Getting Started in Stata Without Losing Time

Edwin Leuven
ENSAE-CREST

September 9, 2009

Contents

Preface

This document is intended to get you quickly going in Stata. It focuses on getting common data tasks done and introduces Stata programming. It does not discuss the details of statistical/econometrical analysis in Stata. Comments/suggestions are welcome at: edwin.leuven@ensae.fr

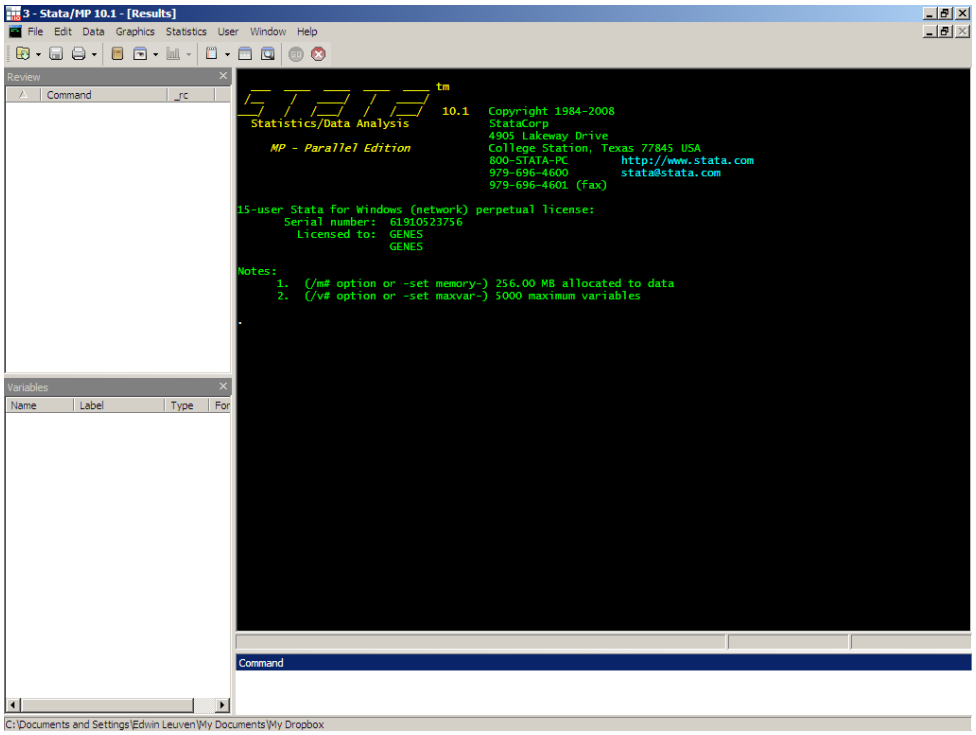


Figure 1: Stata Window

1 The Stata User Interface

Figure 1 shows how Stata looks (in MS Windows, the general layout is similar on other platforms). At the bottom you see the Command line. In the command panel you can type Stata commands which on <Enter> are executed immediately. You will see the results in the large Results panel. At the left you find the Review and Variables panels. The former shows which commands you have executed in your current Stata session. Clicking on a command in the Review panel will copy it to the Command line. An often more convenient way to recall earlier commands in the command line is by pushing <PgUp>. Similarly <PgDn> moves you down in the command history. The "Variables" panel lists the names of the variables in the currently loaded dataset. Clicking in the panel will copy the variable name to the command line.

You can execute Stata commands on the command line, or put them in a plain text file with the extension .do. Stata has a simple build-in editor which you can use to edit these files (you can also use the editor to run the file or a selection in Stata). Using these so-called do-files is of course the preferred way to organize your work., but the command line is very convenient when exploring your data. You can also use it as a calculator: to calculate for example the square root of 2, type the following (without the dot which is the Stata prompt)

```
. display sqrt(2)
1.4142136
```

note that names in Stata are case sensitive which means that -display- is not the same as -Display-. More in general -display- prints strings and values of scalar expressions which is handy to produce output in the programs that you may write.

2 Moving around on your file system

In the status line (completely at the bottom of the Stata window) you can see the current working directory, which is also displayed with the command -pwd-. It usually a good idea to ensure that this is the directory where your project files are. You can do this using the DOS-style command -cd-, as in

```
. cd "c:\Documents and Settings\John Doe\My Documents\myproject"
```

You need the quotes here because of the space in the path. To move a directory down (to c:\Documents and Settings\John Doe\My Documents) use `'cd ..'`, and `'cd myproject'` brings you back up to where you were before.

To list all files in the directory use the command -dir-. The following lists the do-files

```
. dir *.do
```

3 Stata syntax

With a few exceptions, the basic language syntax in Stata is

```
[prefix :]  command [varlist] [=exp] [if] [in] [weight] [using filename]
[, options]
```

where the square brackets indicate optional elements (what is optional depends in practice of course on the specific command you are invoking). Some commands can be abbreviated. If this is the case then the documentation and the help files will underline the shortest possible abbreviation as above.

Suppose you want to estimate an OLS regression of the variable *lnincome* on the variable *educ* for men only, this would look something like this:

```
. reg lnincome educ if female==0
```

the full syntax of -regress- is

```
. regress depvar [indepvars] [if] [in] [weight] [, options]
```

4 Getting help

Getting help on a command in Stata is easy, typing

```
. help command
```

will open a window that explains the full syntax of -command- and often includes examples. Use -help- if you want to find out more about the commands discussed in this document than discussed in this document!

To search for a command you can use

```
. findit keyword
```

which will search the keynote database and the Internet and pop-up a window with the search results. -hsearch- searches the help files only.

5 Reading and writing data

Stata loads data into memory. The size of the data you can handle in Stata is therefore limited by the physical memory on your computer (this is often lest restrictive than it sounds like). One implication is that you will need to ensure that Stata has access to sufficient memory to fit your dataset. If this is for example 500MB, then you can set the size of Stata's memory area to 500 MB as follows

```
. set memory 500m
```

You can only -set memory- when no data is loaded. To clear Stata's memory use

```
. clear
```

Stata datasets are single rectangular tables (n observations, k variables) and have the extension "dta". You can read these into memory with the command -use- as follows

```
. use filename
```

If there is unsaved data in memory Stata will refuse (for your own safety) to read in the data. You will first need to clear Stata's memory area using -clear-. You can add hte option 'clear' to -use- which will automatically clear the memory before loading the data.

You can also read (ASCII) non-Stata data files. If they are in fixed-column format use -infix-. For example like this

```
. infix rate 1-4 speed 6-7 acc 9-11 using highway.raw
```

which reads the file highway.raw where the variable rate occupies position 1-4, the variable speed position 6-7 and the variable acc position 9-11. Note that variable names can be up to 32 characters and are case sensitive.

If you have data with one observation per line and where variables are separated by a delimiter such as a comma or a tab you use -insheet-. These files are typically generated with a spreadsheet or database. The following reads a file auto.csv where variables are delimited with a semicolon

```
. insheet using auto.csv, delim(";")
```

It is also possible to read free format data files. See -help infiling- for a more elaborate reference on how to read non-Stata data into memory.

To write the data in memory to disk you need to -save- like this

```
. save filename
```

If the file already exists, Stata will refuse to save the data and you will need to add the option 'replace' to overwrite the file.

6 Looking at your data

Once you have read the data into memory you might want to get a description of your dataset. Below we read one of Stata's example datasets, and the command -describe- generates a data description like this:

```
. sysuse lifeexp
. d

Contains data from C:\Program Files\Stata10\ado\base/l/lifeexp.dta
  obs:            68                          Life expectancy, 1998
 vars:             6                          26 Mar 2007 09:40
```

```
 size:         3,196 (99.9% of memory free)   (_dta has notes)
-------------------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
-------------------------------------------------------------------------------
region          byte   %12.0g      region     Region
country         str28  %28s                   Country
popgrowth       float  %9.0g                 * Avg. annual % growth
lexp            byte   %9.0g                 * Life expectancy at birth
gnppc           float  %9.0g                 * GNP per capita
safewater       byte   %9.0g                 *
                                              * indicated variables have notes
-------------------------------------------------------------------------------
Sorted by:
```

the output starts by listing the filename, the number of observations and variables and size. It then lists all the variables, their storage type, labels and concludes with the sort order of your data. The variable "country" for example is a string variable, and the rest of the variables are numeric. Not all numeric variables have the same type. "region" for example is stored as a byte whereas "popgrowth" is stored as a float. Because Stata needs to keep the dataset in memory it is is economical in the way it stores data. There is for example no need to store a binary variable as a float. Taking advantage of this, the command -compress- will try to reduce the amount of memory used by your data. To read more about Stata's data types read -help data_types-

The following list the contents of a dataset without loading it into memory

```
. describe using filename
```

The command -summarize- calculates summary statistics:

```
. sum

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
      region |        68         1.5    .7431277          1          3
     country |         0
   popgrowth |        68    .9720588    .9311918        -.5          3
        lexp |        68    72.27941    4.715315         54         79
       gnppc |        63    8674.857    10634.68        370      39980
-------------+--------------------------------------------------------
   safewater |        40        76.1    17.89112         28        100
```

The output lists the number of observations used to compute the statistics, the sample mean, standard deviation and the smallest and largest value in your data. For the variable "country" the output lists 0 observations and no summary statistics. This is of course because "country" is a string variable as we saw in the output of -describe-.

We could also see above that our dataset has 68 observations. The output for "gnppc" however shows only 63 observations which means that 5 observations have missing values for this variable. The basic missing (numerical) value is shown in Stata as a single dot ("."). Most (but not all) Stata commands drop observations with one or more missing values.

Missing values are internally represented by a value higher than the highest possible value of the data type of your variable. As a consequence you can use missing values in logical conditions as follows

```
. sum popgrowth if gnppc>=.

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   popgrowth |         5          .9    1.202082        -.5        2.8

. sum popgrowth if gnppc<.
```

```
   Variable |       Obs        Mean    Std. Dev.       Min        Max
------------+--------------------------------------------------------
   popgrowth |        63    .9777778    .9183513        -.4          3
```

where the first command summarizes the variable *popgrowth* for all observations where *gnppc* is missing, and the second for the observations where *gnppc* is not missing. You can also use the function `missing(`*`expression`*`)` which returns 1 (true) if *`expression`* is missing and 0 (false) otherwise. The following gives some examples of missing values

```
. di missing(1)
0
. di missing(1/0)
1
. di missing(.)
1
. di missing(.*4)
1
. di missing("")
1
. di missing("a")
0
. sum pop if mi(gnp)

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   popgrowth |         5          .9    1.202082        -.5        2.8
```

The last line used `mi()` the abbreviated version of `missing()`. Note that the variable names are abbreviated here, which one can do if the abbreviation has a unique match.

When passing varlists to commands one can not only abbreviate variable names, but also use wildcards. For example: `kid*` are all variables starting with "kid", `kid*x` all variables starting with "kid" and ending with "x", `ind?` all variables starting with "ind" and having one following character, etc. See varlist for more details. This feature is very useful when you like to avoid a lot of useless typing.

To list the values of your variables (i.e. look at your data) you can use -list-. I personally prefer -clist- which does not format the output but is much quicker in large datasets. Both -list- and -clist- allow you to list the values of selected variables and for a subset of your data. For example like this

```
. cl country-safe if safe>=. & gnp>=.

                         country  popgrowth       lexp      gnppc  safewater
  7.       Bosnia and Herzegovina        -.5         73          .          .
 44. Yugoslavia, FR (Serb./Mont.)         .5         72          .          .
```

Alternatively you can use -browse- which will open a spreadsheet type window that allows you to browse your data area. -edit- does the same but then you can change your data and drop variables in the browser.

One- and two-way frequency tables are generated with -tabulate-. For example to obtain a frequency table of region for the countries where at least 50 percent of the population have access to safe water you might type

```
. ta region if safe>50

      Region |      Freq.     Percent        Cum.
-------------+-----------------------------------
Eur & C.Asia |         44       66.67       66.67
        N.A. |         13       19.70       86.36
        S.A. |          9       13.64      100.00
-------------+-----------------------------------
       Total |         66      100.00
```

until you realize that this also includes observations where safewater has missing values, after which the following

```
. ta region if safe>50 & safe<.

      Region |      Freq.     Percent        Cum.
-------------+-----------------------------------
Eur & C.Asia |         17       44.74       44.74
        N.A. |         12       31.58       76.32
        S.A. |          9       23.68      100.00
-------------+-----------------------------------
       Total |         38      100.00
```

produces the correct table. By default -tabulate- does not include missing values of the variable it is tabulating in the frequency table, to add them use the option 'missing'. Similar to the one-way table above, -`tab var1 var2`- will produce a two-way tabulation.

The command -table- produces tables with summary statistics as follows

```
. table region, c(m safe n safe p10 lexp n lexp ) row

------------------------------------------------------------------------------
      Region | mean(safewa~r)     N(safewa~r)       p10(lexp)         N(lexp)
-------------+----------------------------------------------------------------
Eur & C.Asia |       79.82353              17              67              44
        N.A. |              75              13              64              14
        S.A. |            71.2              10            64.5              10
             |
       Total |            76.1              40              67              68
------------------------------------------------------------------------------
```

the option 'c()' is an abbreviation of the full option '`contents(`*`clist`*`)`' where *`clist`* is a list of up to 5 statistics in which the statistic is followed by the varname as in the example: 'm safe' will calculate the mean of the variable safe, 'n safe' a count of non-missing observations of safe, and 'p10 lexp' the 10th percentile of the variable lexp. See -help table- for the complete syntax and a full list of available statistics.

An alternative way of getting summary statistics per region would be to repeat the command -summarize- on subsets of the data defined by the variable *region*. You can achieve this in Stata with the -by- prefix with the following syntax:

```
by varlist : command
```

by will run `command` on the subset of your data consisting of the observations having the same values for *`varlist`*. Most Stata commands can be used with -by-. To -summarize- for example population growth by region we do the following

```
. by region: sum popg
not sorted
r(5);
. sort region
. by region: sum popg

-> region = Eur & C.Asia

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   popgrowth |        44        .525    .7175945        -.5        2.8

-> region = N.A.

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   popgrowth |        14    1.692857    .7488086         .7          3

-> region = S.A.
```

```
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
   popgrowth |        10        1.93    .6165316         .7        2.9
```

As can be seen here, the data need to be sorted by the variables passed to -by-. At first this was not the case and Stata threw an error. To sort the data we use -sort-, after which Stata calculates summary statistics for every region separately.

7 Manipulating data

After having read your data into Stata you will probably like to prepare your data for analysis. You may want to create variables using information in other variables, or make modifications to existing variables.

The command generate creates new variables based on an expression:

```
. gen agesq = age^2                    // use power operator to calculate square
. gen lnincome = log(income)           // use logarithmic function
. gen byte retire = age>=65 if age<.   // create indicator variable (0/1)
. gen rnd = uniform()                  // random nr from uniform distribution
. gen z = x + 50*invnormal(uniform()) // create random z-score
. gen prob = normal(z)                 // prob from cumulative normal
. gen price = real(stringvar)          // numeric from string variable with numbers
```

As can be seen from the examples above, algebraic or string expressions are used in a standard way and can contain operators, functions and combinations of both. Arithmetic expressions containing a missing value will evaluate to a missing value.

Stata has a broad collection of functions that cover mathematical, probability, random number, string, date and time, matrix operations. As usual -findit- is your friend, as is -help functions-.

Stata uses the following operators in expressions:

Arithmetic		Logical		Relational	
+	addition	&	and	>	greater than
-	subtraction	|	or	<	less than
*	multiplication	!	not	>=	> or equal
/	division	~	not	<=	< or equal
^	power			==	equal
-	negation			!=	not equal
+	string concatenation			~=	not equal

The order of evaluation (from first to last) of all operators is ! (or ~), ^, - (negation), /, *, - (subtraction), +, != (or \=), >, <, <=, >=, ==, &, and |.

The command -replace- works in the same way as -generate-, but instead of generating a new variable it changes the values of observations in an existing one, for example like this

```
. replace agesq = agesq/100
. replace educ = 12 if missing(educ) & highschool==1
```

where the first expression divides all observations of the variable *agesq* by 100, the second changes missing values to the number 12 in the variable *educ* for all observations where the variable *highschool* equals 1.

A function that deserves explicit mention is `sum()`. To illustrate what it does consider the following

```
. set obs 3
obs was 0, now 3
. g x = uniform()
. g y = sum(x)
. cl
             x          y
  1.  .1369841   .1369841
  2.  .6432207   .7802048
  3.  .5578017   1.338006
```

as you can see the *j*-th observation on y contains the sum of the first through *j*th observations on x. To create the total sum we can use something called "explicit subscripting on variables". In Stata you can access individual observations of a variable by putting the observation number in square brackets after the variable name. This can be a number or an expression. In expressions the current observation number is referred to by '_n', the last observation by '_N'. It works like this:

```
. g y1 = x[3]
. g y2 = y[_n-1]
(1 missing value generated)
. g y3 = y[_N]
. cl
             x          y         y1         y2         y3
  1.  .1369841   .1369841   .5578017          .   1.338006
  2.  .6432207   .7802048   .5578017   .1369841   1.338006
  3.  .5578017   1.338006   .5578017   .7802048   1.338006
```

as you can see *y1* contains the 3rd observation of *x*, *y2* is the lagged value of *y* (note that the first observation has a missing value because there is no observation 0), and *y3* now contains the total sum of *x* which was the last value of *y*.

The by prefix command may be combined with _n to provide subscripting within groups. For example:

```
. sort country year
. by country: gen gdpgrowth = (gdp - gdp[_n-1])/bp if year[_n]==(year[_n-1] + 1)
```

Note that the -if- condition checks that the current and previous observation are actually two consecutive years. If this is not the case Stata will set the value of *gdpgrowth* for that observation to missing. For more details on subscripting see -help subscripting-.

We saw above that we could calculate the total sum of a variable as follows

```
. gen y = sum(x)
. replace y = y[_N]
```

another way of achieving this is to use extensions to generate called -egen-

```
. egen y = sum(x)
```

-egen- provides many convenient functions to create new variables, some of which (`sum()` among others) can be used with -by-. Others provide quick ways to calculate statistics of lists of variables such as the sum or the average. If you find that have difficulties creating the variable you need you might get lucky in -help egen-.

We now have the tools to do more fancy things. One of them is to make datasets of summary statistics. We can use -egen- to calculate means, percentiles, etc., say by country and year using -by-. After this we keep one observation per by-combination as follows

```
. sort region
. by region: egen mx = mean(lexp)
. by region: egen p10x = pctile(lexp), p(10)
```

```
. by region: keep if _n==1
. keep region mx p10x
. list, noobs
  +--------------------------------+
  |       region         mx   p10x |
  |--------------------------------|
  | Eur & C.Asia   73.06818     67 |
  |         N.A.   71.21429     64 |
  |         S.A.       70.3   64.5 |
  +--------------------------------+
```

It turns out there is a convenient command called -collapse-, which does the same in a single line:

```
. collapse (mean) mx = lexp (p10) p10x = lexp, by(region)
```

Note above the two different uses of -keep-. The first, `keep [if]`, keeps a subset of the observations, whereas the second, `keep varlist`, keeps a subset of the variables. You can use the command -drop- in a similar way if this is more convenient.

Finally, to rename variables use -rename-, and to change the order in which the variables appear in your data use -order-.

8 Combining and reshaping data

When preparing data for analysis you may need to combine several datasets. To stack data sets you can use -append-, more exactly like this:

```
. append using filename
```

which will append a Stata dataset on your file system to the dataset in memory. Variables with the same name in both datasets get stacked on top of each other. If a variable appears in only one dataset the observations corresponding to the other dataset will be set to missing, as illustrated here:

(a.dta)

x	y
1	1.2
2	2.3
3	0.5

(b.dta)

x	z
6	0.03
12	0.01

(b appended to a)

x	y	z
1	1.2	.
2	2.3	.
3	0.5	.
6	.	0.03
12	.	0.01

Another common task is (match) merging as in this example:

(c.dta)

id	y
1	1.2
2	2.3
3	0.5

(d.dta)

id	x
1	3.5
2	1.0
6	0.1

(d merged to c)

id	y	x	_merge
1	1.2	3.5	3
2	2.3	1.0	3
3	0.5		1
6	.	0.1	2

which is done with the command -merge-. Assuming that both datasets are sorted on the merge variable(s), the following loads dataset c.dta and then joins dataset d.dta in a 1-to-1 merge

```
. use c
. merge id using d
```

-merge- will create a variable _merge, the values of which indicate the merge result: 1) observations from the master dataset (the one in memory) without a

match in the using dataset, 2) observations from the using dataset without a match in the master dataset, and 3) successful merges. -merge- can do 1-to-1 and match merges.

Finally you might need to change the layout of your data. This arises for example when your data is organized in columns (for example one variable per year) whereas you need your data to be organized in rows (one observation per year). This you can achieve with -reshape-. Consider the following two example datasets from Stata's help file in 'wide' form and 'long' form.

(wide form)

id	sex	inc80	inc81	inc82
1	0	5000	5500	6000
2	1	2000	2200	3300
3	0	3000	2000	1000

(long form)

id	year	sex	inc
1	80	0	5000
1	81	0	5500
1	82	0	6000
2	80	1	2000
2	81	1	2200
2	82	1	3300
3	80	0	3000
3	81	0	2000
3	82	0	1000

You can move from wide to long with

```
. reshape long inc, i(id sex) j(year)
```

where `year` does not exist yet in your original (wide) dataset, but is the name of the variable that Stata will create to contain the values of the stubs (80, 81, 82). Similarly, one can go from long to wide with

```
. reshape wide inc, i(id sex) j(year)
```

Although much less common, you can also completely transpose your data (-xpose-), or stack lists of variables in new ones (-stack-).

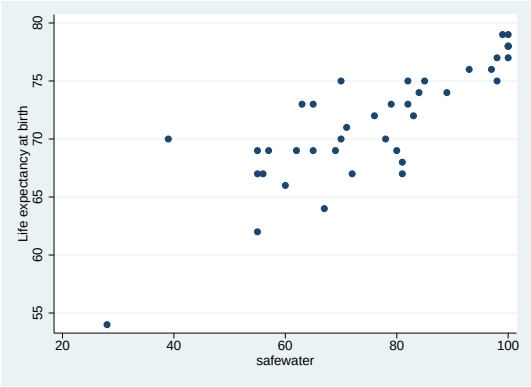
9 Plotting data

Stata's user interface has an elaborate Menu which you should ignore with one exception: Graphics. Through the Menu you can access dialogs that will assist you in creating graphs. When you use the dialogs to construct a graph its syntax will also be printed in the Results window. This can be very convenient because the syntax of graphics is rather intricate and constructing non-standard graphs is not always straightforward.

This being said, you do not need an advanced degree to create simple plots to look at your data. Using the system example dataset from above, the following uses -scatter- to create a scatter plot of the life expectancy vs. access to safe water:

```
. sysuse lifeexp, clear
. scatter lifeexp safewater
```

which produces a plot like this:

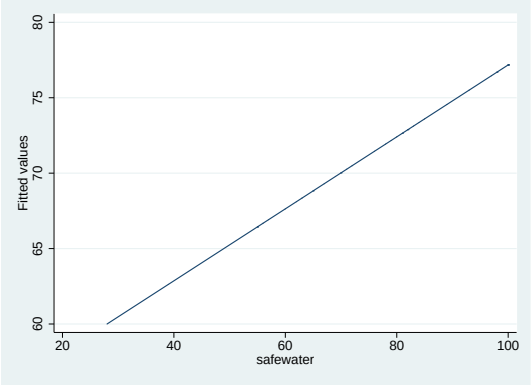


note that the syntax is such that the last variable is the x-variable.

Line plots are created in a similar fashion. To illustrate let's -regress- lifeexp on safewater, create the predicted value using -predict-, and plot the result with -line-:

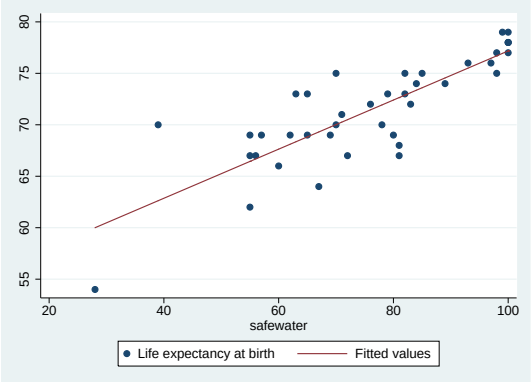
```
. reg lifeexp safewater
(output omitted)
. predict py
. line py safewater
```

which results in the following plot:



Both -scatter- and -line- fall in the class of -twoway- plots and are shorthand for -twoway scatter- and -twoway line-. It is easy to overlay plots:

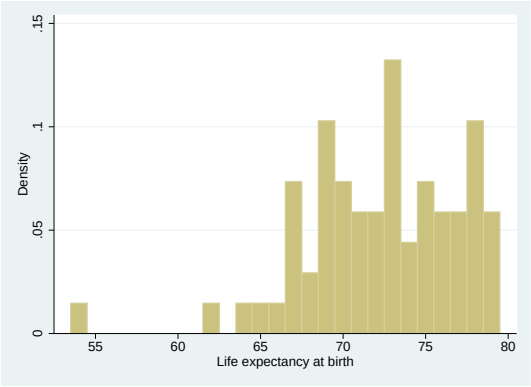
```
. twoway (sc lifeexp safewater) || (line py safewater)
```



To plot the distribution of a variable you can use -histogram-, this for example

```
. hist lifeexp, discrete
```

will result in the following graph



The option 'discrete' tells Stata that *lifeexp* is a discrete variable so that -histogram- will not bin the data.

Finally, to export a graph to disk you can use -graph export-

```
. graph export plot.eps
```

where the extension is an easy way to specify the file format. The example here exports to an encapsulated postscript file (eps).

Stata's graphing capabilities are very rich and there is nothing you cannot tweak, see -graph- for much more, and as mentioned above: use the dialogs to learn Stata's graphing syntax.

10 Organizing yourself

Although the command line is a great way to explore your data, it is absolutely essential that you organize your analyses in do-files. Remember, these are plain text files with the extension .do. You can execute these files from the command line like this

```
. do filename
```

where it is not necessary to type the .do extension.

To ensure that your old do-files will run under future versions of Stata is suffices to put

```
version #
```

at the start of your do-file. This sets Stata's command interpreter version to #, where you should substitute # with your current version of Stata which you can check with -version-.

It is also a good habit to put comments in your Stata code at points where you think that some explanation might help others – or yourself months later – understand why you did what you did.

You can begin comments with // or ///, or put them between /* and */. The latter can be put anywhere, span multiple lines and this is therefore a good way to comment out large chunks of code. Both // and /// are single line comments, they need to be preceded by at least one space and everything after them is commented out. The difference between // and ///, is that with /// the next line will be joined to the current one when executing the do-file (while ignoring the comment). It is therefore a good way to break long lines over several ones.

It is advisable to give meaningful names to your do-files. One convention uses crxxx.do for a do-file that creates the dataset xxx.dta, gryyy for the do-file that creates the graph yyy, and anzzz.do for the do-file that does analysis zzz. Do-files can be called from do-files, so finally you can create a do-file main.do which calls all do-files in your project. This way you can always reproduce your results later simply by running main.do.

When running a do-file with -do- the output is printed in Stata's Results window. To write the results to disk you can use -log-. The following opens a plain text log-file (extension .log) and will overwrite an existing file with the same name

```
. log using filename, text replace
```

everything which follows this statement will be written to the log file. This

```
. log close
```

closes the log file. When a log file is open you can switch logging on and off with -`log on`- and -`log off`-.

Another good habit is not only to document your Stata code, but also your data. Give your variables short meaningful names in *only lowercase* letters. This reduces the amount of typing on the command line, and makes your code and datasets more understandable. You can also attach labels and notes to your data. For example variables are labelled like this:

```
. label var lifeexp "Life expectancy at birth"
```

or define value labels and attach them to a variable:

```
. label define region 1 "Europe/Central Asia" 2 "N. America" 3 "S. America"
. label value region region
```

See -label- for more details. Finally you can attach longer -notes- to your data or variables.

11 Programming Stata

At one point you might find yourself in the position that you want to automate a repetitive but systematic task (are you copy-pasting pieces of code and changing them a bit?), or reuse your code in other places. For this you will need to learn a bit of Stata programming.

The first thing you should understand is what Stata calls "macros". A -macro- is nothing more than a named piece of text that you can access and modify in your code. Macros can be local or global. The scope of local macros is limited to the program or do-file where they were defined. Global macros are visible to all programs and do-files. Although global macros are sometimes handy they are EVIL (because of namespace clashes, and propagating bad programming style) and you should avoid them.

The syntax for local macro definition is

```
local name [=] expression
```

For example

```
. local a 1 two three
```

defines a local macro called *a* that contains the text "1 two three".

To access (dereference) a local macro you need to enclose the name in a single left quote (`) and single right quote ('). Stata will replace all occurrences of local

macros with their values. To illustrate how this work we print the value of the local macro with -display-

```
. di "`a'"
1 two three
```

the (dereferenced) macro is put between double quotes to ensure the result is a string. Otherwise -display- will evaluate the expresson '1 two three' and complain that it cannot find the variable/scalar 'two' (and 'three').

You should not use the equals sign (=) when storing an expression in a macro *unless* you want the expression to be evaluated. To appreciate the difference consider the following example.

```
. // example 1
. local a 1
. local a `a' + 1
. di `a'
2
. di "`a'"
1 + 1

. // example 2
. local a 1
. local a = `a' + 1
. di `a'
2
. di "`a'"
2
```

In the first example the number 1 is stored in macro *a*. In the second -local- statement the contents of macro *a* is substituted in the expression which is then stored in macro *a*. The first display statement prints out 2. Is "2" the literal content of macro *a*? No, the second -display- statement shows that it is "1+1", this is passto and evaluated by -display- which then prints out the result: 2.

Example 2 shows what happens when we use the equals sign in the local statement: The expression will *first* be evaluated and then stored in the macro! As a consequence both -display- statements print out the same result. It is therefore important to be aware when and where expressions are evaluated.

The seemingly simple concept of these 'macros' is in practice very powerful. We can use them for example in loops to create indicator variables:

```
forvalues r=1/3 {
        g reg`r' = region==`r'
}
```

-forvalues- loops over a range of values (in this case 1, 2, 3) and on each iteration the command(s) between braces get executed after the value of the local macro is substituted. In the above example this implies that Stata first executes

```
g reg1 = region==1
```

then

```
g reg2 = region==2
```

etc. See -foreach- for other ways to loop over lists of items such as variables, arbitrary strings or numbers.

In expressions that are evaluated, macros behave in many ways like conventional single (scalar) string/numerical variables. One important difference arises from the fact that their contents get substituted everywhere, not only in expressions. This means that macros can be many things at the same time: numbers, string, parts of variable names, etc. In the end the important thing to remember is that

you can use local macros *everywhere* and *in any way* as long as the result after macro substitution is a valid Stata command.

To create and manipulate macros there are extended macro functions. These allow you for example extract a word from a macro, or count the number of words it contains. There are also functions which store the variable or value labels in a local, or gets a list of files that match a given file pattern in a directory.

Macros are also often used to contruct and manipulate lists. The command -levelsof- for example stores all unique values of a variable in a local macro. There are also extended functions to combine and such lists.

Stata also has scalars which can store both numbers and strings. Their common use is to store numerical results which is done in double precision. Scalars behave like conventional (scalar) variables and many commands, such as -summarize-, return results in the form of scalars:

```
. sum lexp if region==1

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        lexp |        44    73.06818    4.150639         65         79

. return list
scalars:
                  r(N) =  44
              r(sum_w) =  44
               r(mean) =  73.06818181818181
                r(Var) =  17.22780126849894
                 r(sd) =  4.150638657905424
               r(min) =  65
               r(max) =  79
               r(sum) =  3215

. di r(mean)*r(N)
3215
```

Stata commands return their results can not only return scalars, but also macros, and matrices. These can all be used in calculations or stored. After you run a command you can get the list of returned objects using -return list- or -ereturn list-. Alternatively you can look it up in the help file or the documentation.

Finally you can use -program- to write programs that you can use in the same way as you use regular Stata commands. The following introduces the key concept.

```
1 program foo
2 di "All: `*'"
3 local n : word count `*'
4 forv i=1/`n' {
5         di "`i': ``i''"
6 }
7 end
```

The first line starts the program defininition and names the program (foo). All code in between -`program name`- and -`end`- (on the last line) will define your program. Like in any other programming language you can pass arguments to your program. Let's see what happens when we do that:

```
. foo the quick brown fox jumped
All: the quick brown fox jumped
1: the
2: quick
3: brown
4: fox
5: jumped
```

First note that we called this program in the same way as we would have called a regular Stata program. Now let's step through the program line by line. As can be seen from the output of line 2, Stata stores all the arguments in a local macro named *. Stata also splits up (-tokenize-) the complete argument into separate tokens based on the spaces. The first argument is stored in the macro 1, the second in macro 2, etc.

Line 3 uses an extended macro function to count the number of arguments, and line 4 defines a loop to print out each token separately using -display-.

The outputting is done in line 5 which merits some attention because of the way it uses macros. The first occurance of *i*, in single quotes, prints out its name (number) whereas the second, in double single quotes, prints out the contents. Now remember what Stata does: it (repeatedly) replaces all macros with their contents. Consider the first iteration of the loop (i=1) when Stata interprets the following command:

```
di "`i': ``i''"
```

it will substitute the value of macro *i* resulting in

```
di "1: `1'"
```

but there is still an unsubstituted macro left, and Stata therefore does another round of substitution resulting in

```
di "1: the"
```

which is then printed. This happens on every iteration of the -forvalues- loop resulting in the print-out above.

We have now seen how to define a Stata program and access the arguments. Let's consider a sligthly more useful program that should called after -regress-.

The program will print out the coefficient coefficient of an explanatory variable – the name of which will be the argument passed to the program – followed by the standard error enclosed in parentheses and stars to indicate the level of statistical significance.

```
 1  capture program drop _res
 2  program _res
 3  args xvar
 4          di %9.3f _b[`xvar'] _c
 5          di "   (" trim(string(_se[`xvar'], "%9.3f")) ")" _c
 6          scalar pval = 2*ttail(e(df_r), abs(_b[`xvar']/_se[`xvar']))
 7          if (pval<0.01) di "***"
 8          else if (pval<0.05) di "** "
 9          else if (pval<0.1) di "*  "
10          else di "   "
11  end
```

The first line drops the program from memory. This is a common thing to do when you define programs in your do-files just to avoid that execution will stop: Stata will not allow you to define a program with a name that already exists and throw an error. At the same time -program drop- will throw an error if a program with this name is not defined. The preceding -capture- will "swallow" the error and execution will continue.

As above line 2 starts the program definition. Line 3 uses -args- which does nothing more than give the tokenized arguments 1, 2, 3, etc more meaningful names to make the code more readable. This program uses one argument, namely the name of one of the regressors, which is now called xvar.

Line 4 prints the regression coefficient which can be accessed with _b[varname] preceded by the format string %9.3f (-format-). This prints out the coefficient on 9

fixed positions and 3 decimals. The '_c' at the end tells -display- not to print a line break (and continue). Line 5 prints the formatted standard error _se[varname] in parentheses.

The only thing left to do is to print out the significance stars. In a first step, line 6 calculates the p-value and stores it in a scalar. This is done by calculating the t-statistic, getting the residual degrees of freedom returned by -regress- in e(df_r), and plugging these in the function ttail() which calculates Pr(T>t). Finally lines 7-10 use -ifcmd- to determine how many stars to print out based on the p-value.

The following illustrates what the program does where _col(12) tells -display- to continue on column (position) 12, and -quietly- supresses all regression output on the terminal.

```
foreach y of varlist lexp popgrowth gnppc {
        di "`y'" _col(12) _c
        quietly reg `y' safewater
        _res safewater
}

lexp            0.239   (0.026)***
popgrowth      -0.022   (0.008)***
gnppc         406.353   (68.844)***
```

as you can see we now have nicely formatted output from three different regressions.

12 Where to go from here...

As mentioned from the outset, many topics are not covered in this document which concentrates on presenting basic data management and programming skills.

Estimation and testing for example is beyond the scope of this presentation. This includes generic maximum likelihood estimation (-ml-), boostrapping (-bootstrap-), monte carlo simulation (-simulate-) and permutation (-permute-). One way to get started on these and many more subjects is by browsing the overview in the help files, and of course -findit-.

Then there is the possibility to write full-fledged Stata commands yourself. A Stata command is essentially a -program- in a plain text file with the extension .ado. If it is on your adopath you can call it from any do-file. If you want to write ado-files you should learn things like how to parse standard Stata syntax (-syntax-), mark your sample (-mark-), make your program by-able (-byprog-), format output (-displaying_output-), write help files, return results (-return-), debug your programs (-contents_debugging-). See help -help contents_programming - for much more.

Finally, matrices were not discussed (-help contents_matrices-). Stata comes with a full fledged matrix programming language called Mata (-help mata-). Stata code is interpreted and can therefore be slow, especially when doing things like looping over observations. Mata however is compiled by Stata and is therefore much faster. This also means your code is platform-independent. A Mata routine written under Windows will run without changes on a Linux box (or any other platform supported by Stata). Mata has a C-like syntax and its matrix language is comparable to programs like Matlab (porting existing code is usually straightforward). It also features optimization routines, and I/O and text processing functions. At the same time Mata is tightly integrated with Stata giving you the advantage of having access to the same data management and statistical procedures you have access to on the command line. Finally, you can benefit from Mata's speed and call Mata routines in your do- and ado-files.

Before plunging into these more advanced techniques the first thing to do however is to put the ones discussed in this document into practice. The only way to learn how to program is by dirtying your hands.