistics with standard errors.

```
esttab using 'filepath'table2.tex, replace booktabs compress ///
    mtitles("Model 1" "Model 2" "Model 3") ///
    title("My Regression Table") se
```

My Regression Table

	(1) Model 1	(2) Model 2	(3) Model 3
mpg	-49.51	-294.2***	21.85
	(86.16)	(55.69)	(74.22)
weight	1.747**		3.465***
	(0.641)		(0.631)
foreign		1767.3*	3673.1***
		(700.2)	(684.0)
_cons	1946.1	11905.4***	-5853.7
	(3597.0)	(1158.6)	(3377.0)
N	74	74	74

Standard errors in parentheses
* $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$

A few key points about outputting results

- **You should NEVER be copying and pasting Stata output from the results window or a log file!**
- The only way to learn `tabout` and `esttab` is to start using them, dig through the documentation, Google your questions, or ask other RAs for help. You will pick them up quickly.
- Once you have spent some time using `tabout` and `esttab` you will have built up your own library of templates, which you will be able to copy, paste, and make minor edits to for future use!

Programs & Ado-files

First, some Stata nomenclature:

1. Stata formally defines a program as a set of Stata commands that includes a `program` statement.
2. An **ado-file** is simply a file that stores a program (see, for example, `viewsource estout.ado`

You can easily write a Stata program, stored in an ado-file, that handles all the features of official Stata commands such as the `if` and `in` options. You can even write a help file that documents its operation, for your benefit and for those with whom you share the code.

But you probably shouldnt! In most cases, existing Stata commands – official or user-written – will perform the tasks you need. Use Stata's search features such as `findit` and the Stata user community (Statalist) to ensure that the program you envision has not already been written.

A Stata program adds a command to Statas language. The name of the program is the command name, and the program must be stored in a file of that same name with extension .ado, and placed on the adopath i.e., the list of directories that Stata will search to locate programs. A program begins with the `program define program_name` statement, which usually includes the option , `rclass`. The program name should not be the same as any Stata command, nor the same as any accessible user-written command!

The `syntax` command will almost always be used to define the commands format. For instance, a command that accesses one or more variables in the current data set will have a `syntax varlist` statement. The `syntax` statement does allow you to specify `[if]` and `[in]` arguments, which allow commands to limit the observations used. The `syntax` statement may also include a `using` qualifier, allowing your command to read or write files. Really, any feature you find in an official Stata command can be implemented with an appropriate `syntax` statement.

A sample program from `help return`:

```
program mysum, rclass
    syntax varname
    tempvar new
    quietly {
        count if !missing(`varlist')
        return scalar N = r(N)
        gen double `new' = sum(`varlist')
        return scalar sum = `new'[_N]
        return scalar mean = return(sum)/return(N)
    }
end
```

This program can be executed as `mysum varname`. It prints nothing but places three scalars in the return list. The values `r(N)`, `r(sum)`, and `r(mean)` can now be referred to directly.

```
mysum mpg
display r(N) "  " r(sum) "  " r(mean)
>74  1576  21.297297
```

Mata — Stata's very own matrix programming language!

Stata contains a full-blown matrix language, *Mata*, with many of the capabilities of MATLAB and R. Why should you care?

1. Mata is automatically compiled into bytecode and hence runs many times faster than the standard Stata language.
2. Mata circumvents the limitations of Stata's traditional matrix commands
3. Mata contains a large library of mathematical and matrix functions, including optimization routines, equation solvers, decompositions, and probability density functions.

I am not going to dwell on Mata syntax in this training — it is largely similar to C and its handling of matrices is broadly similar to that of other matrix languages like MATLAB or R (see `help mata`).

```
sysuse auto , clear

// Entering Mata
mata
// Defining matrices
x = st_data(., ("mpg"))
y = st_data(., ("price"))
cons = J(rows(x), 1, 1)
X = (x, cons)

// Some linear algebra
beta_hat = ( invsym( X   *X))*( X   *y)
e_hat = y - X * beta_hat
s2 = (1 / (rows(X) - cols(X))) * (e_hat     * e_hat )
V_ols = s2 * invsym (X    *X)
se_ols = sqrt(diagonal(V_ols))

// Returning results
st_matrix ("beta_hat", beta_hat)
st_matrix ("se_ola", se_ols)
// Leaving Mata
end
```

But you should never program OLS (or something equally simple) in Mata. **Why?**

Stata has already programmed it for you in its `regress` command! You do not want to write redundant commands or code because it is a waste of time, may contain errors, is unlikely to be as efficient as the commands Stata or other users have written, and because when you share your code with someone else, they will have a much easier time understanding your code if you stick to standard Stata commands.

Writing your own programs and Mata code can be very useful, but only when you know that you will be performing the same operation over and over and you are confident that no one else has written code that will accomplish the same task.

Automating Data Downloads (Linux Stata)

Downloading files from a website is easy to automate if the files are stored in a systematic way. This is a good practice for several reasons it makes the task more efficient, well documented, and easily replicable. Generally, the actual downloading takes place outside of the Stata application, but Stata can coordinate the downloads using local variables and invoking shell commands (see previous section) within a loop. Perhaps the simplest Linux command to quickly download a file is the `wget` command, which downloads the path it is provided (for example, `wget "http://www.example.com/file.dat"`). The `wget` command also has many other options to enhance its functionality.

For example, to download ten years of annual CSV data one might write:

```
forvalues year = 2000/2010 {
    !wget "http://www.example.com/‘year’data.csv"
}
```

Debugging

- `pause` — `pause` will pause execution of your code and return control of Stata. You may continue execution from that point by typing `end` or break execution with `BREAK`.
- `trace` — `trace` displays each line from each ado file it runs. It allows you to see exactly what is going on within each command you use. Unfortunately, `trace` produces a large amount of output that can be difficult to sift through.
- `findbug` — `findbug` is a user-written command that is a nice replacement for `trace`. `findbug` keeps a trace of of the execution of your code and, if an error occurs, displays the line that triggered the error along with several preceding and succeeding lines.

capture

`capture` executes a command, suppressing all its output (including error messages, if any) and issues a return code of zero. The actual return code generated by command is stored in the built-in scalar _rc.

Capture is useful if you know that a command may cause an error message but you would like to disregard the message and continue execution. For example suppose you want to close any open log file, but you aren't sure that a log file is open: `apture log` lose=.

`assert` and `confirm`

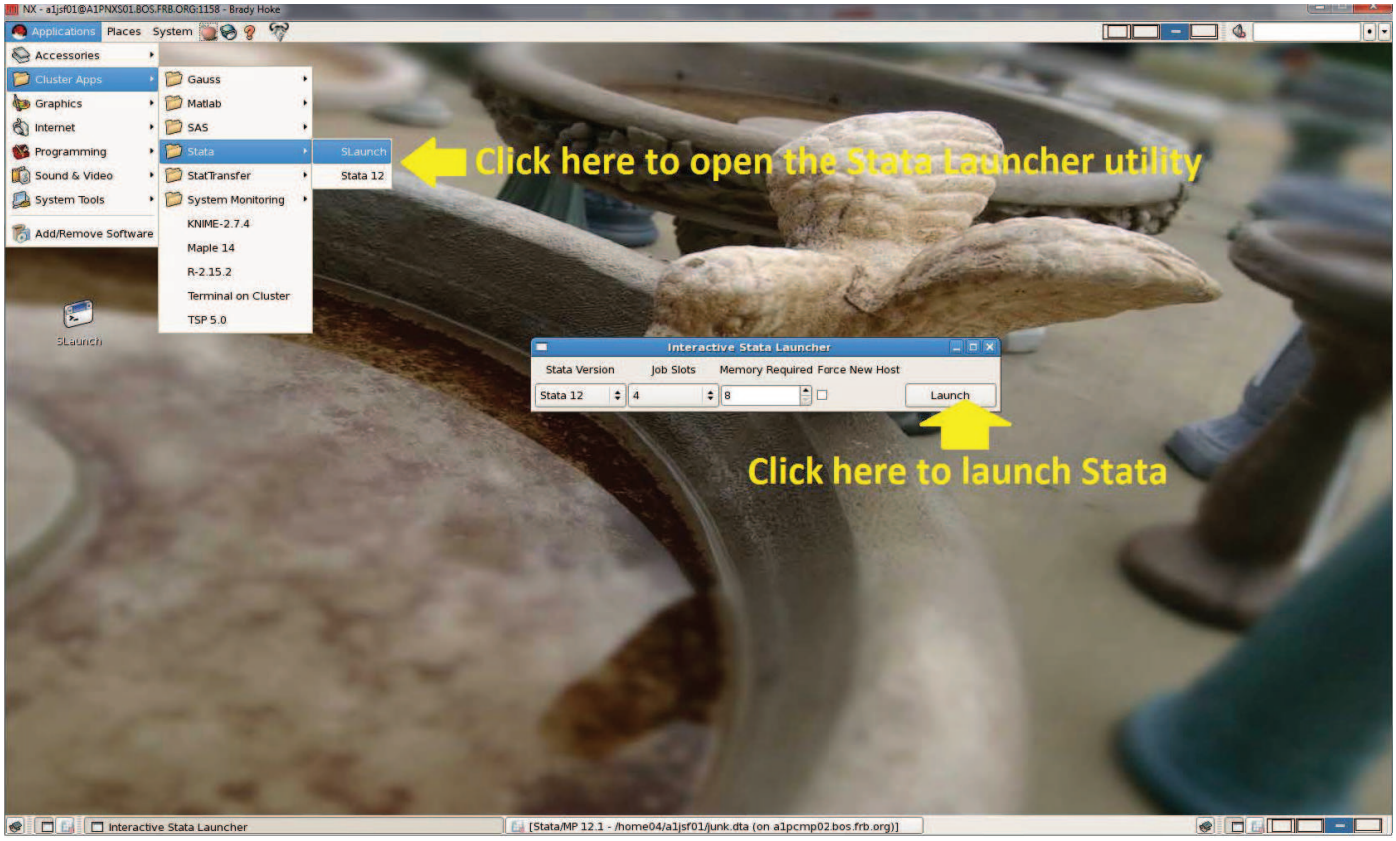
`assert` tests whether or not a condition is true and generates an error message if it is false. It is a very useful tool for ensuring that your code is running as you expect it should. For example, `assert _merge==3==` after a merge will check that all observations merged successfully and will generate an error if not.

`confirm` may be used to verify that files or variables exist and/or are of the expected type.

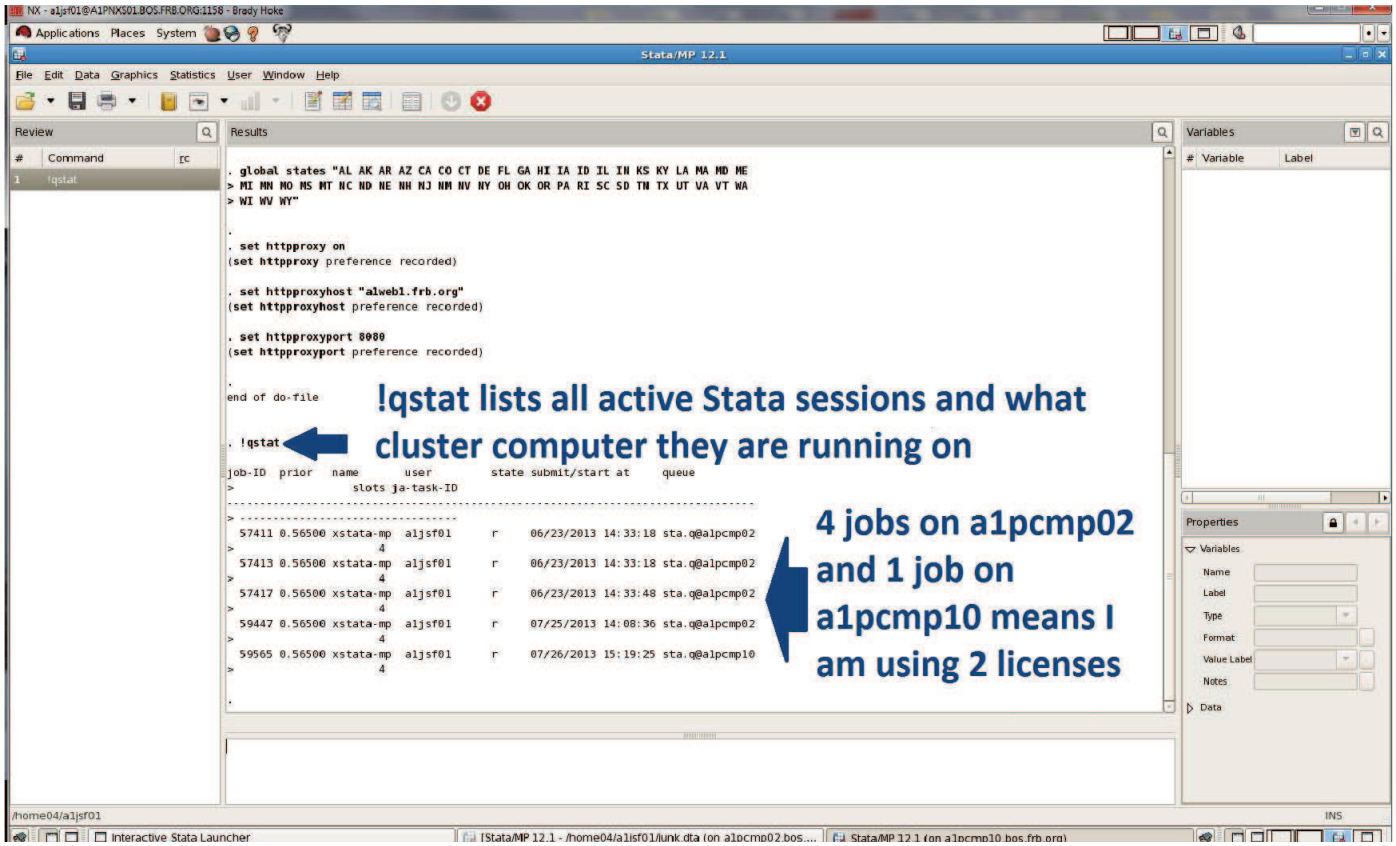
Managing Licenses on the Linux Cluster

We only have 25 licenses for Linux Stata so if you use more than you need you may prevent someone else from using Stata. We have about 45 economists and RAs so it is very important to not waste licenses.

Use the Stata Launch utility to run multiple sessions on one license



You can check your license usage with the `!qstat` command



Batch Stata

Instead of running Stata interactively, you may submit individual jobs in batch mode. In batch mode Stata executes your do-file entirely behind the scenes and creates a log file to keep track of execution. Batch mode uses a license only while Stata is executing and immediately relinquishes that license once execution completes. Batch mode is ideal for running do-files that take a long time but don't need to be actively monitored, such as do-files that automate large data set creation tasks or perform complex and time-consuming estimation routines.

To run Stata in batch mode open the terminal window and enter at the command line:

```
qbatch stata filepath/yourdofile.do
```

Speak with Jones for further details.

Stata Resources

Have questions? Use the following resources:

1. Stata help files
2. Google
3. Stata manuals (located by CLS on T-8)
4. Other RAs
5. Statalist (be sure to read the Statalist FAQs before posting!).