Advanced Stata Topics

CEP and STICERD
London School of Economics
Lent Term 2009

Alexander C. Lembcke

eMail: a.c.lembcke@lse.ac.uk
Homepage: http://personal.lse.ac.uk/lembcke

This is an updated version of Michal McMahon's Stata notes. He taught this course at the Bank of England (2008) and at the LSE (2006, 2007). It builds on earlier courses given by Martin Stewart (2004) and Holger Breinlich (2005). Any errors are my sole responsibility.

Full Table of contents

PROGRAMMING **4**

GOOD PROGRAMMING PRACTICE 4
PROGRAMMING BASICS 5
Macros *5*
Macro contents *8*
Text *8*
Statements *8*
Numbers and expressions *9*
Manipulation of macros *9*
Deferred macro evaluation and advanced macro usage *10*
Temporary objects *10*
Looping *11*
for *11*
foreach and forvalues *12*
Combining loops and macros *14*
Repeating commands using Stata's inbuilt functions. *15*
Branching *17*
WRITING STATA PROGRAMS 19
Creating or "defining" a program *19*
Macro shift (number of loops is variable) *20*
Naming a program *21*
Debugging a program *21*
Program arguments *22*
Renaming arguments *22*
Programs with return values and other options *24*
Help files and publishing programs *28*
MAXIMUM LIKELIHOOD METHODS 31
Maximization theory *31*
Creating the first ml estimation *32*
Specifying the gradient and Hessian by hand *34*
Extension to non-standard estimation *37*
Utilities to check our estimation *39*
Flexible functional forms and constraints *42*
Further reading *43*
MATA 44
What is Mata and why bother? *44*
Mata basics *44*
Object types *48*
Precision issues ..... ***Error! Bookmark not defined.***
Stata data in Mata *49*
Sorting and permutations *50*
Mata functions *53*
Returning values ..... ***Error! Bookmark not defined.***
Looping and branching *55*
Using structures or pointers instead of macro tricks *56*
Mata's optimize command *57*
Some programs ..... ***Error! Bookmark not defined.***

Course Outline

This course is run over 8 weeks during this time it is not possible to cover everything – it never is with a program as large and as flexible as Stata. Therefore, I shall endeavor to take you from a position of complete novice (some having never seen the program before), to a position from which you are confident users who, through practice, can become intermediate and onto expert users.

In order to help you, the course is based around practical examples – these examples use macro data but have no economic meaning to them. They are simply there to show you how the program works. There will be some optional exercises, for which data is provided on my website – http://personal.lse.ac.uk/lembcke. These are to be completed in your own time, there should be some time at the end of each meeting where you can play around with Stata yourself and ask specific questions.

The course will follow the layout of this handout and the plan is to cover the following topics.

Week	**Time/Place**	**Activity**
Week 1	Tue, 17:30 – 19:30 (S169)	Getting started with Stata
Week 2	Tue, 17:30 – 19:30 (S169)	Database Manipulation and graphs
Week 3	Tue, 17:30 – 19:30 (S169)	More database manipulation, regression and post-regression analysis
Week 4	Tue, 17:30 – 19:30 (S169)	Advanced estimation methods in Stata
Week 5	Tue, 17:30 – 19:30 (S169)	Programming basics in Stata
Week 6	Tue, 17:30 – 19:30 (S169)	Writing Stata programs
Week 7	Tue, 17:30 – 19:30 (S169)	Maximum Likelihood Methods in Stata
Week 8	Tue, 17:30 – 19:30 (S169)	Mata

I am very flexible about the actual classes, and I am happy to move at the pace desired by the participants. But if there is anything specific that you wish you to ask me, or material that you would like to see covered in greater detail, I am happy to accommodate these requests.

Programming

Programming in general refers to the writing of a computer program. What we will do in the next chapters is far more basic than writing a whole program. In the end you should be able to extend Stata by writing own subroutines, such as estimation or post-estimation commands. Subsequently we will introduce some basic programming skills (looping and branching), Stata specific commands that make writing extensions easier and finally Mata the matrix language of Stata. The chapter concludes with command for running maximum likelihood estimation.

This introduction into programming cannot cover all of Stata's capabilities. One important and noteworthy omission is programming of Stata plug-ins. Plug-ins can be written in C (a "classical" programming languages) and compiled to become part of Stata's core. These plug-ins can be faster than extensions of the type we introduce here, but writing these plug-ins involves learning a different programming language and they are platform specific, i.e. you cannot simply transfer the compiled code from Windows to Unix/Linux.

Good programming practice

Before we turn to programming itself we should consider a few simple guidelines which will help to make our code more accessible, easier to maintain and compatible with other commands. Entire books are dedicated to the topic of good programming practice. Here we will only consider a few guidelines that one should always follow. Some more specific aspects about good programming will be mentioned in the following sections. In general good programming practice has a two-fold purpose. For one it is supposed to make your code as easy to understand as possible, and two it should be as self-explanatory as possible. Even if you never intend to share your code with another person, you will at some point revisit your old code and good programming practice will help you minimize the time you need to get back into your code.

- Use comments! This point cannot be emphasized enough. Make general comments on what the code is supposed to do, what purpose it was written for, when it was written, what inputs you need and what output it will produce. Make specific comments on parts of your code, what does this routine achieve, why did you include a certain piece of code. It is also very helpful to mark the start and the end of large chunks of code explicitly, e.g. "loop xyz starts here", "loop xyz ends here".
- When using names in your code try to be explicit, when generating variables, label them. Prefixes can be a huge help in avoiding confusion and ambiguities. For example name temporary variables with the prefix "tmp_", scalars with "s_" etc. You can also use upper and lower case letters for different objects (remember Stata is case sensitive).
- Make it easy to assess what belongs together. Indent blocks of code that form an entity. Use spaces in mathematical formulas to separate the operators from variables and scalars. And when you write long lines of code, wrap them by using comments or special end-of-line delimiters. To change the standard end-of-line delimiter (i.e. a new line in our code) we can use `#delimit ;` or `#d;` for short. Now Stata will consider all code to belong to the same line until it encounters a semicolon. To go back to the standard (new line new command) delimiter we use `#d cr`.
- Repetitive tasks should be completed by functions instead of copy and pasting your code. Try to write your functions as general as possible so you can reuse them for all your projects. I have an ever-expanding file with useful programs from which I use one or two in each of my projects.
- When you write your code, test it. Most of programming is actually debugging (finding the source of errors and correcting them) and no code works perfectly from scratch. When testing your code think about nonstandard situations as well, what happens with your code if there are missing values, what influence does the storage type of a variable have, will your code work on every machine or is it designed for your specific setup, etc.
- When writing programs there is usually a trade-off between memory usage and calculation speed which is hard to avoid. In general we can either save intermediate results, thereby avoiding rerunning calculations over and over again or we can save memory by not doing so.
- Accessing the hard drive is slow, saving data or using `preserve` will do so. So in general you should avoid using these commands. On the other hand using `if` and `in` can be quite time consuming as well, so if you run several commands on the same subset, using `preserve` and dropping the unnecessary observations can save time.
- Avoid creating and recreating variables, especially if you do not need them, they clutter you memory and take time to be created.
- While we cannot do much about trade-offs we can avoid some inefficiencies. One thing you should never ever do is to loop over observations, try subsetting the data either in Stata or Mata. Inverting matrices is necessary for a lot of estimators, but inverting a matrix is computationally cumbersome and thanks to rounding usually rather inaccurate task. Instead of using the inverse you should use a linear equation solver (we will see this in the Mata section).
- When you use matrices you should define them as a whole and not let them "grow" by adding columns or rows.
- User written programs that extend Stata fulfill one of three purposes: report results, create a variable or modify the data. Try to avoid writing programs that do several of these tasks.

Programming basics

One very important issue that can result in problems in Stata (and in any other statistical software) is that data is stored digitally. The data is stored in binary format which means that numbers that are perfectly fine in the base 10 system (e.g. 1.1) will have infinitely many digits in binary format (for 1.1 the binary equivalent is 1.00011001100…). Since infinitely many digits cannot be stored, the software makes a cut at some point. Where this cut is made depends on the format you store your data in (i.e. float or double). Standard when you generate a variable is the "float" format, which uses less memory than the "double" format. But Stata works internally with highest precision (and in fact it is good practice that you write your programs using the highest precision as well). So what is the problem? If you generate a variable with "float" precision but use a scalar with "double" precision in conjunction with the values of this variable, you might not get the desired results:

```
. gen test = 1.1
. li if test == 1.1
```

Stata will list no observations. There are two ways to deal with this problem, either you change the format of the variable or you change the format of the scalar.

```
. gen double test = 1.1
. li if test == 1.1
. drop test
. gen test = 1.1
. li if test == float(1.1)
```

If you write programs just for your own use you can also use

```
set type double, permanently
```

to make the standard format of newly generated variables not "float" but "double". But beware, a variable in "double" format uses twice as much memory as variable in "float" format.

Macros

A Stata macro is different to an Excel macro. In Excel, a macro is like a recording of repeated actions which is then stored as a mini-program that can be easily run – this is what a do file is in Stata. Macros in Stata are the equivalent of variables in other programming languages. A macro is used as shorthand – you type a short macro name but are actually referring to some numerical value or a string of characters. For example, you may use the same list of independent variables in several regressions and want to avoid retyping the list several times. Just assign this list to a macro. Using the PWT dataset:

```
. local varlist gdp60 openk kc kg ki
. regress grgdpch `varlist' if year==1990

      Source |       SS       df       MS              Number of obs =     111
-------------+------------------------------           F(  5,   105) =    6.70
       Model |  694.520607     5  138.904121           Prob > F      =  0.0000
    Residual |  2175.67123   105  20.7206784           R-squared     =  0.2420
-------------+------------------------------           Adj R-squared =  0.2059
       Total |  2870.19184   110  26.0926531           Root MSE      =   4.552

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -1.853244   .6078333    -3.05   0.003    -3.058465   -.6480229
       openk |  -.0033326   .0104782    -0.32   0.751    -.0241088    .0174437
          kc |  -.0823043   .0356628    -2.31   0.023     -.153017   -.0115916
          kg |  -.0712923   .0462435    -1.54   0.126    -.1629847    .0204001
          ki |   .2327257   .0651346     3.57   0.001     .1035758    .3618757
       _cons |   16.31192   5.851553     2.79   0.006     4.709367    27.91447
------------------------------------------------------------------------------
```

Macros are of two types – local and global. Local macros are "private" – they will only work within the program or do-file in which they are created. Thus, for example, if you are using several programs within a single do-file, using local macros for each means that you need not worry about whether some other program has been using local macros with the same names – one program can use `varlist` to refer to one set of variables, while another program uses `varlist` to refer to a completely different set of variables. Global macros on the other hand are "public" – they will work in all programs and do files – `varlist`

refers to exactly the same list of variables irrespective of the program that uses it. Each type of macro has its uses, although local macros are the most commonly used type and you should in general stick with them.

Just to illustrate this, let's work with an example. The program `reg1` will create a local macro called `varlist` and will also use that macro. The program `reg2` will not create any macro, but will try to use a macro called `varlist`. Although `reg1` has a macro by that name, it is local or private to it, so `reg2` cannot use it:

```
. program define reg1
  1. local varlist gdp60 openk kc kg ki
  2. reg  grgdpch `varlist' if year==1990
  3. end

. reg1

      Source |       SS       df       MS              Number of obs =     111
-------------+------------------------------           F(  5,   105) =    6.70
       Model |  694.520607     5  138.904121           Prob > F      =  0.0000
    Residual |  2175.67123   105  20.7206784           R-squared     =  0.2420
-------------+------------------------------           Adj R-squared =  0.2059
       Total |  2870.19184   110  26.0926531           Root MSE      =   4.552

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -1.853244   .6078333    -3.05   0.003    -3.058465   -.6480229
       openk |  -.0033326   .0104782    -0.32   0.751    -.0241088    .0174437
          kc |  -.0823043   .0356628    -2.31   0.023     -.153017   -.0115916
          kg |  -.0712923   .0462435    -1.54   0.126    -.1629847    .0204001
          ki |   .2327257   .0651346     3.57   0.001     .1035758    .3618757
       _cons |   16.31192   5.851553     2.79   0.006     4.709367    27.91447
------------------------------------------------------------------------------

. capture program drop reg2
. program define reg2
  1. reg  grgdpch `varlist' if year==1990
  2. end

. reg2
      Source |       SS       df       MS              Number of obs =     129
-------------+------------------------------           F(  0,   128) =    0.00
       Model |           0     0           .           Prob > F      =       .
    Residual |  4008.61956   128  31.3173404           R-squared     =  0.0000
-------------+------------------------------           Adj R-squared =  0.0000
       Total |  4008.61956   128  31.3173404           Root MSE      =  5.5962

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |   .9033816    .492717     1.83   0.069    -.0715433    1.878306
------------------------------------------------------------------------------
```

Now, suppose we create a global macro called `varlist` – it will be accessible to all programs. Note, local macros are enclosed in the special quotes (`` `' ``), global macros are prefixed by the dollar sign (`$`). In some cases it is important to denote the start and the end of the name of a global explicitly. When this is the case you enclose the name of the global in curly brackets. It is not necessary to use the curly brackets in the following example, but we can use them nonetheless.

```
. global varlist gdp60 openk kc kg ki

. capture program drop reg1
. program define reg1
  1. local varlist gdp60 openk kc kg ki
  2. reg grgdpch `varlist'
  3. reg grgdpch $varlist
  4. end
```

```
. capture program drop reg2
. program define reg2
  1. reg grgdpch ${varlist}
  2. reg grgdpch `varlist'
  3. end

. reg1
      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------

      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------
. reg2

      Source |       SS       df       MS              Number of obs =    5067
-------------+------------------------------           F(  5,  5061) =   41.21
       Model |  8692.09731     5  1738.41946           Prob > F      =  0.0000
    Residual |  213498.605  5061  42.1850632           R-squared     =  0.0391
-------------+------------------------------           Adj R-squared =  0.0382
       Total |  222190.702  5066   43.859199           Root MSE      =   6.495

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       gdp60 |  -.5393328   .1268537    -4.25   0.000    -.7880209   -.2906447
       openk |  -.0003768   .0020639    -0.18   0.855    -.0044229    .0036693
          kc |  -.0249966   .0055462    -4.51   0.000    -.0358694   -.0141237
          kg |  -.0454862   .0089808    -5.06   0.000    -.0630924     -.02788
          ki |   .1182029    .011505    10.27   0.000     .0956481    .1407578
       _cons |   6.344897   1.045222     6.07   0.000     4.295809    8.393985
------------------------------------------------------------------------------

      Source |       SS       df       MS              Number of obs =    5621
-------------+------------------------------           F(  0,  5620) =    0.00
       Model |           0     0           .           Prob > F      =       .
    Residual |  250604.237  5620  44.5915012           R-squared     =  0.0000
```

```
-------------+------------------------------           Adj R-squared =  0.0000
       Total |  250604.237  5620  44.5915012           Root MSE      =  6.6777

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |   2.069907   .0890675    23.24   0.000       1.8953    2.244513
------------------------------------------------------------------------------
```

As you can see, Stata runs two fully specified regressions in the first case but only one in the last case since again, the program reg2 does not recognize `` `varlist' ``.

You should refrain from using global macros when a local macro suffices. This is good programming practice as it forces you to explicitly pass arguments between functions, commands and programs instead of defining them in some hard-to-find place in your code. If you use global macros you should make sure that you define them at the beginning of your code and only pass constants, that is you do not change the contents somewhere in your code.

Macro contents

We introduced macros by showing how they can be used as shorthand for a list of variables. In fact, macros can contain practically anything you want – variable names, specific values, strings of text, command names, if statements, and so on. Some examples of what macros can contain are:

Text

Text is usually contained in double quotes (`"`) though this is not necessary for macro definitions:

```
. local ctyname "United States"
```

gives the same result as

```
. local ctyname United States
```

A problem arises whenever your macro name follows a backslash (\). Whenever this happens, Stata fails to properly load the macro (more on this later in this section):

```
. local filename PWT.dta
. use "H:\ECStata\`filename'"

file H:\ECStata`filename'.dta not found

r(601);
```

To get around this problem, use double backslashes (\\) instead of a single one or slashes (/) instead of backslashes:

```
. use "H:\ECStata\\`filename'"
. use "H:/ECStata/`filename'"
```

I prefer using the normal slashes (/) since forward slashes are recognized as folder separators on all operating systems, backslashes are Windows specific.

Statements

Using macros to contain statements is essentially an extension of using macros to contain text. For example, if we define the local macro:

```
. local year90 "if year==1990"
```

then,

```
. reg grgdpch $varlist `year90'
```

is the same as:

```
. reg grgdpch gdp60 openk kc kg ki if year==1990
```

Note that when using `if` statements, double quotes become important again. For simplicity, consider running a regression for all countries whose codes start with "B". First, I define a local macro and then use it in the `reg` command:

```
. local ctyname B
. reg grgdpch gdp60 openk kc kg ki if substr(country,1,1)=="`ctyname'"
```

Although it does not matter whether I define `ctyname` using double quotes or not, it is important to include them in the if-statement since the variable `country` is string. The best way to think about this is to do what Stata does: replace `` `ctyname' `` by its content. Thus, `substr(country,1,1)=="`ctyname'"` becomes `substr(country,1,1)=="B"`. Omitting the double quotes would yield `substr(country,1,1)==B` which as usual results in an error message (since the results of the `substr`-operation is a string).

Numbers and expressions

```
. local i = 1
. local result = 2+2
```

Note, when the macro contains explicitly defined numbers or equations, an equality sign must be used. Furthermore, there must be no double-quotes, otherwise Stata will interpret the macro contents as text:

```
. local problem="2+2"
```

Thus, the `problem` macro contains the text `2+2` and the `result` macro contains the number `4`. Note that as before we could also have assigned "2+2" to `problem` while omitting the equality sign. The difference between the two assignments is that assignments using "=" are evaluations, those without "=" are copy operations. That is, in the latter case, Stata simply copies "2+2" into the macro `problem` while in the former case it evaluates the expression behind the "=" and then assigns it to the corresponding macro. In the case of strings these two ways turn out to be equivalent. There is one subtle difference though: evaluations are limited to string lengths of 244 characters (80 in Intercooled Stata) while copy operations are de facto only limited by available memory. Thus, it is usually safer to omit the equality sign to avoid parts of the macro being secretly cut off (which can lead to very high levels of confusion…)

While a macro can contain numbers, it is essentially holding a string of text that can be converted back and forth into numbers whenever calculations are necessary. For this reason, macros containing numbers are only accurate up to 13 digits. When precise accuracy is crucial, scalars should be used instead:

```
. scalar root2=sqrt(2)
. display root2
1.4142136
```

Note, when you call upon a macro, it must be contained in special quotes (e.g. `` display `result' ``), but this is not so when you call upon a scalar (e.g. `display root2` and not `` display `root2' ``).

An important special case for a mathematical expression is the shorthand notation that increments or decrements a number stored in a local macro by one (this only works for locals).

```
. local i = 1
. local ++i
. di `i'
2
. local --i
. di `i'
1
```

Manipulation of macros

Contents of macros can be changed by simply redefining a macro. For example, if the global macro `result` contains the value "2+2" typing:

```
. global result "2+3"
```

overwrites its contents. If you want to drop a specific macro, use the `macro drop` command:

```
. macro drop year90
```

To drop all macros in memory, use `_all` instead of specific macro names. If you want to list all macros Stata has saved in memory instead (including a number of pre-defined macros), type:

```
. macro list
```

or

```
. macro dir
```

Macro names starting with an underscore ("_") are local macros, the others are global macros. Similarly, to drop or list scalars, use the commands `scalar drop` and `scalar list` (or `scalar dir`) respectively.

Deferred macro evaluation and advanced macro usage

We saw earlier that macros following a backslash (\) are not evaluated as expected. This is not an error but in fact a very useful programming tool. What the backslash does is to defer evaluation of the macro until it is called. In other words, instead of plugging the content of a macro in, the name of the macro is plugged in. How can you use this cleverly?

Assume that you want to run regressions on a set of covariates where one part of that set is fixed and another part varies. Let's start with the case where you introduce additional regressors after running a regression on a base specification.

```
. local rhs "gdp60 openk"
. reg grgdpch `rhs' if year == 1990
. local rhs "`rhs' kc kg ki"
. reg grgdpch `rhs' if year == 1990
```

Notice the self-referential use of the local `rhs` in the third line. We plug in the current value of `rhs`, i.e. the string "`gdp60 openk`" and add "` kc kg ki`" afterwards. We could do the same with a global macro:

```
. global rhs "$rhs kc kg ki"
```

Now if we want to evaluate the effect of `kc kg ki` separately we cannot simply add them one after another. One solution is to use

```
. local rhs "gdp60 openk"
. local new_rhs "`rhs' kc"
. reg grgdpch `new_rhs' if year == 1990
. local new_rhs "`rhs' kg"
...
```

An elegant alternative would be to defer the evaluation of the macro. In the following we will use a local, the same principle applies for global macros as well.

```
. local rhs "gdp60 openk \`add_var'"
. local add_var "kc"
. reg grgdpch `rhs' if year == 1990
. local add_var "kg"
...
```

What happens when we refer to a macro is that its value is plugged in at that point. Using a backslash instead results in the reference of the macro being plugged in, i.e. not the value of `add_var` but the term `` `add_var' `` will be substituted. So each time we call upon the local `rhs` the current value of the local `add_var` is substituted in.

Since evaluating a macro is equivalent to plugging in its value directly we can also use it in the definition of macros.

```
. local post "1990"
. local pre "year"
. local `pre'`post' = 1990
. di "Year: ``pre'`post''"
```

In the last line we nest macros within each other. First the inner values are evaluated, i.e. `year` and `1990` are plugged in and then the outer apostrophes lead to an evaluation of `year1990` which has the value `1990`. Uses of this will become clearer when we turn to looping in the next section.

Temporary objects

Another important feature of Stata is that it can also store information in so-called temporary objects which are often used in longer programs. The advantage is that temporary objects vanish once they are no longer of use (so you don't need to drop them by hand) and you won't run into problems using names for variables or other objects that are already in use.

- `tempvar` defines a temporary variable and saves the name of that variable in a local macro (we have already seen this type earlier on). The temporary variable can be referred to using the contents of the corresponding local. When the program or do-

file concludes, any variables with these assigned names are dropped:

```
. program define temporary
  1. tempvar logpop
  2. gen `logpop'=log(pop)
  3. sum pop if `logpop'>=8
  4. end

.
. temporary
(2721 missing values generated)

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
         pop |      4162    43433.71    126246.4       2989    1258821
```

Since the tempvar logcgdp is dropped at the end of the program, trying to access it later on yields an error message:

```
. sum pop if `logpop'>=8
>8 invalid name
r(198);
```

- `tempname` defines a name for a temporary object (e.g. a matrix or scalar) and stores this name in a local. When the program or do-file concludes, any scalars or matrices with these assigned names are dropped. This command is used more rarely then `tempvar` but can be useful if you want to do matrix-algebra in Stata subroutines.
- `tempfile` defines a temporary data file to be saved on the hard drive. Again the name is stored in a local macro. When the program or do-file concludes, any datasets created with these assigned names are erased. For example, try the following program:

```
. program define temporary2
  1.   tempfile cgdp
  2.   keep country year cgdp
  3.   save "`cgdp'"
  4.   clear
  5.   use "`cgdp'"
  6.   sum year
  7. end

. temporary2
file C:\DOCUME~1\Michael\LOCALS~1\Temp\ST_0c000012.tmp saved

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        year |      8568        1975    14.72046       1950       2000
.
```

This saves the variables `country year cgdp` in a temporary file that is automatically erased as soon as the program terminates (check this by trying to reload "`cgdp'" after termination of the program "temporary"). This is most useful when used in combination with `merge`.

Looping

There are a number of techniques for looping or repeating commands within your do-file, thus saving you laborious retyping or cutting and pasting. These techniques are not always mutually exclusive – you can often use one or more different techniques to achieve the same goal. However, it is usually the case that one technique is more suitable or more efficient in a given instance than the others. Therefore, it is best to learn about each one and then choose whichever is most suitable when you come across a looping situation.

for

While the `for` command is outdated in Stata 10 it is still useful in some settings. for-processing allows you to easily repeat Stata commands. As an example, we can use the PWT dataset and create the mean of several variables all at once:

```
. for varlist kc ki kg: egen mean_X=mean(X)
->  egen mean_kc=mean(kc)
```

```
->  egen mean_ki=mean(ki)
->  egen mean_kg=mean(kg)
```

The `egen` command is repeated for every variable in the specified `varlist`, with the X standing in for the relevant variable each time. You can see in the variables window that our three desired variables have been created.

```
. for varlist kc ki kg: display "Mean of X = " mean_X

->  display `"Mean of kc = "' mean_kc
Mean of kc = 72.536438

->  display `"Mean of ki = "' mean_ki
Mean of ki = 15.740885

->  display `"Mean of kg = "' mean_kg
Mean of kg = 20.606308
```

The onscreen display includes both the individual commands and their results. To suppress the display of the individual commands, use the `noheader` option:

```
for varlist kc ki kg, noheader: display "Mean of X = " mean_X

Mean of kc = 72.536438
Mean of ki = 15.740885
Mean of kg = 20.606308
```

Note that when using a variable name in an expression, it refers to the first observation of that variable. Also, to suppress both the individual commands and their results, you need to specify `quietly` before `for`. The example we have used above repeats commands for a list of existing variables (`varlist`). You can also repeat for a list of new variables you want to create (`newlist`):

```
. for newlist ARG FRA USA : gen Xpop=pop if countryisocode=="X" & year==1995
```

It is also possible to repeat for a list of numbers (`numlist`) or any text you like (`anylist`). For example, suppose we wanted to append several similarly named data files to our existing dataset:

```
. for numlist 1995/1998: append using "H:\ECStata\dataX.dta"
```

Note, the full file name `H:\ECStata\dataX.dta` must be enclosed in double quotes, otherwise Stata will get confused and think the backslash `\` is a separator belonging to the `for` command:

```
. for numlist 1995/1998: append using H:\ECStata\dataX.dta

-> append using H:

file H: not found
r(601);
```

It is possible to nest several loops within each other. In this case, you need to specify the name of the macro Stata uses for the list specified after `for` (in above examples, Stata automatically used "X"):

```
. for X in varlist kg cgdp: for Y in numlist 1990/1995: sum X if year==Y
```

It is also possible to combine two or more commands into a single for-process by separating each command with a backslash `\`:

```
. for varlist kg cgdp, noheader: egen mean_X=mean(X) \ display "Mean of X = " mean_X
```

foreach and forvalues

The `for` loop is officially replaced from version 8 onwards by the `foreach` and `forvalues` loops. The former is used in conjunction with strings, the latter with numeric values. We know that it is possible to combine several commands into a single for-process. This can get quite complicated if the list of commands is quite long, but we saw how you can overcome this by

combining `for` with a custom-made program containing your list of commands. The `foreach` command does the same thing without the need for creating a separate program:

```
foreach var in kg cgdp {
	egen mean_`var'=mean(`var')
	display "Mean of `var' = " mean_`var'
}

Mean of kg = 20.606308
Mean of cgdp = 7.4677978
```

With the `foreach...in` command, `foreach` is followed by a macro name that you assign (e.g. `var`) and `in` is followed by the list of arguments that you want to loop (e.g. `kg cgdp`). This command can be easily used with variable names, numbers, or any string of text – just as `for`.

While this command is quite versatile, it still needs to be redefined each time you want to execute the same list of commands for a different set of arguments. For example, the program above will display the mean of `kg` and `cgdp`, but suppose that later on in your do-file you want to display the means of some other variables – you will have to create a new `foreach` loop. One way to get around this is to write the `foreach` loop into a custom-made program that you can then call on at different points in your do-file:

```
capture program drop mymean
program define mymean
foreach var of local 1 {
	egen mean_`var'=mean(`var')
	display "Mean of `var' = " mean_`var'
}
end

. mymean "kg cgdp"
Mean of kg = 20.606308
Mean of cgdp = 7.4677978
. mymean "ki pop"
Mean of ki = 15.740885
Mean of pop = 31252.467
```

This method works, but can be quite confusing. Firstly, `of local` is used in place of `in`. Secondly, reference to the `local macro `1'` in the first line does not actually use the single quotes we are used to. And thirdly, the list of arguments after the executing command must be in double quotes (so that everything is passed to the macro `` `1' `` in a single go). For these reasons, it can be a good idea to use `foreach` only when looping a one-off list. A technique called macro shift can be used when you want to loop a number of different lists (see later).

The `forvalues` loop works similarly but is defined over a range of values. Instead of `in` an equality sign is used and the values are given as a range, increasing by 1 unit or by a certain increment.

```
forvalues t = 1980/1983 {
	qui sum pop if countryisocode == "USA" & year == `t'
	di "pop of country USA in `t' is: " r(mean)
}
pop of country USA in 1980 is: 227726
pop of country USA in 1981 is: 230008
pop of country USA in 1982 is: 232218
pop of country USA in 1983 is: 234332

forvalues t = 1980(5)1990 {
	qui sum pop if countryisocode == "USA" & year == `t'
	di "pop of country USA in `t' is: " r(mean)
}
pop of country USA in 1980 is: 227726
pop of country USA in 1985 is: 238506
pop of country USA in 1990 is: 249981
```

Combining loops and macros

Now the `foreach` and `forvalues` loops combined with macros can be used cleverly to safe us a lot of work. We could for example generate a whole set of locals that hold the population for each country in each year in our dataset.

```
levelsof countryisocode
local ctries "`r(levels)'"
levelsof year
local years "`r(levels)'"
foreach ctr in `ctries' {
	foreach yr in `years' {
		sum pop if country == "`ctr'" & year == `yr'
		local `ctr'`year' = `r(mean)'
	}
}
```

The command `levelsof` returns a list of all distinct values of a categorical variable and saves them in macro `r(levels)`. We can use this list for the countries and years in our sample to define two loops that range over all possible values. For each value we summarize the population and define a local macro consisting of the countryisocode and year (e.g. `USA1990`) which takes on the value of the population in that year for that country.

Incremental shift (number of loops is fixed)

You can loop or repeat a list of commands within your do-file using the `while` command – as long as the `while` condition is true, the loop will keep on looping. There are two broad instances of its use – the list of commands are to be repeated a fixed number of times (e.g. 5 loops, one for each year 1980-84) or the number of repetitions may vary (e.g. maybe 5 loops for a list of 5 years, or maybe 10 loops for a list of 10 years). We will look first at the incremental shift technique for a fixed number of loops. We can see how it works using the following very simple example:

```
local i=1
while `i'<=5 {
	display "loop number " `i'
	local i=`i'+1
}
loop number 1
loop number 2
loop number 3
loop number 4
loop number 5
```

The first command defines a local macro that is going to be the loop increment – it can be seen as a counter and is set to start at 1. It doesn't have to start at 1, e.g. if you are looping over years, it may start at 1980.

The second command is the `while` condition that must be satisfied if the loop is to be executed. This effectively sets the upper limit of the loop counter. At the end of the `while` command is an open bracket `{` that signifies the start of the looped or repeated set of commands. Everything between the two curly brackets { } will be executed each time you go through the `while` loop.

The final command before the closing bracket `}` increases or increments the counter, readying it to go through the loop again (as long as the `while` condition is still satisfied). In actuality, it is redefining the local macro `i` – which is why there are no single quotes on the left of the equality but there are on the right. The increase in the counter does not have to be unitary, e.g. if you are using bi-annual data you may want to fix your increment to 2. All the looped commands within the brackets are defined in terms of the local macro `i`, so in the first loop everywhere there is an `i` there will now be a 1, in the second loop a 2, and so on. If the increase is unitary we can use the shorthand notation:

```
local ++i     /* same result as local i=`i'+1 */
```

To see a more concrete example, we will create a program to display the largest per capita GDP each year for every year 1980-84:

```
capture program drop maxcgdp
program define maxcgdp
	local i=1980
	while `i'<=1984 {
		tempvar mcgdp
		quietly egen `mcgdp'=max(cgdp) if year==`i'
```

```
            so `mcgdp'
            display `i' " " `mcgdp'
            local ++i
        }
end


maxcgdp

1980 9.4067564
1981 9.5102491
1982 9.5399132
1983 9.6121063
1984 9.7101154
```

Repeating commands using Stata's inbuilt functions.

Stata has some built-in routines to simplify standard repetitive tasks. The main function for this are `simulate` and `bootstrap`. The former allows us to generate random samples and evaluate some functions on this sample, while the latter resamples from a given dataset and again allows us to estimate some parameter of interest for each resample.

The bootstrap principle states that we treat the sample as the population of interest and draw samples (with replacement) from this population. This allows for repeated estimation of some parameter of interest and simulation of a finite sample distribution of that parameter. The most important aspect when resampling is to replicate the dependency structure in the original data as closely as possible. When data is sampled split by groups we need to use the `strata` option (cf. the section on survey commands). If there is correlation in the data (i.e. clustered) we need to account for this (by resampling clusters instead of observations).

The advantage of using the bootstrap command is that it saves the results nicely, and we have a lot of post bootstraps options and commands to choose from. The disadvantage is that Stata always uses a pair wise bootstrap and without additional programming other bootstrap variants are not applicable. The second major disadvantage is that weights cannot be used with `bootstrap`. This can be a major issue when working with survey data where sampling is often conducted with differing probabilities. So if we are interested in either nonstandard bootstrap methods or estimation with weights we will have to write our own bootstrap routine.

```
. reg cgdp openc pop

      Source |       SS       df       MS              Number of obs =      66
-------------+------------------------------           F(  2,    63) =   54.23
       Model |   698847107     2   349423553           Prob > F      =  0.0000
    Residual |   405925632    63     6443264           R-squared     =  0.6326
-------------+------------------------------           Adj R-squared =  0.6209
       Total |  1.1048e+09    65  16996503.7           Root MSE      =  2538.4

------------------------------------------------------------------------------
        cgdp |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       openc |   129.6153   24.69582     5.25   0.000     80.26461    178.9659
         pop |   .0577258   .0057838     9.98   0.000     .0461677    .0692838
       _cons |   11075.61    1550.57     7.14   0.000     7977.047    14174.18
------------------------------------------------------------------------------
```

We are interested in the standard error for the coefficient of the `openc` variable. In the most basic setting (leaving the number of repetitions at its default value and not saving the results after each repetition – something you should do in practice) we call the bootstrap as a prefix:

```
. bootstrap _b[openc]: reg cgdp openc pop
(running regress on estimation sample)

Bootstrap replications (50)
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
..................................................    50

Linear regression                               Number of obs      =        66
                                                Replications       =        50
```

```
      command:  regress cgdp openc pop
        _bs_1:  _b[openc]

------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _bs_1 |   129.6153   24.85778     5.21   0.000     80.89492    178.3356
------------------------------------------------------------------------------
```

The result is close to the standard error reported by Stata. But we do not account for the sample structure properly. Variables within country are probably correlated and thinking of countries as clusters is probably reasonable.

```
. bootstrap _b[openc], cluster(country): reg cgdp openc pop
(running regress on estimation sample)

Bootstrap replications (50)
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
..................................................    50

Linear regression                               Number of obs      =        66
                                                Replications       =        50

      command:  regress cgdp openc pop
        _bs_1:  _b[openc]

                                (Replications based on 6 clusters in country)
------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _bs_1 |   129.6153   93.78537     1.38   0.167    -54.20066    313.4312
------------------------------------------------------------------------------
```

When we have panel data or fixed groups in our data we need to make sure that the number of observations per cluster remains the same. Since clusters are resampled and some of them might appear twice (or even more often) the cluster identifier is not unique anymore. To avoid this problem we can specify a new variable that will be a unique cluster identifier again:

```
. bootstrap _b[openc], cluster(country) idcluster(tmpCountry): reg cgdp openc pop
(running regress on estimation sample)

Bootstrap replications (50)
----+--- 1 ---+--- 2 ---+--- 3 ---+--- 4 ---+--- 5
..................................................    50

Linear regression                               Number of obs      =        66
                                                Replications       =        50

      command:  regress cgdp openc pop
        _bs_1:  _b[openc]

                                (Replications based on 6 clusters in country)
------------------------------------------------------------------------------
             |   Observed   Bootstrap                         Normal-based
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _bs_1 |   129.6153   93.90392     1.38   0.167    -54.43303    313.6636
------------------------------------------------------------------------------
```

While `bootstrap` resamples from an existing data set, `simulate` allows us to generate new data from scratch and run the same command on several of those data sets. That is `simulate` allows us to perform Monte Carlo simulations. For this we write a program (see below) and call it using the `simulate` prefix.

Branching

Branching allows us to do one thing if a certain condition is true, and something else when that condition is false. For example, suppose you are doing some sort of analysis year-by-year but you want to perform different types of analyses for the earlier and later years. For simplicity, suppose you want to display the minimum per capita GDP for the years to 1982 and the maximum value thereafter:

```
capture program drop minmaxcgdp
program define minmaxcgdp
       local i=1980
       while `i'<=1984 {
              if `i'<=1982 {
                     local function min
                     }
              else {
                     local function max
              }
              tempvar mcgdp
              quietly egen `mcgdp'=`function'(cgdp) if year==`i'
              so `mcgdp'
              display `i' " " `mcgdp'
              local ++i
       }
end
. minmaxcgdp
1980 5.518826
1981 5.9500399
1982 6.0682001
1983 9.6121063
1984 9.7101154
```

The structure of this program is almost identical to that of the `maxcgdp` program created earlier. The only difference is that `egen` in line 6 is now a `min` or `max` function depending on the if/else conditions in lines 3 and 4.

It is very important to get the brackets `{}` correct in your programs. Firstly, every `if` statement, every `else` statement, and every `while` statement must have their conditions fully enclosed in their own set of brackets – thus, if there are three condition with three open brackets `{`, there must also be three close brackets `}`. Secondly, nothing except comments in `/* */` should be typed after a close bracket, as Stata automatically moves on to the next line when it encounters a close bracket. Thus, Stata would ignore the `else` condition if you typed:

```
. if `i'<=1982 {local function min} else {local function max}
```

Thirdly, it is necessary to place the brackets and their contents on different lines, irrespective of whether the brackets contain one or more lines of commands. Finally, it is possible to embed if/else statements within other if/else statements for extra levels of complexity, so it is crucial to get each set of brackets right. Suppose you want to display the minimum per capita GDP for 1980 and 1981, the maximum for 1982 and 1983, and the minimum for 1984:

```
capture program drop minmaxrgdpl
program define minmaxrgdpl
       local i=1980
       while `i'<=1984 {
              if `i'<=1981 {
                     local function min
              }
              else {
                     if `i'<=1983 {
                            local function max
                     }
                     else {
                            local function min
                     }
              }
              tempvar mrgdpl
              quietly egen `mrgdpl'=`function'(rgdpl) if year==`i'
              so `mrgdpl'
```

```
            display `i' " " `mrgdpl'
            local i=`i'+1
      }
end

. minmaxcgdp
1980 5.518826
1981 5.9500399
1982 9.5399132
1983 9.6121063
1984 6.1164122
```

One final thing to note is that it is important to distinguish between the conditional `if`:

```
. sum cgdp if cgdp>8
```

and the programming `if`:

```
if cgdp >8 {
      sum cgdp
}
```

The conditional `if` summarizes all the observations on `cgdp` that are greater than 8. The programming `if` looks at the first observation on `cgdp` to see if it is greater than 8, and if so, it executes the sum `cgdp` command, i.e. it summarizes *all* observations on `cgdp` (try out the two commands and watch the number of observations).

Writing Stata programs

In the preceding sections we already used Stata programs. In the following we will be more explicit about the syntax of Stata programs and we introduce some commands that facilitate writing programs.

Creating or "defining" a program

A program contains a set of commands and is activated by a single command. A do-file is essentially one big program – it contains a list of commands and is activated by typing:

```
. do "class 4.do"
```

You can also create special programs within a do-file, especially useful when you have a set of commands that are going to be used repetitively. The use of these programs will initially be demonstrated interactively, but they are best used within a do-file.

We will keep on using the PWT dataset from last week's classes. Suppose you want to create new variables that contain the average values (across countries and years) of some of the underlying variables in the dataset and at the same time display on screen these averages. No single Stata command will do this for you, but there are a couple of ways you can combine separate Stata commands to reach your goal. The most efficient method is:

```
. egen mean_kc=mean(kc)
. tab mean_kc

    mean_kc |      Freq.     Percent        Cum.
------------+-----------------------------------
   72.53644 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00

. egen mean_kg=mean(kg)
. tab mean_kg

    mean_kg |      Freq.     Percent        Cum.
------------+-----------------------------------
   20.60631 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00.
```

The tasks are the same for each variable you are interested in. To avoid repetitive typing or repetitive copying and pasting, you can create your own program that combines both tasks into a single command (note, in what follows the first inverted comma or single-quote of `` `1' `` is on the top-left key of your keyboard, the second inverted comma is on the right-hand side on the key with the @ symbol, and inside the inverted commas is the number one, not the letter L):

```
capture program drop mymean
program define mymean
       egen mean_`1'=mean(`1')
       tab mean_`1'
end
```

We have now created your own Stata command called `mean`, and the variable you type after this new command will be used in the program where there is a `` `1' ``. For example, `mean kg` will use `kg` everywhere there is a `` `1' ``. This command can now be applied to any variable you wish:

```
. mean ki

    mean_ki |      Freq.     Percent        Cum.
------------+-----------------------------------
   15.74088 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00
```

The syntax for a program is very basic. We define a program by `program define progname` and everything that follows until the `end` command is issued will be interpreted as belonging to the program. Stata will not let us define a program that is

already in memory and so we will need to erase any program before we define it anew. Otherwise we might get an error message. This is accomplished in the first line. The `capture` prefix is necessary because Stata return an error if there is no program called `mean` defined.

Macro shift (number of loops is variable)

Now we allowed only one variable in our function, but that seems rather restrictive. Instead of running the command again and again for each variable, we could write it more flexibly and allow for an arbitrary number of variables to be supplied as arguments. Without any other information Stata parses whatever we type after the command is called, using blanks to separate individual elements. The parsed string is saved in several local macros which are succinctly numbered. The macro `` `0' `` contains the whole string, `` `1' `` the first element, `` `2' `` the second, etc. We can loop over all elements without knowing how many there are by using the incremental shift technique. Let's rewrite the mean program to allow for an arbitrary number of variables:

```
capture program drop mymean
program define mymean
       while "`1'"~="" {
              egen mean_`1'=mean(`1')
              tab mean_`1'
              macro shift
       }
end

. mymean ki year

    mean_ki |      Freq.     Percent        Cum.
------------+-----------------------------------
   15.74088 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00

  mean_year |      Freq.     Percent        Cum.
------------+-----------------------------------
       1975 |      8,568      100.00      100.00
------------+-----------------------------------
      Total |      8,568      100.00
```

The command `macro shift` is used here instead of the counter increment device – it shifts the contents of local macros one place to the left; `` `1' `` disappears and `` `2' `` becomes `` `1' ``, `` `3' `` becomes `` `2' ``, and so on. So, in the example above, `` `1' `` initially contained the variable name "`kg`", `` `2' `` contained "`year`" and `` `3' `` was empty. The looped commands are in terms of `` `1' `` only so the first replication uses the variable `kg`. The `mac shift` command then shifts `` `2' `` into the `` `1' `` slot, so the second replication uses the variable `year`. The third macro is empty and therefore the loop ends.

The `while` command at the start of the loop ensures that it will keep on looping until the local macro `` `1' `` is empty, i.e. it will work as long as `` "`1'" `` is not an empty string `""`. This is similar to the `while` command in the incremental shift technique, but here the loop is defined in terms of `` `1' `` instead of `` `i' `` and it is contained in double quotes. The use of double quotes is a convenient way to ensure the loop continues until the argument or macro `` `1' `` contains nothing – it has nothing to do with whether the arguments are strings of text or numbers.

But the result looks rather ragged, we can improve our program by using some self-defined output.

```
capture program drop mymean
program define mymean
       while "`1'"~="" {
              quietly egen `1'_mean = mean(`1')
              display "Mean of `1' = " `1'_mean
              macro shift
       }
end
```

Now, we can display and generate the mean of a single variable:

```
. mymean kg
Mean of kg = 20.606308
```

or of a list of variables:

```
. mymean kg pop cgdp
Mean of kg = 20.606308
Mean of pop = 31252.467
Mean of cgdp = 7.4677978
```

The `quietly` prefix stops any results from being displayed. This is a very useful feature when writing programs. When displaying the mean of the variables we abuse the feature that when we refer to a variable in an expression, Stata considers only the first element of the variable. Here all elements of the mean variables are the same. But in general we should be very careful when referring to a variable in an expression since the sort order of that variable matters. If you are serious about good programming practice you will avoid this. The macro shift command is convenient, but it can be rather slow (all the macro contents need to be shifted). We can achieve the same by using an incrementing macro which involves only adding a 1 to a scalar, which is much faster.

```
capture program drop mymean
program define mymean
       local i = 1
       while "`i'"~="" {
              quietly egen `i'_mean = mean(`i')
              display "Mean of `i' = " `i'_mean
              local ++i
       }
end
```

Naming a program

You can give your program any name you want as long as it isn't the name of a command already in Stata, e.g. you cannot name it `summarize`. Actually, you can create a program called `summarize`, but Stata will simply ignore it and use its own `summarize` program every time you try using it. To check whether Stata has already reserved a particular name:

```
. which sum
built-in command: summarize
. which mymean
command mymean not found as either built-in or ado-file
r(111);
```

Debugging a program

Your program may crash out half-way through for some reason:

```
. mymean kc
mean_kc already defined
r(110);
```

Here, Stata tells us the reason why the program has crashed – you are trying to create a new variable called `mean_kc` but there a variable with that name already exists. Our `mean` program is a very simple one, so we can figure out very quickly that the problem arises with the first line, `egen mean_`1' = mean(`1')`. However, with more intricate programs, it is not always so obvious where the problem lies. This is where the `set trace` command comes in handy. This command traces the execution of the program line-by-line so you can see exactly where it trips up. Because the trace details are often very long, it is usually a good idea to `log` them for review afterwards.

```
. log using "debug.log",replace
. set more off
. set trace on
. mymean kg
. set trace off
. log close
```

You can also limit the amount of output by using `set tracedepth` to follow nested commands only to a certain depth. So if your program calls another program (which in turn calls another program) you can avoid the output from the second level onwards by using

```
. set tracedepth 1
```

Program arguments

Our `mean` command was defined to handle only one argument – `` `1' ``. We saw that it is possible to define more complicated programs to handle several arguments, `` `1' ``, `` `2' ``, `` `3' ``, and so on. These arguments can refer to anything you want – variable names, specific values, strings of text, command names, if statements, and so on. For example, we can define a program that displays the value of a particular variable (argument `` `1' ``) for a particular country (argument `` `2' ``) and year (argument `` `3' ``):

```
capture program drop show
program define show
       tempvar obs
       quietly gen `obs' = `1' if (countryisocode=="`2'" & year==`3')
       so `obs'
       display "`1' of country `2' in `3' is: " `obs'
end
```

To see this in action:

```
. show pop USA 1980
pop of country USA in 1980 is: 227726

. show pop FRA 1990
pop of country FRA in 1990 is: 58026.102
```

Some things to note about this program:

- Line 1 creates a temporary variable that will exist while the program is running but will be automatically dropped once the program has finished running. Once this `tempvar` has been defined, it must be referred to within the special quotes (`` `' ``), as explained in the previous section.
- Line 2 starts with `quietly`, which tells Stata to suppress any on screen messages resulting from the operation of the command on this line. We can use quietly either line by line or for whole blocks by using
  ```
  quietly {
      [some code]
  }
  ```
- Make sure to properly enclose any string arguments within double-quotes. `` `2' `` will contain a string of text, such as ARG or FRA, so when `` `2' `` is being used in a command it should be placed within double-quotes (`"`"). When the string we refer to actually entails quotation marks itself (or might contain if we write a program for general use) we need to use `` `"`"' ``, i.e. we need to further enclose the quotation marks by single quotation marks.
- Line 3 ensures that observation number one will contain the value we are interested in. Missings are interpreted by Stata as arbitrarily large, so when the data is sorted in ascending order our value will be at the top of the list, ahead of these missings.

As we have seen, use of strings can cause a bit of a headache. A further complication may arise if the argument itself is a string containing blank spaces, such as United States instead of USA. Stata uses blank spaces to number the different arguments, so if we tried `show kg United States 1980`, Stata would assign kg to `` `1' ``, United to `` `2' ``, States to `` `3' `` and 1980 to `` `4' ``. The way to get around this is to enclose any text containing important blank spaces within double-quotes – the proper command then would be:

```
. show kg "United States" 1980
```

Is this a good program? Most certainly not! Let's have a closer look at what is wrong. First of all we never mention what are the expected inputs, what the program does or how to use it properly. So comments are missing. Another unappealing feature is that we generate a variable where we could achieve the same result by using a macro or a scalar. So we waste memory. Lastly the sort order of the data is changed. Sorting is not only computationally very intensive but also changes the data. When writing programs you should try to leave the original data unchanged, unless the user explicitly specifies that the data should be modified.

Renaming arguments

Using `` `1' ``, `` `2' ``, `` `3' ``, and so on can be confusing and prone to error. It is possible to assign more meaningful names to the arguments at the very beginning of your program so that the rest of the program is easier to create. Make sure to continue to include your new arguments within the special quotes (`` `' ``):

```
capture program drop show
program define show
       args var cty yr
```

```
	tempvar obs
	qui gen `obs'=`var' if countryisocode==“`cty'” & year==`yr'
	so `obs'
	di “`var' of country `cty' in `yr' is: ” `obs'
end

show kg USA 1980
kg of country USA in 1980 is: 13.660507
```

The `args` command assigns the local macros `var`, `cty` and `yr` the value of `` `1' ``, `` `2' `` and `` `3' `` respectively. The three numbered local macros are not dropped but you can still access them as before. Note that if you call the program `show` with four or more arguments the code will not return an error but simply fill the (unused) local macros `` `4' ``, `` `5' `` etc. with the additional arguments.

A more robust method to parse arguments, which checks whether the program is called correctly, is the `syntax` command. Instead of referring to each element of a program call by its position in the list, we can specify proper “grammar”, that is we use the standard layout of Stata programs.

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

Stata automatically checks whether the program call satisfies the provided syntax and returns an error message if that is not the case. Let us look at some examples.

```
capture program drop mymean
program define mymean
syntax varlist(min=1 numeric) [if] [in] [, SUFfix(string)]
	if "`suffix'" == "" local suffix "_mean"
	foreach var in `varlist' {
		confirm new var `var'`suffix'
	}
	foreach var in `varlist' {
		qui egen `var'`suffix' = mean(`var') `if' `in'
		qui sum `var'`suffix'
		display "Mean of `var' = " r(mean)
	}
end

. mymean country
varlist:  country:  string variable not allowed
r(109);

. mymean pop cgdp openk
Mean of pop = 31252.467
Mean of cgdp = 3819.5117
Mean of openk = 67.921783

. mymean pop cgdp openk if year < 1995
pop_mean already defined
r(110);

. mymean pop cgdp openk if year < 1995, suf(_t)
Mean of pop = 30172.367
Mean of cgdp = 3019.5283
Mean of openk = 64.883232
```

We specify that our program `mean` requires a list of variables (consisting of at least one variable and all variables having to be numeric). Optionally the user can choose to subset the data using the `if` and `in` qualifier or to provide a custom suffix other than the standard ending we chose. Brackets indicate optional arguments and for the options the upper case letters indicate the shortest abbreviation that can be used for the option. In the following we check whether a custom suffix was provided and choose the standard extension if that is not the case. We also control that each variable that should be generated does not already exist.

Notice that we exchanged the `while` loop for a `foreach` loop. Instead of working with the variable list we could have kept the local macros `` `1' ``, `` `2' ``, etc. by parsing the variable list using the `tokenize` command. The `tokenize` command splits a string

and saves the result in numbered local macros. By default a blank space is assumed to separate elements and so we could have used `tokenize `varlist'` to fill the local macros and keep the original version of the code.

Programs with return values and other options

So far our program generates variables that take on averages and displays them. We might want to access the averages after running the program. To do this we need to define the program as one of three return classes: `eclass`, `rclass` or `sclass`. The first class should be used when writing estimation commands. Programs that are `eclass` allow for a variety of post-estimation commands and so they should be well integrated (meaning provide all the necessary values for post-estimation commands). The second class is for general results, for example descriptive statistics and the last class is seldom used since it is limited to returning local macros.

When we define a program such that it allows to save results in `e()` or `r()`, all results in `e()` and `r()` that are present will be deleted once we specify to return the new results. So if we are interested in some of those results we need to explicitly keep them.

```
capture program drop mymean
program define mymean, rclass
syntax varlist(min=1 numeric) [if] [in] [, SUFfix(string)]
       tempname meanmat
       matrix `meanmat' = J(wordcount("`varlist'"),1,.)
       mat rownames `meanmat' = `varlist'
       mat coln `meanmat' = "mean
       if "`suffix'" == "" local suffix "_mean"
       foreach var in `varlist' {
              confirm new var `var'`suffix'
       }
       local i = 0
       foreach var in `varlist' {
              local ++i
              qui egen `var'`suffix' = mean(`var') `if' `in'
              qui sum `var'`suffix'
              mat `meanmat'[`i',1] = r(mean)
              display "Mean of `var' = " r(mean)
       }
       return matrix mean = `meanmat'
end

. mymean grgdpch ki kg kc
Mean of grgdpch = 2.0699065
Mean of ki = 15.740885
Mean of kg = 20.606308
Mean of kc = 72.536438

. mat li r(mean)

r(mean)[4,1]
              mean
grgdpch  2.0699065
     ki  15.740885
     kg  20.606308
     kc  72.536438
```

We create a temporary matrix `` `meanmat' `` to save the mean values of our variables. The first line after the `syntax` command tells Stata to create a temporary object, the second line fills the object with a column vector of missing values of the same length as the number of words in the provided variable list. The third and fourth line assign names to the rows and columns of the matrix and finally we augment the previous code by adding another line in the `foreach` loop. We assign the average value of the `` `i'- `` th variable to the `` `i'- `` th row of our matrix. To make the matrix accessible after the program has finished we use the `return` command, specifying the type of return value, here a matrix. Note that the returned object will cease to exist once we execute the `return` command. If we want to avoid this we need to use the `copy` option.

We can be fairly lax about naming of our return values (though we should stick to some conventions – see the programming manual on this). When we want to define an `eclass` program, this is not the case. Stata's post-estimation commands check whether certain `e()` values exist and only if they do the program will execute. To facilitate the return of estimation results there is a special subcommand to post the coefficient vector and the variance matrix (among other quantities of interest, such as the

degrees of freedom of the estimation and an indicator for the sample that was used).

```
cap prog drop myols
prog def myols, eclass sortpreserve // sortpreserve restores current sort order
/* Bad programming version of the OLS command without small sample SE correction
	varlist	: dependent variable followed by explanatory variables (all models
include an intercept)
	robust : white heteroscedasticity robust standard errors
*/
syntax varlist(min=1) [if] [in] [, robust]
	tempname xx xy beta vcov // temporary matrices
	tempvar residual `varlist' cons id // temporary variables
	tokenize `varlist' // load varlist into numbered local macros
	marksample touse // create a temporary variable that indicates the sample
	markout `touse' `varlist' // use only those observations w/o missing
	qui count if `touse' // determine number of obs
	scalar nobs = r(N)
	local yvar "`1'" // first variable in the provided varlist is the dep. var.
	mac shift // get rid of `1', `*' will take the value of `2' and higher
	qui matrix accum `xx' = `*' if `touse' // generate the X'X matrix
	matrix vecaccum `xy' = `varlist' if `touse' // generate X'y
	matrix `beta' = inv(`xx') * (`xy')' // OLS estimator b for beta
	local residpred "`cons'" // string that takes the formula to predict X'b
	local i = 1 // count the number of variables
	foreach var in `*' {
		qui gen ``var'' = `beta'[`i',1] if `touse' // generate b_k
		local ++i
		local residpred "`residpred' + ``var'' * `var'" // + b_k * X_k
	}
	qui gen `cons' = `beta'[`i',1] if `touse'
	qui gen `residual' = `yvar' - (`residpred') if `touse' // predict residual
	if "`robust'" != "" { // if robust option is used
		qui gen `id' = _n if `touse' // indicator for clusters
		sort `id' // opaccum requires sorted data
		matrix opaccum `vcov' = `*' if `touse', group(`id') opvar(`residual')
		// X'ee'X
		matrix `vcov' = inv(`xx') * `vcov' * inv(`xx') // sandwich formula
	}
	else { // w/o robust option
		qui matrix accum `vcov' = `residual', nocons // e'e
		matrix `vcov' = inv(`xx') * `vcov' / nobs // (X'X)^(-1) * e'e
	}
	local nobs = nobs // n
	matrix `beta' = (`beta')' // beta is colvector, need rowvector for ereturn post
	local dof = `nobs' - `i' // n-k
	ereturn post `beta' `vcov', dep("`yvar'") obs(`nobs') esample(`touse')
dof(`dof')
	ereturn local cmd "myols"
	ereturn display
end

. reg pop cgdp openk

      Source |       SS       df       MS              Number of obs =    5842
-------------+------------------------------           F(  2,  5839) =  124.48
       Model |  2.7997e+12     2  1.3998e+12           Prob > F      =  0.0000
    Residual |  6.5660e+13  5839  1.1245e+10           R-squared     =  0.0409
-------------+------------------------------           Adj R-squared =  0.0406
       Total |  6.8459e+13  5841  1.1720e+10           Root MSE      =  1.1e+05

------------------------------------------------------------------------------
         pop |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        cgdp |   .4531388   .2647725     1.71   0.087    -.0659133    .9721909
       openk |  -430.7306   27.34693   -15.75   0.000    -484.3407   -377.1205
       _cons |   58807.65   2400.476    24.50   0.000     54101.83    63513.47
```

```
------------------------------------------------------------------------------

. myols pop cgdp openk
------------------------------------------------------------------------------
         pop |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        cgdp |   .4531388   .2647045     1.71   0.087      -.06578    .9720576
       openk |  -430.7306   27.33991   -15.75   0.000    -484.3269   -377.1342
       _cons |   58807.65   2399.859    24.50   0.000     54103.04    63512.26
------------------------------------------------------------------------------

. reg pop cgdp openk, robust

Linear regression                                      Number of obs =    5842
                                                       F(  2,  5839) =  164.93
                                                       Prob > F      =  0.0000
                                                       R-squared     =  0.0409
                                                       Root MSE      =  1.1e+05

------------------------------------------------------------------------------
             |               Robust
         pop |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        cgdp |   .4531388   .1692357     2.68   0.007     .1213742    .7849035
       openk |  -430.7306   25.71796   -16.75   0.000    -481.1473   -380.3139
       _cons |   58807.65   3249.723    18.10   0.000     52436.99    65178.31
------------------------------------------------------------------------------

. myols pop cgdp openk, robust
------------------------------------------------------------------------------
         pop |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        cgdp |   .4531388   .1691922     2.68   0.007     .1214594    .7848183
       openk |  -430.7306   25.71136   -16.75   0.000    -481.1344   -380.3268
       _cons |   58807.65   3248.888    18.10   0.000     52438.63    65176.67
------------------------------------------------------------------------------
```

The program produces exactly the same coefficients but slightly different standard errors. The reason is that Stata automatically uses some small sample correction, but our program doesn't. Notice that we returned more than just the coefficient vector and the variance matrix. We need to return `e(cmd)` if we want post-estimation commands to run after the estimation. We also return the number of observations, the degrees of freedom and an indicator that marks the observations that were used. Some post-estimation commands require these inputs to work correctly (e.g. `test`). We also used a new set of commands in this program, the `marksample` and `markout` commands. The former generates a temporary variable that takes on the value 1 if an observation is part of the estimation sample and 0 otherwise. The latter sets the temporary variable to 0 if there are missing observations in the provided variable list. The reason for using a temporary variable is that in the following we can use `if` `` `touse' `` as an addition to all our commands to make sure that all operations are conducted on the same sample, no lengthy logical expressions beyond this one temporary variable are necessary.

One of the most useful features of Stata is the `by` prefix. We use it a lot for variable creation and for generating descriptive statistics. To allow our programs to work with `by`, we need to use the `byable` option. What `by` does is to execute the code several times restricting the sample to the respective cell indicated by the variable list of `by`. The easiest case is when our program does not change the data (i.e. no variables are changed or generated), e.g. if our program calculates some summary statistics or an estimator. In this case we only need to make sure that our program utilizes the `marksample` command to set the sample that we want to use and add the `byable(recall)` option to our program definition. If we generate variables or the data is changed in some way we will run into problems. The reason is that repeated execution of our program will lead to different results.

To handle this problem we need to change our code so that it behaves differently depending on whether the code runs with or without the `by` prefix and if it runs with the prefix we need to control the behavior for each execution of our program. We can still use the `byable(recall)` option, but if it makes things easier there is also the `byable(onecall)` option which tells Stata that we will control the execution of the program ourselves, i.e. Stata does not loop automatically over the groups formed in `by`.

```
capture program drop mymean
program define mymean, rclass byable(recall)
////////////////////////////////////////////////////////////////////////////////
// generate and display average values. allows for by prefix                  //
//     suffix():    suffix for variable with mean as value (_mean as default) //
//     return:      matrix with all average values for all groups in r(mean)  //
//                  command to generate group identifier, use: egen smth = `r(grp)' //
////////////////////////////////////////////////////////////////////////////////
syntax varlist(min=1 numeric) [if] [in] [, SUFfix(string)]
       marksample touse
       tempname meanmat
       cap matrix `meanmat' = r(mean)
       if _byindex() == 1 {
              if _by() == 1 {
                     qui sum `_byindex'
                     local grps = r(max)
                     qui replace `touse' = 0 if `_byindex' != _byindex()
              }
              else local grps = 0
              matrix `meanmat' = J(wordcount("`varlist'"),max(1, `grps'),.)
              mat rownames `meanmat' = `varlist'
              if `grps' == 0 local matcolnames "mean"
              else {
                     local matcolnames ""
                     forvalues grp = 1/`grps' {
                            local matcolnames "`matcolnames' group`grp'"
                     }
              }
              mat coln `meanmat' = `matcolnames'
       }
       if "`suffix'" == "" local suffix "_mean"

       if _byindex() == 1 {
              foreach var in `varlist' {
                     confirm new var `var'`suffix'
              }
       }
       local i = 0
       if _byindex() == 1 {
              foreach var in `varlist' {
                     local ++i
                     qui egen `var'`suffix' = mean(`var') if `touse'
                     qui sum `var'`suffix' if `touse'
                     mat `meanmat'[`i',1] = r(mean)
                     if _by() == 0 display "Mean of `var' = " r(mean)
                     else display "Mean of `var' = " r(mean)
              }
       }
       else {
              tempvar tmp
              foreach var in `varlist' {
                     local ++i
                     qui egen `tmp' = mean(`var') if `touse'
                     qui replace `var'`suffix' = `tmp' if `touse'
                     qui sum `var'`suffix' if `touse'
                     mat `meanmat'[`i',_byindex()] = r(mean)
                     display `"Mean of `var' = "' r(mean)
                     qui drop `tmp'
              }
       }
       return matrix mean = `meanmat'
       return local grp "group(`_byvars')"
end
```

The code for our `mean` program is now a lot more complex. The reason is that we have to distinguish between the first and repeated calls of the program. We do this by branching depending on the value of Stata's inbuilt `by`-related functions. We use the indicator function _by() to determine whether the program was called using the `by` prefix. The function is 1 if `by` was used, 0 if not. The second function we use is `_byindex()`. That function takes on the value of the current repetition of the code, i.e. it is 1 if the code runs for the first time (the first group or if `by` was not used), 2 if it is executed for the second time (the second group) and so on. In addition to the `by`-related functions we also use temporary variables generated `by` the by command. We need to be careful with them, because those temporary variables only exist when `by` was actually used. The temporary variable `` `_byindex' `` is a unique group identifier which is 1 for the first group, 2 for the second, and so on.

Help files and publishing programs

If we want to make our programs available to a wider audience we should (and probably want to) provide some guidelines as how to use the program. The standard way of doing this is to create a help document that the user can access (when properly installed) by typing `help commandname`. To ensure operating system independence Stata uses its own syntax to generate help files. The coding used is SMCL (Stata Markup and Control Language) which is similar to HTML (HypterText Markup Language), the coding used for (most) internet websites. In SMCL we control the display of text by tags, commands that are enclosed in curly brackets: {}. For example we should start our help file always with `{smcl}` indicating that the following text is formatted in SMCL. Tags can take two forms, either a tag applies to all text that follows afterwards, like `{smcl}` which indicates that all the following text is SMCL text, or the tag encompasses the text that we want to format, for example `{Title:MyTitle}` will display "MyTitle" formatted as the title of a help file.

```
{smcl}
{* smcl ensures that Stata uses SMCL code instead of plain ASCII}
{* *! version 1.0.1  18jan2009}{...}{* * is the command for comments}
{* when we add ... in curly brackets at the end of a comment Stata will interpret}{...}
{* the line as nonexistent, otherwise an empty line will show in the help file}{...}
{cmd:help mymean} {* cmd: highlights the following text as a command}
{hline}{* draws a horizontal line, if followed by a number the line is not drawn over
the whole page but only a certain amount of characters long}
{title:Title} {* section title}

{p2colset 7 15 17 2}{...}
{* changes the display preferences for a line with two columns. The first column is 7
characters indented and the second column starts 15 characters from the left and ends 2
characters before the line ends, the second line of the second column starts 17
characters from the left (and also ends 2 characters from the right).}{...}
{p2col :{hi:mymean} {hline 2}}Report the mean and generate a variable taking the
mean{p_end}{* p2col : and what follows in the same brackets defines the first column,
everything outside the curly brackets until p_end defines the second column.}
{p2colreset}{* reset the display preferences}{...}


{title:Syntax}

{p 6 16 2}{* The following paragraph is indented 6 characters on the first line, 16
characters on the second line and stops 2 characters from the right end.}
{cmd:mymean} {varlist} {ifin}
[{cmd:,}
{it:options}]

{synoptset 20 tabbed}{* starts a table with syntax options}{...}
{synopthdr}
{synoptline}
{syntab:Options}
{synopt :{opth suf:fix(string)}}manually set the suffix of the new variable; default is
_mean{p_end}{* opth highlights the abbreviations by underlining the part of the option
before the : and links the term "string" to the appropriate help file}
{synoptline}
{p2colreset}{...}
{p 4 6 2}
{opt by} is allowed; see {manhelp by D}.


{title:Description}
```

```
{pstd}
{cmd:mymean} generates variables based on {varlist} that take the average value.
{cmd:mymean} also reports the average value and returns a matrix of all averages.


{title:Options}

{dlgtab:Options}

{phang}{* shortcut for p 4 8 2. There are several other paragraph shortcuts available,
e.g. pstd or phang2.}
{opt suffix} optionally specify a string that changes the default suffix (_mean).


{title:Examples}

{pstd}Setup{p_end}
{phang2}{cmd:. sysuse auto}{p_end}

{pstd}Calculate the average for {cmd:weight} and {cmd:length}{p_end}
{phang2}{cmd:. mymean weight length}{p_end}

{pstd}Calculate the average for {cmd:weight} and {cmd:length} seperately for domestic
and foreign cars with different suffix.{p_end}
{phang2}{cmd:. bysort foreign: mymean weight length, suffix("_meanf")}{p_end}


{title:Saved results}

{pstd}
{cmd:mymean} saves the following in {cmd:r()}:

{synoptset 15 tabbed}{...}
{p2col 5 15 19 2: Matrices}{p_end}
{synopt:{cmd:r(mean)}}Matrix of average values for all variables of {varlist} and for
all groups if {opt by} was used.{p_end}
{synoptset 15 tabbed}{...}
{p2col 5 15 19 2: Macros}{p_end}
{synopt:{cmd:r(grp)}}command to generate a unique group identifier. Usage: {cmd:egen
uniqueID = `r(grp)'}{p_end}
{p2colreset}{...}


{title:Also see}

{psee}
Online:  {manhelp egen D}, {manhelp summarize R}
{p_end}{* manhelp links the appropriate helpfiles}
```

You can save the above code in a file called “mymean.sthlp” or “mymean.hlp” and open it in Stata by using the `view` command followed by the path and the filename, e.g. `view H:\ECStata\mymean.sthlp`.

Writing a help file is one step we should take before making our code public. The second step we should take is to make our code readily accessible from within Stata. To do this we need to create a package. A package is simply a short description of your program and all the files that accompany the main program. Our file mymean.pkg would look like this:

```
v 1.0
* v is followed by the version number
d mymean: generate and display the average value of variables.
d Allows for "by" and returns a matrix with all results.
d Program by Alexander C. Lembcke
* d is followed by a description of the program
f mymean.ado
f mymean.sthlp
* and lastly f denotes filenames that belong to your program
```

Before we can publish the file one further step is necessary. To make sure that Stata understands that the folder on my website includes Stata programs we need a file called "stata.toc" in the same folder. This file is helpful, especially when we have several programs on our website.

```
v 1.0
d Programs accompanying the course in advanced Stata methods
d mymean: Generate and display the average value of variables.
d Author: Alexander C. Lembcke
p mymean Average value generation
* p links to a program package (to the mymean.pkg file) in the same folder.
```

Now if we put this file along with the help, package and ado file on my website (e.g. in http://personal.lse.ac.uk/lembcke/ecStata/programs) we can simply install the program from Stata by typing:

```
net from http://personal.lse.ac.uk/lembcke/ecStata/programs
net install mymean
```

Maximum likelihood methods

Stata has an inbuilt maximum likelihood estimation routine: `ml`. The `ml` command uses numerical optimization to fit likelihood functions for non-linear optimization problems. In general we can use `ml` to maximize any objective function that can be represented as the sum of scalars, i.e. we could set up a least squares estimator with `ml`. Stata's own commands make use of `ml`, for example the `logit` and `probit` estimators. Here we will consider an extension to the simple binary choice model. We will explicitly model the (expected) utility of agents and back out the parameters of the utility function. There is a restriction on the type of likelihood functions that we can estimate using `ml`. We need that the likelihood can be written as a linear form (linear form restriction) in the observations. That means we need the (log) likelihood to be the sum of scalar functions over the range of our observations. This mainly excludes estimators where contributions to the likelihood are correlated within groups.

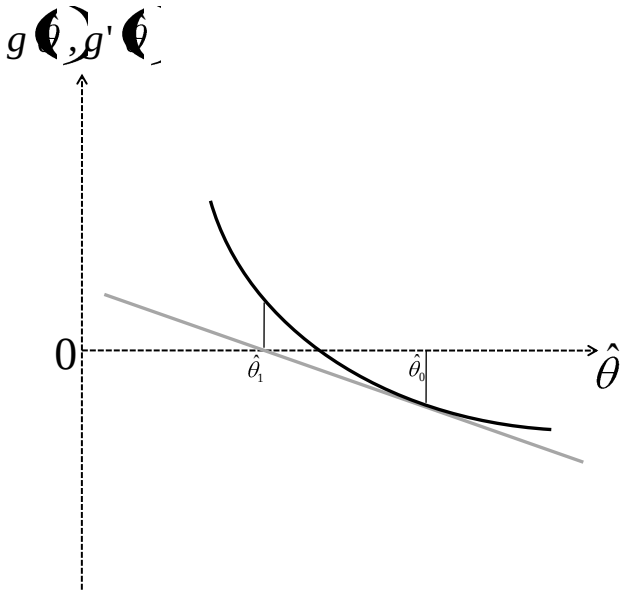
Stata uses one of four maximization algorithms to find the maximum likelihood with a (Stata modified) Newton-Raphson (NR) algorithm as standard. The other algorithms are less computationally expensive but NR is supposedly very robust in its convergence properties, so unless we are worried about computation time we can stick to it. Although if NR fails due to some peculiar aspects of the data interacting with the NR algorithm, one of the other three might converge.

Maximization theory

When we try to find the maximum of a function (here the likelihood function) we "simply" take the first derivative and set it equal to 0. Checking the second derivative tells us whether we found a maximum or a minimum. The mechanism for Stata's `ml` command is the same. Stata tries to find values for the parameters that set the gradient, i.e. the first derivative of the likelihood function, equal to 0. This is achieved in an iterative process:

1. Use some initial value for your parameter vector
2. Calculate the value of the gradient
3. Given the current value of the gradient, update the parameter vector
4. Calculate the new value of the gradient and repeat until convergence

This algorithm is the same for all maximization methods, what differs is the way the parameter update in step 3 is calculated. The update involves two questions. First, should we increase or decrease a parameter, i.e. what direction should the change take and second how far should the step go. We can depict this idea for a one dimensional parameter vector. NR uses the tangent of the gradient at our initial guess $\hat{\theta}_0$ to determine the direction and the step size. The direction is chosen by going in the "steeper" direction (here to the left) and the step size is determined by finding the value at which the tangent crosses the horizontal axis.



So from $0 = g'(\hat{\theta}_0)(\hat{\theta}_1 - \hat{\theta}_0) + g(\hat{\theta}_0)$ we have that $\hat{\theta}_1 = \hat{\theta}_0 - \frac{g(\hat{\theta}_0)}{g'(\hat{\theta}_0)}$. Therefore we need three functions for our algorithm, the initial (likelihood) function, its gradient and the derivative of the gradient, the Hessian. If we can find analytical solutions for the gradient and the Hessian we can provide `ml` with the functions and make our estimation more precise and probably faster. But

Stata only requires the specification of the likelihood function, the derivatives can be calculated numerically. For example for the derivative of the gradient:

$$g'(\hat{\theta}_0) = \lim_{h \to 0} \frac{g(\hat{\theta}_0 + h) - g(\hat{\theta}_0)}{h}$$

Which can be (and is) approximated by simply using a small enough h. One of the problems with this approach is that it leads to imprecision. Even if we generate all the functions in double format, the most precise format Stata knows, the numerical derivative is only exact up to the 8th digit (half the precision of the double format).

Creating the first ml estimation

The problem that we consider for exposition purposes is whether someone decides to enroll in higher education or not. Again, we will not make any claim about running sensible regressions, but use the example at hand for exposition purposes only. We use an artificial sample that has information on (expected) future earnings and assume that the decision to enroll in higher education is solely based on this information. To start things off let's look at a simple logit model in which a student enrolls at a university if the expected lifetime earnings with a university degree are higher than the lifetime earnings without a degree. Formalizing this argument yields (all equations can be found in standard Econometrics textbooks, e.g. W.H. Greene *Econometric Analysis*):

$$Prob(enroll = 1|x) = F(totearningsUNI - totearningsNoUNI, \beta) = F(x, \beta)$$

$$F(x, \beta) = \frac{e^{x'\beta}}{1 + e^{x'\beta}}$$

$$f(x, \beta) = \frac{e^{x'\beta}}{(1 + e^{x'\beta})^2} = F(x, \beta)\big(1 - F(x, \beta)\big)$$

Where $F(.)$ is the CDF and f(.) the PDF of the logistic distribution and $enroll$ is a dummy that is 1 if a student enrolls in higher education and 0 otherwise. The likelihood function for binary choice models is given by

$$L(\beta|enroll, x) = \prod_i [F(x_i, \beta)]^{enroll_i} [1 - F(x_i, \beta)]^{1 - enroll_i}$$

Analytically the product is quite tedious to deal with, since we need to take derivatives to find the maximum of the likelihood function. Computationally we have a similar problem. The elements of the product are values in the range [0,1] and therefore the product over all observations will be a very (very very very) small number, even for rather small sample sizes (e.g. 40 observations). It is not possible to generate variables precise enough for Stata to evaluate the product unambiguously. The solution is the same as for the analytical case, taking logs and maximizing the log-likelihood makes the product a sum and the precision problem a lot less severe.

$$lnL(\beta|enroll, x) = \sum_i enroll_i * \ln[F(x_i, \beta)] + (1 - enroll_i) * \ln[1 - F(x_i, \beta)]$$

$$\frac{\partial lnL}{\partial \beta} = \sum_i \left(enroll_i * \frac{f(x_i, \beta)}{F(x_i, \beta)} + (1 - enroll_i) * \frac{-f(x_i, \beta)}{1 - F(x_i, \beta)} \right) x_i = \sum_i \big(enroll_i - F(x_i, \beta)\big) x_i$$

$$\frac{\partial lnL}{\partial \beta \partial \beta'} = -\sum_i f(x_i, \beta) x_i x_i'$$

These three equations, log-likelihood, the gradient and the Hessian are all we need to run a logit estimation by hand (as explained above we only need the gradient and the Hessian if we want to use `ml` at its full potential).

So let's run a comparison between `ml` and the Stata `logit` command. To facilitate the use of the `ml` command we need to define a program that conducts the calculation of the likelihood function. As before we drop the program to make sure we always use the most current version and no errors occur in our do file. Afterwards we define the program and use the `args` command to parse the syntax. The first argument we pass depends on the type of likelihood evaluation we want to conduct. We can call the `ml` command using one of the four methods: `lf`, `d0`, `d1` or `d2`. The first method only requires the specification of the likelihood function and everything else is automated. The latter three require more manual adjustments, including the specification of the gradient (`d1` and `d2`) and the Hessian (`d2`). For our first pass we will use the most straightforward method `lf`. If we use `lf` we need to provide `lnf` as the first argument of the `args` command. The second argument we provide is `xb`. This argument can be named anything, but always entails the inner product of regressor and coefficient vector ($x_i'\beta$) for each equation we specify. Note

that we need to specify an argument for each equation we will use in our call of the `ml` command.

```
cap program drop ml_logit
program def ml_logit
      args lnf xb
      tempvar enroll
      qui gen double `enroll' = $ML_y1 if $ML_samp
      qui replace `lnf' = `enroll' * ln(invlogit(`xb')) ///
                        + (1 - `enroll') * ln(1-invlogit(`xb')) if $ML_samp
      // F(z) = exp(z)/(1+exp(z)) = invlogit(z)
end
```

The only obligatory element after we parsed the arguments is the assignment of the likelihood function. The likelihood function is automatically generated (thanks to `args`) with double precision with all missing values. All we have to do is replace the temporary variable `` `lnf' `` with the appropriate function. But in order to make our code more readable we also generate a temporary variable `` `enroll' `` which takes the value of our dependent variable. Dependent variables are named in the order that we specify them, so `$ML_y1` is the global macro containing the name of the first dependent variable. To allow for `if` and `in` qualifiers we restrict our sample range using the `$ML_samp` global macro. The `$ML_samp` macro works in the same way as the `` `touse' `` temporary variable that we used in the first section of this handout. Note that technically we do not need restriction here because (unless specified otherwise) `ml` will preserve the data and drop all non-sample observations when called. The way the `ml` command works is that it will provide a first guess for all parameters in our model (by default all parameters are set to 0) and simply calculate the value of the likelihood function, first the likelihood is calculated for each observation (that is what we do when we replace the `` `lnf' `` temporary variable) and then Stata takes the sum over all observations to find the log likelihood.

Writing a program to calculate the likelihood is the first step in finding our parameter estimates. Two more steps are necessary, first we need to specify the model by telling Stata what the dependent and independent variable(s) is(are) and second we need to start the maximization. The `model` subcommand defines the model we want to estimate. The first argument it takes is the method we want to use (`lf`, `d0`, `d1` or `d2`), the second argument is the name of the program that defines the likelihood function. Anything after these two arguments is considered the definition of an estimation equation. For our simple example here we only need one equation with one dependent and one independent variable. Note that by default each equation will have a constant in it. So we specify the first equation, separating the dependent variables from the independent variables by an equal sign. To start the maximization, all we need to do is to use the `maximize` subcommand.

```
. ml model lf ml_logit (enroll = totearndiff)

. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -723.57698
rescale:       log likelihood = -693.14518
Iteration 0:   log likelihood = -693.14518
Iteration 1:   log likelihood = -581.87399
Iteration 2:   log likelihood = -581.05598
Iteration 3:   log likelihood =  -581.0537
Iteration 4:   log likelihood =  -581.0537

                                                Number of obs   =       1000
                                                Wald chi2(1)    =     164.95
Log likelihood =  -581.0537                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
      enroll |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
 totearndiff |   .1606867   .0125112    12.84   0.000     .1361651    .1852083
       _cons |    -.81457   .0952173    -8.55   0.000    -1.001193   -.6279476
------------------------------------------------------------------------------

. scalar s_mine = _b[totearndiff]
```

We achieve convergence after four iterations, that means the change of the likelihood function in the last step is so small that it falls below the threshold of the `ml` routine, i.e. we consider the change to be (about) zero at machine precision. With this one equation setting Stata reports automatically the number of observations we used for the estimation as well as a Wald test for the joint significance of our regressors. We also get the standard, nicely formatted, output table that Stata produces after estimation commands. Note that we did not specify any options regarding the variance, so Stata calculates the variance matrix based on the inverse of the negative Hessian. Instead of using the Hessian we can use the outer product of the gradient vector as our variance

estimator, we do this by invoking the vce(opg) option when we specify `ml model`. Robust and clustered variance estimators are also readily available and can be applied with the usual `robust` and `cluster()` options. Both types of robust estimates are sandwich estimates where the "bread" is the inverse of the negative Hessian and the "butter" is some sum over the outer product of gradients.

```
. logit enroll totearndiff

Iteration 0:   log likelihood = -693.14518
Iteration 1:   log likelihood =  -584.5413
Iteration 2:   log likelihood = -581.07937
Iteration 3:   log likelihood =  -581.0537
Iteration 4:   log likelihood =  -581.0537

Logistic regression                                     Number of obs   =       1000
                                                        LR chi2(1)      =     224.18
                                                        Prob > chi2     =     0.0000
Log likelihood =  -581.0537                             Pseudo R2       =     0.1617

------------------------------------------------------------------------------
      enroll |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
 totearndiff |   .1606867   .0125112    12.84   0.000     .1361651    .1852083
       _cons |    -.81457   .0952173    -8.55   0.000    -1.001192   -.6279475
------------------------------------------------------------------------------

. scalar s_stata = _b[totearndiff]

. scalar s_diff = s_stata - s_mine

. di "Stata coefficient: " s_stata " ML lf coefficient: " s_mine " Difference: " s_diff
Stata coefficient: .16068668 ML lf coefficient: .16068668 Difference: -4.433e-09
```

At first glance our estimator yields the same results as Stata's `logit` command. The log likelihood is the same, the coefficients are the same and so are the standard errors. Only for the confidence interval of the intercept term the numbers differ. But at second glance we see that the numbers for the other estimates differ as well, just not at the first few digits, for example the coefficient estimate of our estimator compared to Stata's estimator differs at the 9th digit. The reason is, that `logit` provides the gradient and the Hessian and we used only the likelihood function, leaving the gradient and Hessian to be approximated numerically. Lastly the `logit` command reports a LR instead of a Wald test for joint significance of the regressors.

Specifying the gradient and Hessian by hand

Simple likelihood functions are easily handled with the `lf` method. But if we want to evaluate more complex functions finding the numerical derivatives might be cumbersome or even lead to problems in convergence. If this is the case and we can find analytic solutions for the gradient and the Hessian we can provide them explicitly to alleviate the problems. Another reason why we might want to make use of analytic derivatives is that they are more precise than numerical derivatives and lastly execution of our code might be quicker. A real advantage over the `lf` method for the last two aspects is only achieved when we specify both gradient and Hessian, i.e. we use the `d2` method. The `d1` method uses an analytic gradient but a numerically approximated Hessian and the `d0` method only uses the likelihood and as `lf` derives gradient and Hessian numerically.

The `lf` method in the previous section did a lot of things automatically that we now have to control by hand. We start our program again by parsing the syntax through the `args` command. Now the first argument is `todo` which takes on the value 0, 1 or 2, depending on the method we use when calling the program. The next argument is `b` which is the parameter vector. The parameter vector is a row vector that includes all parameters from all equations. Next is `lnf` which defines a scalar (as opposed to a temporary variable in the `lf` case!) which takes on the log likelihood. The last two arguments are `g` and `negH`, a row vector and a matrix with the same number of rows as `b`, that we use to specify the gradient and the (negative) Hessian. The programs we write for the `d` methods are backward compatible, i.e. a program that works running with `d2` also works with `d1` and `d0` and a program for `d1` can also be used with `d0`.

```
// logit d0
cap program drop ml_logit0
program def ml_logit0
       args todo b lnf
       tempvar enroll xb lnf_i
```

```
        qui gen double `enroll' = $ML_y1 if $ML_samp
        qui mleval `xb' = `b'
        qui gen double `lnf_i' = `enroll' * ln(invlogit(`xb')) ///
                                  + (1 - `enroll') * ln(1-invlogit(`xb')) if $ML_samp
        qui mlsum `lnf' = `lnf_i'
        if (`lnf' >= .) exit
end
```

The inner product of regressor and parameter vector was automatically generated when we used `lf`. For the `d` methods we use the command `mleval` to generate the same variable. The `mleval`, as well as all the other utility commands starting with `ml` that we will see in the next two programs account automatically for the selected sample and, if necessary, for missing values. Since `` `lnf' `` is now a scalar, we need to aggregate the individual likelihood contributions by hand. We could simply use one of the in-built sum functions but this is a bad idea. The sum functions ignore missing values by treating them as 0. But here missing values are informative, they tell us that our likelihood cannot be evaluated and therefore something is going wrong. To avoid this problem we need to use `mlsum`. Another advantage of `mlsum` is that it also accounts for weights if they were specified. If there are missing values in the aggregation that leads to `` `lnf' ``, the scalar `` `lnf' `` will be set to missing. We stop the optimization if we encounter such a situation.

```
// logit d1
cap program drop ml_logit1
program def ml_logit1
        args todo b lnf g
        tempvar enroll xb lnf_i
        qui gen double `enroll' = $ML_y1 if $ML_samp
        qui mleval `xb' = `b'
        qui gen double `lnf_i' = `enroll' * ln(invlogit(`xb')) ///
                                  + (1 - `enroll') * ln(1-invlogit(`xb')) if $ML_samp
        qui mlsum `lnf' = `lnf_i'
        if (`lnf' >= .) exit
        qui mlvecsum `lnf' `g' = `enroll' - invlogit(`xb')
end
```

Changing a `d0` into a `d1` program requires only one addition, the specification of the gradient vector. We add the `g` when parsing the syntax and generate a matrix `` `g' `` using the `mlvecsum` command. When our model specification involves the parameter vector only as a linear combination with the regressors, i.e. the gradient is the outer derivative multiplied with the regressor vector, we can use `mlvecsum` to aggregate the gradient easily. All we need to specify is the outer derivative of the likelihood and `mlvecsum` aggregates the product of the outer derivative and the vector of regressors for all observations. If the gradient is more complex we need to do more work, generating the gradient for each parameter and aggregating the resulting variables into a matrix `` `g' ``.

```
// logit d2
cap program drop ml_logit2
program def ml_logit2
        args todo b lnf g negH
        tempvar enroll xb lnf_i
        qui gen double `enroll' = $ML_y1 if $ML_samp
        qui mleval `xb' = `b'
        qui gen double `lnf_i' = `enroll' * ln(invlogit(`xb')) ///
                                  + (1 - `enroll') * ln(1-invlogit(`xb')) if $ML_samp
        qui mlsum `lnf' = `lnf_i'
        if (`lnf' >= .) exit
        qui mlvecsum `lnf' `g' = `enroll' - invlogit(`xb')
        qui mlmatsum `lnf' `negH' = invlogit(`xb')*(1-invlogit(`xb'))
end
```

Finally we add the Hessian (or rather the negative Hessian) to the program. Again if our parameters come in only linearly combined with the regressors, i.e. in the form $x_i'\beta$, we can use a utility command to do the aggregation for us. The `mlmatsum` command generates a matrix out of the outer derivative (specified after the equal sign) and the outer product of the regressor vector with itself ($x_i x_i'$). To run the programs we use the same commands as before, now with the appropriate `d` method instead of `lf` as argument.

```
. ml model d0 ml_logit0 (enroll = totearndiff)

. ml maximize
```

```
initial:       log likelihood = -693.14718
alternative:   log likelihood = -723.57698
rescale:       log likelihood = -693.14518
Iteration 0:   log likelihood = -693.14518
Iteration 1:   log likelihood = -581.87237
Iteration 2:   log likelihood = -581.05598
Iteration 3:   log likelihood =  -581.0537
Iteration 4:   log likelihood =  -581.0537

                                                Number of obs   =       1000
                                                Wald chi2(1)    =     164.95
Log likelihood =  -581.0537                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
      enroll |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
 totearndiff |   .1606867   .0125113    12.84   0.000     .1361651    .1852083
       _cons |  -.8145701   .0952173    -8.55   0.000    -1.001193   -.6279476
------------------------------------------------------------------------------

. ml model d1 ml_logit1 (enroll = totearndiff)

. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -723.57698
rescale:       log likelihood = -693.14518
Iteration 0:   log likelihood = -693.14518
Iteration 1:   log likelihood = -581.87399
Iteration 2:   log likelihood = -581.05598
Iteration 3:   log likelihood =  -581.0537
Iteration 4:   log likelihood =  -581.0537

                                                Number of obs   =       1000
                                                Wald chi2(1)    =     164.95
Log likelihood =  -581.0537                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
      enroll |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
 totearndiff |   .1606867   .0125112    12.84   0.000     .1361651    .1852083
       _cons |    -.81457   .0952173    -8.55   0.000    -1.001192   -.6279475
------------------------------------------------------------------------------

. ml model d2 ml_logit2 (enroll = totearndiff)

. ml maximize

initial:       log likelihood = -693.14718
alternative:   log likelihood = -723.57698
rescale:       log likelihood = -693.14518
Iteration 0:   log likelihood = -693.14518
Iteration 1:   log likelihood = -581.87399
Iteration 2:   log likelihood = -581.05598
Iteration 3:   log likelihood =  -581.0537
Iteration 4:   log likelihood =  -581.0537

                                                Number of obs   =       1000
                                                Wald chi2(1)    =     164.95
Log likelihood =  -581.0537                     Prob > chi2     =     0.0000

------------------------------------------------------------------------------
      enroll |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
 totearndiff |   .1606867   .0125112    12.84   0.000     .1361651    .1852083
```

```
       _cons |    -.81457   .0952173    -8.55   0.000    -1.001192   -.6279475
------------------------------------------------------------------------------
. scalar s_d2 = _b[totearndiff]

. scalar s_diff2 = s_stata - s_d2

. di "Stata coefficient: " s_stata " ML d2 coefficient: " s_d2 " Difference: " s_diff2
Stata coefficient: .16068668 ML d2 coefficient: .16068668 Difference: 1.428e-11
```

Extension to non-standard estimation

The logit example helps understand how the `ml` command works. But since it is already implemented there is not much use in reprogramming it. However if we make the combination of parameters and regressors a little more complex we will find that the `logit` command will not suffice. Let's have a look at a slight change in our problem specification. Instead of using the total future earnings as determinant for whether someone takes up tertiary education or not we will use the discounted (expected) utility derived from future earnings.

So we consider high school graduates at the time of graduation and we will try to estimate the time preference and the intertemporal elasticity of substitution based on their decision to attend or not to attend to a university. We model the decision as:

$$enroll = 1 \; if \; E(u_{uni}) - E(u_{nouni}) > \varepsilon$$

where ε follows the logistic distribution with mean zero and variance $\frac{\pi^2}{3}$ and for the utilities (which are functions of the PDV of future earnings) we use a second order Taylor approximation:

$$E[u(x_i)] = u(E[x_i]) + u'(E[x_i]) \underbrace{E(x_i - E[x_i])}_{0} + \frac{1}{2}u''(E[x_i]) \underbrace{E(x_i - E[x_i])^2}_{var(x_i)}$$

with the PDV: $x_i = \sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it}$

We still need to specify a utility function, I choose a CRRA utility function:

$$u(x_i) = x_i^{1-\rho} \qquad u'(x_i) = (1-\rho)x_i^{-\rho} \qquad u''(x_i) = -\rho(1-\rho)x_i^{-1-\rho}$$

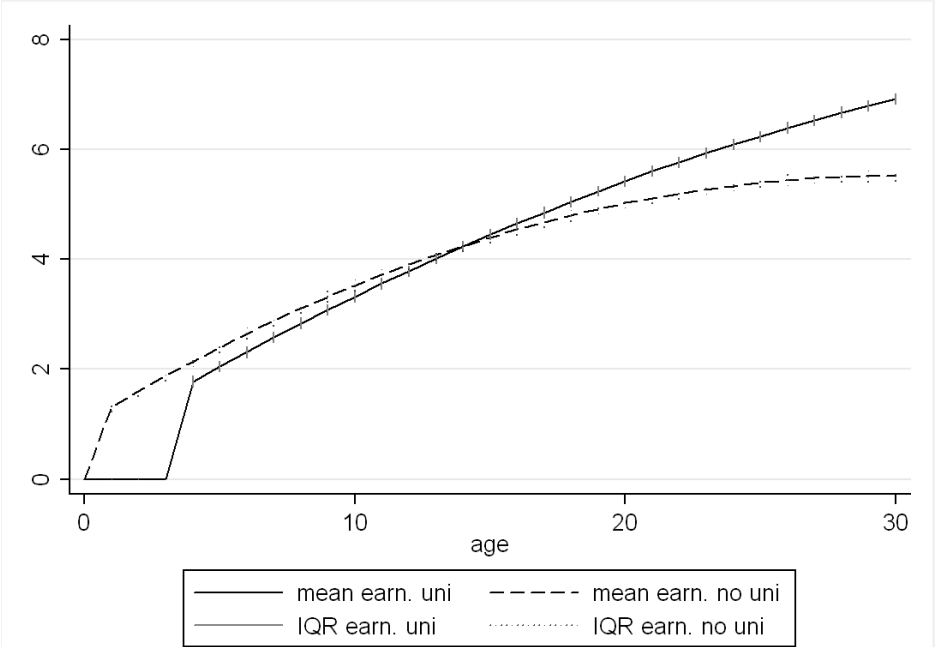
Putting everything together we find the log-likelihood to be:

$$lnL(\beta|enroll, x) = \sum_i enroll_i * \ln[F(DIFF_i)] + (1 - enroll_i) * \ln[1 - F(DIFF_i)]$$

$$\begin{aligned} DIFF_i = \; & \left[\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,uni}\right]^{1-\rho} - \left[\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,nouni}\right]^{1-\rho} \\ & -\frac{1}{2}\,\rho(1-\rho)\left[\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,uni}\right]^{-1-\rho} VAR\left(\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,uni}\right) \\ & +\frac{1}{2}\,\rho(1-\rho)\left[\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,nouni}\right]^{-1-\rho} VAR\left(\sum_{t=0}^{30} \frac{1}{(1+\delta)^t} x_{it,nouni}\right) \end{aligned}$$

with $VAR(z)$ the sample variance of z.

I generated the data using $\delta = 0.05$ and $\rho = .5$, for the earnings I generated the future earnings streams for both going to university and not going to university. In both cases the paths are concave, they cross after 15 (of 30) years and have about the same within age variation. I assume that acquiring a university degree takes 3 years during which the wage is zero. For the variance I used the variation of PDVs across individuals.



The earnings are saved in wide format, i.e. earnings at age 1 are earn_uni1 and earn_nouni1, at age 2: earn_uni2 and earn_nouni2, etc. So let's see how `ml` fares when we estimate this rather complex model.

For our first try we will impose no restrictions and just estimate the model given the functional form that we chose.

```
cap program drop ml_CRRA
program def ml_CRRA
// CRRA utility: U(y) = (SUM_t=0^T 1/(1+delta)^t * x_t)^(1-rho), no restriction on rho
    args lnf delta rho
    tempvar earn_uni earn_nouni enroll earn_uni_var earn_nouni_var
    qui gen double `enroll'             = $ML_y1
    qui gen double `earn_uni'           = $ML_y2
    qui gen double `earn_nouni'         = $ML_y3
    forvalues j = 1(1)$horizon {
        qui replace `earn_uni' = `earn_uni' + 1/((1+`delta')^`j') * (${ML_y2}`j')
        qui replace `earn_nouni'=`earn_nouni' + 1/((1+`delta')^`j') * (${ML_y3}`j')
    }
    qui egen double `earn_uni_var' = sd(`earn_uni')
    qui egen double `earn_nouni_var' = sd(`earn_nouni')
    qui replace `earn_uni_var' = (`earn_uni_var')^2
    qui replace `earn_nouni_var' = (`earn_nouni_var')^2

  qui replace `lnf' = ln(invlogit(`earn_uni'^(1-`rho') - `earn_nouni'^(1-`rho')     ///
            - .5 * `rho' * (1-`rho') * `earn_uni_var' * `earn_uni'^(-1-`rho')      ///
            + .5 *`rho' * (1-`rho') * `earn_nouni_var' * `earn_nouni'^(-1-`rho'))) ///
            if `enroll' == 1
  qui replace `lnf' = ln(1 - invlogit(`earn_uni'^(1-`rho') - `earn_nouni'^(1-`rho') ///
            - .5 *`rho' * (1-`rho') * `earn_uni_var' * `earn_uni'^(-1-`rho')       ///
            + .5 *`rho' * (1-`rho') * `earn_nouni_var' * `earn_nouni'^(-1-`rho'))) ///
            if `enroll' == 0
end
```

We use the `lf` method again and leave finding the derivatives to Stata. Since we don't want to call our code with 62 explanatory variables (the 31 years in our sample), we generate them in a way that we can simply refer to each of them. The earnings in the first year are `earn_uni` and `earn_nouni`, for the second year the earnings are `earn_uni1` and `earn_nouni1` and so on. We call our program providing three dependent variables but no explanatory variables. In principle we could estimate both parameters (delta and rho) conditional on some other covariates, i.e. we could make them vary by groups, but that extension is not very complicated. We could also add further explanatory variables, i.e. make the decision to enroll not only a function of earnings and the corresponding parameters but add other background characteristics.

The first thing we do is generate new variables that make it easier to understand the content of the underlying variables, the first of the dependent variables is the actual outcome and the second and third are the earnings variables (without a suffix). We generate the PDV in a loop using a global macro so we can easily adapt the horizon from outside the program and we don't have to hardcode (i.e. plug a number in) it. The likelihood function is more complex than before, but otherwise we do not introduce anything new. So let's see how our code fares.

```
. global horizon = 30
. ml model lf ml_CRRA (sigma: enroll earn_uni earn_nouni=) /rho
. ml maximize

initial:       log likelihood = -1858.9705
alternative:   log likelihood = -720.48316
rescale:       log likelihood = -693.14718
rescale eq:    log likelihood = -693.14718
could not calculate numerical derivatives
flat or discontinuous region encountered
r(430);
```

Utilities to check our estimation

That's not good. Before the iterative maximization process starts we get an error message. The first four lines are just Stata trying to find good starting values. The initial values Stata tries are all parameters equal to zero (unless we specify this to be otherwise). It is no problem if your likelihood function is actually not defined for all zeros, Stata will try different values (second line) and try to improve upon them by using a scaled version of these initial guesses (third and fourth line) only after this process the iterations (should) start.

So what is the problem? Did we make a mistake when writing our program? Or did the algorithm just run into a bad set of parameters and the likelihood is not smooth enough at that point to steer it out of that corner? Well, this is hard to say. First we should check whether the result we received makes sense. If for example our first line would have read:

```
initial:       log likelihood =     -<inf>  (could not be evaluated)
```

This should have given us pause. Using all zero parameter values is absolutely feasible in our model (i.e. discount rate 0 and risk neutrality), so we might have a problem with our program. But since that is not the case here we can try other utilities. The `query` subcommand shows us a summary of our current maximization problem.

```
. ml query

Method:           lf
Program:          ml_CRRA
Dep. variables:   enroll earn_uni earn_nouni
2 equations:
       /delta
       /rho
Search bounds:
       /delta          -inf       +inf
       /rho            -inf       +inf
Current (initial) values:
        (zero)
```

The given bounds that Stata uses to find initial guesses for our parameter might be excessive. We know that our discount factor should be something positive and small and we might (for our initial values) assume that agents are risk averse and limit the search bounds. We can do this by explicitly searching for initial values with the `search` subcommand.

```
. ml search /rho .1 5 /delta .0001 .2
initial:       log likelihood = -693.14718
improve:       log likelihood = -693.14068
rescale:       log likelihood = -693.14068
rescale eq:    log likelihood = -693.11935

. ml query

Method:           lf
Program:          ml_CRRA
```

```
Dep. variables:   enroll earn_uni earn_nouni
2 equations:
       /delta
       /rho
Search bounds:
       /rho              .1          5
       /delta         .0001         .2
Current (initial) values:
   rho:_cons           1.8651637
   delta:_cons        -.02359858
lnL(current values) = -693.11935
```

Stata finds initial values, but a negative discount rate does not seem very promising. But nonetheless calling the program with these new initial values gives us convergence.

```
. ml maximize

initial:       log likelihood = -693.11935
rescale:       log likelihood = -693.11935
rescale eq:    log likelihood = -693.11935
Iteration 0:   log likelihood = -693.11935  (not concave)
Iteration 1:   log likelihood = -692.98064  (not concave)
Iteration 2:   log likelihood = -692.88123
Iteration 3:   log likelihood = -692.80287
Iteration 4:   log likelihood =  -692.7813
Iteration 5:   log likelihood = -692.78107
Iteration 6:   log likelihood = -692.78107

                                                Number of obs   =       1000
                                                Wald chi2(0)    =          .
Log likelihood = -692.78107                     Prob > chi2     =          .

------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
rho          |
       _cons |   1.127624   .2447222     4.61   0.000     .6479779    1.607271
-------------+----------------------------------------------------------------
delta        |
       _cons |   -.127306   .2637092    -0.48   0.629    -.6441665    .3895546
------------------------------------------------------------------------------
```

This does not look very good. The first two iterations give us `(not concave)` warning. This is not of much concern because it vanishes after two iterations. If the message persists or occurs in the last iteration we are in trouble. The warning tells us that at the current parameter estimates the derivatives are flat. Either our model is wrong, the program we wrote is wrong or Stata does not find the true maximum but rather some local optimum. If our likelihood is complicated and not very smooth that can easily be the case. But before we look more into the problem of finding a local rather than a global optimum we can run a check whether our program meets Stata's expectations about maximum likelihood programs.

```
. ml check

[...]
------------------------------------------------------------------------------
                          ml_CRRA HAS PASSED ALL TESTS
------------------------------------------------------------------------------
[...]
```

At least we did not make any obvious mistakes when setting up our code. But again the maximization will lead to the same nonsensical estimates. So we can try two things, first we could set plausible initial values by hand or second we could try and use some graphical optimization procedure to find good initial values. The `plot` subcommand allows us to get a graphical impression of the likelihood optimization with respect to one parameter, i.e. all other parameters are fixed at the current value. Stata will choose the new initial value for the parameter we use as the optimal point in the graph. Notice that this means that running the `ml plot` command for each parameter several times might improve the initial value even more. We call the `ml plot` command specifying the parameter (or rather referring to the equation with Stata assuming we mean the intercept of that equation) that we want to graph and optionally select the range of parameter values that should be plotted. The range is given by a

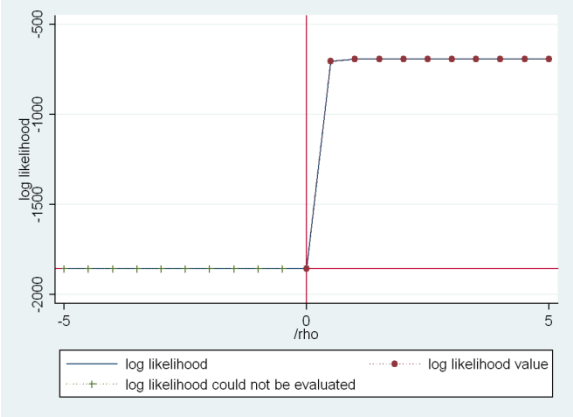
number which tells Stata to plot the graph around the current initial value +/- the given number.

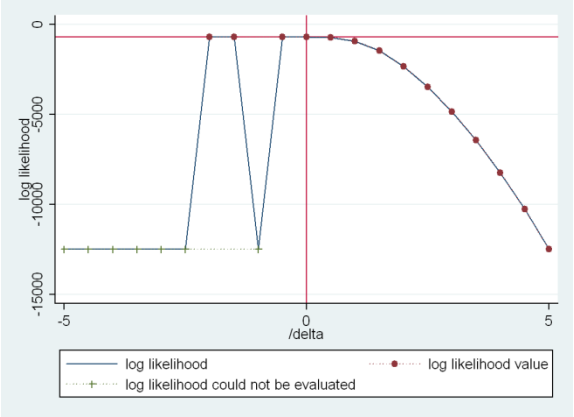
```
. ml plot /rho 5

                reset /rho =        1.5  (was          0)
            log likelihood = -693.08536  (was -1858.9705)

. ml plot /delta 5

             keeping /delta =          0
             log likelihood = -693.08536
```





We can see why Stata chose a negative `delta`, the likelihood (conditional on `rho` being 1.5) has several spikes at negative values. So finally we can try and set our own initial values.

```
. ml init .5 .05, copy

. ml maximize, difficult

initial:       log likelihood = -686.59862
rescale:       log likelihood = -686.59862
rescale eq:    log likelihood = -686.59862
Iteration 0:   log likelihood = -686.59862
Iteration 1:   log likelihood = -686.48508  (not concave)
Iteration 2:   log likelihood = -686.46657
Iteration 3:   log likelihood = -686.45468
Iteration 4:   log likelihood = -686.45446
Iteration 5:   log likelihood = -686.45446

                                                Number of obs   =       1000
                                                Wald chi2(0)    =          .
Log likelihood = -686.45446                     Prob > chi2     =          .

------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
rho          |
       _cons |   .4084066   .1773892     2.30   0.021     .0607302    .7560831
-------------+----------------------------------------------------------------
delta        |
       _cons |   .0370233    .024168     1.53   0.126     -.010345    .0843917
------------------------------------------------------------------------------
```

Now our estimates make a lot more sense and we actually found a better fit, the log likelihood is higher than for the previous result. Note that we maximized using the `difficult` option for maximize. This is a useful option when we have badly behaved likelihood functions. In some cases even with bad starting values I got found the correct estimates when I used this option (though with the right initial values the option is superfluous). What we should take away from this example is that we should always! play around with the initial values to see whether our optimal point is local rather than global.

What else could we have done? Well we can always use a trace command to see whether some part of our code did not work. The commands `ml trace on` or `set trace on` can be used for that. We can also check the current values Stata uses as parameter by simply displaying them from within our code by using `sum `delta' `rho'` or `di `delta' " " `rho'`.

If we want to see the Hessian with each step we can use the `hessian` option with `ml maximize`. And finally if we wrote a `d2` method program we can define the program with debug option: `ml d2debug ...` which will make Stata report the difference between the analytical and the numerical derivatives, a very useful feature to test whether we made a mistake. Lastly we should always use common sense. For example if we encounter positive likelihoods, we should pause for a second and check whether that makes sense (it doesn't we probably just forgot to take the log when generating the log likelihood).

Flexible functional forms and constraints

We saw that the first estimate of our discount parameter was actually negative, something that does not make any sense economically. To prevent this we might want to put constraints on our optimization. The standard way to handle constraints in Stata is by defining them using the `constraint` command and then using the `constraint` option when calling a command. This works as well for `ml`, but sadly only for equality constraints. This might be useful if we want to impose some cross-equation constraints or something like that, but it is not very helpful when we want Stata to evaluate our program only on the positive domain for one of our parameters. The solution is to estimate our model for a transformed version of our parameter. To constrain `delta` to be positive we can simply estimate our program using `ln(delta)` instead.

$$\delta = \exp\big(ln(\delta)\big) = \exp(\ln\_\delta)$$

Since we now use `ln_delta` as our parameter, we have to replace each occurrence of `delta` with `exp(delta)`. In order for us not having to rewrite the whole code, we can just use the above equality to generate a new variable.

```
cap program drop ml_CRRA_dpos
program def ml_CRRA_dpos
// vNM utility: U(y) = (SUM_t=0^T 1/(1+delta)^t * x_t)^(1-rho), rho positive
       args lnf rho ln_delta
       tempvar earn_uni earn_nouni enroll delta earn_uni_var earn_nouni_var
       qui gen double `enroll'          = $ML_y1
       qui gen double `earn_uni'  = $ML_y2
       qui gen double `earn_nouni'      = $ML_y3
       qui gen double `delta' = exp(`ln_delta')
    forvalues j = 1(1)$horizon {
        qui replace `earn_uni' = `earn_uni' + 1/((1+`delta')^`j') * (${ML_y2}`j')
        qui replace `earn_nouni'=`earn_nouni' + 1/((1+`delta')^`j') * (${ML_y3}`j')
    }
    qui egen double `earn_uni_var' = sd(`earn_uni')
    qui egen double `earn_nouni_var' = sd(`earn_nouni')
    qui replace `earn_uni_var' = (`earn_uni_var')^2
    qui replace `earn_nouni_var' = (`earn_nouni_var')^2

  qui replace `lnf' = ln(invlogit(`earn_uni'^(1-`rho') - `earn_nouni'^(1-`rho')     ///
             - .5 * `rho' * (1-`rho') * `earn_uni_var' * `earn_uni'^(-1-`rho')      ///
             + .5 *`rho' * (1-`rho') * `earn_nouni_var' * `earn_nouni'^(-1-`rho'))) ///
             if `enroll' == 1
  qui replace `lnf' = ln(1 - invlogit(`earn_uni'^(1-`rho') - `earn_nouni'^(1-`rho') ///
             - .5 *`rho' * (1-`rho') * `earn_uni_var' * `earn_uni'^(-1-`rho')       ///
             + .5 *`rho' * (1-`rho') * `earn_nouni_var' * `earn_nouni'^(-1-`rho'))) ///
             if `enroll' == 0
end
```

After running the code we need to use `nlcom` to get our estimate for the discount rate. We see that we obtain approximately the same result as before.

```
. ml model lf ml_CRRA_dpos (rho: enroll earn_uni earn_nouni=) /ln_delta

. ml init .5 -3, copy

. ml maximize

initial:       log likelihood = -686.60506
```

```
rescale:       log likelihood = -686.60506
rescale eq:    log likelihood = -686.60506
Iteration 0:   log likelihood = -686.60506
Iteration 1:   log likelihood = -686.46177
Iteration 2:   log likelihood = -686.45743
Iteration 3:   log likelihood = -686.45448
Iteration 4:   log likelihood = -686.45446

                                                Number of obs   =       1000
                                                Wald chi2(0)    =          .
Log likelihood = -686.45446                     Prob > chi2     =          .

------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
rho          |
       _cons |   .4083984   .1773738     2.30   0.021     .0607521    .7560447
-------------+----------------------------------------------------------------
ln_delta     |
       _cons |  -3.296243    .652696    -5.05   0.000    -4.575504   -2.016982
------------------------------------------------------------------------------

. nlcom rho: exp([ln_delta]_b[_cons])

         rho:  exp([ln_delta]_b[_cons])

------------------------------------------------------------------------------
             |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         rho |    .037022   .0241641     1.53   0.125    -.0103388    .0843828
------------------------------------------------------------------------------
```

Constraining a coefficient to lie between positive and negative one, $\theta \in (-1,1)$ is possible by using

$$\theta = \frac{\exp(2 * \text{trans\_}\theta) - 1}{\exp(2 * \text{trans\_}\theta) + 1}$$

This way $\text{trans\_}\theta$ can take any value and the result will lie inside the unit circle. If we want to expand that circle we can use a simple transformation of this constraint. Lastly we might want to constrain our coefficient estimate to be $\theta \in (0,1)$, this can be achieved by using a CDF, e.g.:

$$\theta = \frac{\exp(trans\_\theta)}{1 + \exp(trans\_\theta)}$$

Further reading

Glenn W. Harrison (2008) *Maximum Likelihood Estimation of Utility Functions Using Stata* (http://www.bus.ucf.edu/wp/Working%20Papers/2006/06-12%20Harrison.pdf)
William Gould, Jeffrey Pitblado and William Sribney (2006) *Maximum Likelihood Estimation with Stata (3rd edition)*
A. Colin Cameron and Pravin K. Trivedi (2009) *Microeconometrics using Stata*

Mata

Since its 9th version Stata includes a matrix programming language with similar syntax and capabilities as other matrix languages, e.g. Matlab, Gauss or R have. You can easily access the Stata variables and manipulate them in the Mata, if you choose to do so. On the other hand we can transfer all the Stata variables into the Mata environment and use only Mata in our analysis. But this is probably not such a great idea since it means duplicating the data, i.e. saving it in Stata and Mata at the same time.

What is Mata and why bother?

Mata is a matrix language, so data is used and stored as scalars, vectors and matrices. Mata is quick and useful in subsetting matrices and allows us to circumvent the matrix size restriction in Stata. Even if the maximum matrix size in Stata is not the issue, Mata is able to store data more efficiently (by keeping it as Stata variables) than we could in Stata's traditional matrix language. Another advantage in terms of computational speed comes from Mata being precompiled so that code doesn't have to be interpreted during each run (as it is in ado files). If we plan to do any matrix calculations (e.g. GMM) Mata is probably the best way to implement them. A lot of matrix functions are readily available and more and more user written commands utilize Mata as well.

The main strength of Mata in Stata compared to other matrix languages is that we can use Stata to deal with the data set, i.e. load the data set, generate variables, etc. and then call Mata to do our calculations. Unlike Stata commands, Mata commands might not deal with missing observations nicely. So selecting your sample before calling Mata is a good idea. Mata has a speed advantage over Stata, which mainly comes from Mata functions being compiled instead of interpreted interactively. The compiler checks our code for inconsistencies and streamlines it, i.e. where it can it makes the code more efficient.

Mata basics

We invoke Mata with the `mata` command and finish a Mata session with `end`. We can also start Mata with `mata:`, where the difference is that the latter command will stop the execution of code once an error occurs, starting Mata without the double point will result in error messages if errors occur but no interruption in the code's execution. If we only want to run one Mata command we can do so by putting it on the same line as the `mata` command and omitting the `end`.

Mata is a matrix language and the way it stores information is in matrix form. The easiest way to generate a Mata object is by simply assigning values to it.

```
mata
x = (1, 2 \ 3, 4)
x
end
mata x
```

```
. mata
---------------------------------- mata (type end to exit) -----------------------------
: x = (1, 2 \ 3, 4)

: x
       1   2
    +---------+
  1 |  1   2  |
  2 |  3   4  |
    +---------+

: end
------------------------------------------------------------------------------------------

. mata x
       1   2
    +---------+
  1 |  1   2  |
  2 |  3   4  |
    +---------+
```

The comma joins elements to the same row, the backslash joins elements to form columns. We can either assign a value to an object, here we define the matrix x to be 2x2 with elements 1 through 4, and refer to an object to display its contents, or display some results directly by not assigning the value. We are also not constrained to use only numerical values in our matrices, we

might as well use strings (though not both at the same time).

```
. mata
------------------------------------------- mata (type end to exit) ----------------------
: (1, 2 \ 3, 4) + (1, 2 \ 3, 4)
       1   2
    +---------+
  1 |  2   4  |
  2 |  6   8  |
    +---------+

: : s = ("first", "second" \ "third", "fourth")

: s
             1        2
    +-------------------+
  1 |   first   second  |
  2 |   third   fourth  |
    +-------------------+

: m = ("first", 2 \ 3, 4)
type mismatch:  string , real not allowed
r(3000);

: end
-------------------------------------------------------------------------------------------
```

Operators as well as functions can apply to the whole matrix or to each element separately. With a colon operators and arithmetic functions apply to each element, without a colon to the whole matrix.

```
. mata
------------------------------------------- mata (type end to exit) ----------------------
: x = (1, 2 \ 3, 4)

: x * x
        1    2
    +-----------+
  1 |   7   10  |
  2 |  15   22  |
    +-----------+

: x # x //Kronecker product
        1    2    3    4
    +---------------------+
  1 |   1    2    2    4  |
  2 |   3    4    6    8  |
  3 |   3    6    4    8  |
  4 |   9   12   12   16  |
    +---------------------+

: x :* x
        1    2
    +-----------+
  1 |   1    4  |
  2 |   9   16  |
    +-----------+

: sqrt(x)
                1             2
    +-----------------------------+
  1 |             1   1.414213562  |
  2 |  1.732050808             2  |
    +-----------------------------+

: end
-------------------------------------------------------------------------------------------
```

Two extremely useful functions to generate matrices are the `I()` and the `J()` function. The first generates an identity matrix, the latter a matrix of any shape consisting of any value we choose.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: I(2)
[symmetric]
       1   2
    +---------+
  1 |  1      |
  2 |  0   1  |
    +---------+

: J(2,2,.)
[symmetric]
       1   2
    +---------+
  1 |  .      |
  2 |  .   .  |
    +---------+

: J(2,3,4)
       1   2   3
    +-------------+
  1 |  4   4   4  |
  2 |  4   4   4  |
    +-------------+

: end
--------------------------------------------------------------------------------------
```

We also have a very large set of matrix functions available, e.g. a set of "solvers", i.e. mathematical functions that solve linear problems of the type $AX = B$, where both A and B are known matrices. The solution to this problem will be the inverse of A if we choose B to be an appropriate identity matrix and A is invertible. Solvers are more accurate and faster than actually inverting a matrix, so we should always prefer solving algorithms over inverting algorithms. In fact the Mata invert functions use corresponding solvers instead of actually inverting the matrix.

```
. mata
---------------------------------------- mata (type end to exit) ----------------------
: A = (10, 20 \ .2, .3)

: rank(A)
  2

: lusolve(A,I(2))
         1     2
    +-------------+
  1 |  -.3    20  |
  2 |   .2   -10  |
    +-------------+

: luinv(A)
         1     2
    +-------------+
  1 |  -.3    20  |
  2 |   .2   -10  |
    +-------------+

: end
--------------------------------------------------------------------------------------
```

One thing Mata has problems dealing with is "large" scale differences in the data. Before using Mata it is recommended that we bring our data to the same scale. Otherwise we might find ourselves in trouble. The reason is that the precision with which matrices are handled, or more precisely the precision at which Mata treats matrix elements as zero, depends on the values of the matrix. If there are large differences in the scale of the matrix values, small values might be considered zero.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: A = (.0000000000001, .0000000000002 \ .000000000000002, .000000000000003)

: rank(A)
  2

: luinv(A)
                  1              2
    +--------------------------------+
  1 |  -3.00000e+13    2.00000e+15  |
  2 |   2.00000e+13   -1.00000e+15  |
    +--------------------------------+

: A = (10, .0000000000002 \ .2, .000000000000003)

: rank(A)
  1

: luinv(A)
[symmetric]
       1   2
    +---------+
  1 |  .      |
  2 |  .   .  |
    +---------+

: end
--------------------------------------------------------------------------------------
```

The problem can be circumvented by either bringing the data to the same scale (a difference in scale up to 1,000 shouldn't be any problem) or by setting the tolerance by hand.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: A = (10, .0000000000002 \ .2, .000000000000003)

: rank(A, 1e-15)
  2

: luinv(A,  1e-15)
                  1              2
    +--------------------------------+
  1 |           -.3             20  |
  2 |   2.00000e+13   -1.00000e+15  |
    +--------------------------------+

: end
--------------------------------------------------------------------------------------
```

Choosing subsets of data is extremely easy and quick in Mata. We can refer to single elements the same way we are used to, by adding square brackets after an object and selecting row and column. We can also select ranges of observations, nonconsecutive observations as well as change the order of the elements. But careful, we always need to define objects before we refer to them.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: x[1,1]
  1

: x[1,2]
  2

: row = 1

: col = 2
```

```
: x[row,col]
  2

: A[1,1] = 2

: x[1,.] //a missing values implies all columns (or rows)
       1   2
    +---------+
  1 |  1   2  |
    +---------+

: B[1,1] = 1
                  <istmt>:  3301  subscript invalid
r(3301);

: B = (1,2,3 \ 4,5,6 \ 7,8,9)

: B[|2,2 \ .,.|]
       1   2
    +---------+
  1 |  5   6  |
  2 |  8   9  |
    +---------+

: row = (1 \ 3)

: col = (2 , 1)

: B[row,col]
       1   2
    +---------+
  1 |  2   1  |
  2 |  8   7  |
    +---------+

: end
--------------------------------------------------------------------------------
```

If we want to use logical conditions to form subsets of data, we need to use the `select()` function. So we could for example select all the rows of `B` where the first element of the row is smaller or equal than one. Or all those rows where the first element in the row is less than one.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: select(B, B[.,1] :> 1)
       1   2   3
    +-------------+
  1 |  4   5   6  |
  2 |  7   8   9  |
    +-------------+

: select(B, B[.,1] :<= 1)
       1   2   3
    +-------------+
  1 |  1   2   3  |
    +-------------+

: end
--------------------------------------------------------------------------------
```

Object types

So far we have not distinguished between different object types. But unlike Stata that allows us some leeway in using different objects, e.g. a local can take both string and numerical values and be used in evaluations of either. In Mata we have to be more

precise. A lot of functions only take certain objects as arguments and we need to define our objects accordingly. But the definition of an object is not always explicit. Before, we defined matrices without explicitly stating their type. Mata understood that when we generated a 2x2 matrix containing numerical values we want that matrix to be a "real matrix". There are two aspects to consider in the definition of objects, the values the object takes and the form of the object itself. A "real matrix" is a matrix (i.e. an object of dimension ixj) that takes "real" values, i.e. numerical values that are not complex.

element types	**organizational types**
transmorphic (any of the following)	matrix (ixj)
numeric (real or complex)	vector (ix1 or (1xj))
real	rowvector (1xj)
complex	colvector (ix1)
string	scalar (1x1)
pointer	
structure	

To check the type of our data we can use `mata describe`, the same we would do in Stata or `eltype()` and `orgtype()` to find out the individual element types of our objects. To define an object explicitly we just state its type for example `real matrix A`. In general interactive use that is probably not that useful. The advantage of being explicit in our declarations comes in when we start writing Mata functions. We can define our functions to have return values and be explicit about what they return, we can define all the objects at the beginning of a function, which results in Mata checking whether the later use corresponds to the definitions and lastly defining objects before their use let's Mata make our code more efficient, i.e. run faster.

We already encountered real and string matrices (my use of the term "matrix" usually entails vectors and scalars, which are just special cases of matrices) in our code. We usually don't have use for complex numbers, e.g. $c = sqrt(-1)$, but the other types can be very useful. Pointers and structures are something we will talk about later in this section, but a short summary of the two is that, pointers allow you to work with the address of an object, rather than the object itself. An address is a very short piece of information and stored using only a few bytes, while the object the address belongs to might be rather large. So parsing information between parts of your code does not have to involve all the data, but just the reference to where Mata can find the data. Structures have a different purpose. They basically allow us to collect several types of objects under one roof. Other programming languages allow similar features with lists or arrays.

Stata in Mata

We can interact with Stata from within Mata in several ways. To run a Stata command we simply use the `stata()` function which takes a string scalar as argument, i.e. a one-dimensional string which is just the whole command line we want to execute in Stata. In general functions in Mata that interact with Stata objects use the `st_` prefix, e.g. `st_local()` to access a local macro.

To read Stata data into Mata we can use two different functions: `st_view` and `st_data`. Both functions import numerical data, if we want to use string data we use `st_sview` and `st_sdata`. Note that if you use the command with the wrong type of data, Mata imports a column of missings.

The `st_data` command simply generates a copy of the Stata data as a Mata object. This has the advantage that changes you make in Mata do not affect the original data, but the disadvantage of having to use twice as much memory. The `st_view` command is a lot more memory efficient. Instead of copying the data into Mata it simply links to the data in Stata and accesses the Stata data whenever we call upon a matrix created with `st_view`. The advantage is that the memory use is negligible, but the disadvantage (or rather danger, than disadvantage) is that changes we make in Mata affect the data in Stata as well.

```
. use "H:\ECStata\PWT", clear

. mata
------------------------------------------------- mata (type end to exit) -----------
: st_view(allv=.,.,.)

: alld = st_data(.,.)
```

```
: mata describe

      # bytes   type                          name and extent
------------------------------------------------------------------------------
      685,440   real matrix                   alld[8568,10]
           44   real matrix                   allv[8568,10]
------------------------------------------------------------------------------

: allv[1,.]
          1      2      3      4      5      6      7      8      9     10
    +-------------------------------------------------------------------------+
  1 |     .      .   1950      .      .      .      .      .      .      .  |
    +-------------------------------------------------------------------------+

: alld[1,.]
          1      2      3      4      5      6      7      8      9     10
    +-------------------------------------------------------------------------+
  1 |     .      .   1950      .      .      .      .      .      .      .  |
    +-------------------------------------------------------------------------+

: stata("li in 1")

     +-------------------------------------------------------------------------+
     | country   countr~e   year   pop   cgdp   openk   kc   kg   ki   grgdpch |
     |-------------------------------------------------------------------------|
  1. |  Angola        AGO   1950     .      .       .    .    .    .         . |
     +-------------------------------------------------------------------------+

: end
------------------------------------------------------------------------------------------
```

Notice the difference in size, but also the different use of the two commands. While we don't assign the outcome of `st_view` to an object, we need to specify a matrix (or vector) that takes on the variable values from `st_data`. The matrix that points to our Stata data in the `st_view` command is provided within the its arguments, since we cannot refer to anything that does not exist, we need to define the matrix beforehand, or here we define it within the argument by using `ally=.`, i.e. generating an object that is empty.

The second and third argument of `st_view` and the first and second argument of `st_data` relate to the observations (rows) and variables (columns) that we wish to use in Mata, a missing value indicates that we want to use all observations and all variables. There are other arguments we can provide and we will see some of them in the later sections.

Sorting and permutations

Sorting data is a fairly time intensive task. In Mata we should avoid sorting and use permutations when possible. A permutation is simply the reordering of rows or columns of a matrix. We can achieve this quite easily by subsetting our matrices.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: stata("li in 1/2")

     +-------------------------------------------------------------------------+
     | country   countr~e   year   pop   cgdp   openk   kc   kg   ki   grgdpch |
     |-------------------------------------------------------------------------|
  1. |  Angola        AGO   1950     .      .       .    .    .    .         . |
  2. |  Angola        AGO   1951     .      .       .    .    .    .         . |
     +-------------------------------------------------------------------------+

: stata("cap gen id = _n")

: stata("cap drop country")

: stata("sort kc")

: st_view(order=.,.,("id"))
```

```
: stata("li in 1/2")

     +-------------------------------------------------------------------------+
  1. | countr~e | year |     pop |     cgdp |    openk |       kc |       kg |
     |      MRT | 1962 | 1030.98 | 185.9104 | 64.56567 | 14.30394 | 107.2764 |
     |-------------------------------------------------------------------------|
     |              ki        |          grgdpch        |          id        |
     |        10.16372        |        -3.424893        |        5419        |
     +-------------------------------------------------------------------------+

     +-------------------------------------------------------------------------+
  2. | countr~e | year |     pop |     cgdp |    openk |       kc |       kg |
     |      MRT | 1961 | 1010.69 | 183.5449 | 56.23458 | 14.85863 |  102.059 |
     |-------------------------------------------------------------------------|
     |              ki        |          grgdpch        |          id        |
     |        8.064412        |        -28.62559        |        5418        |
     +-------------------------------------------------------------------------+

: st_view(alld=.,.,.)

: alld[(1\2),.]
                  1              2              3              4              5
    +-------------------------------------------------------------------------+
  1 |             .           1962     1030.97998    185.9104462     64.5656662 |
  2 |             .           1961    1010.690002    183.5448914    56.23458099 |
    +-------------------------------------------------------------------------+
                  6              7              8              9             10
    ------------------------------------------------------------------------------+
  1     14.30393887    107.2763977    10.16372299   -3.424893379           5419  |
  2       14.858634    102.0589905    8.064412117   -28.62559128           5418  |
    ------------------------------------------------------------------------------+

: alld[.,.] = alld[invorder(order),.]

: alld[(1\2),.]
          1      2      3      4      5      6      7      8      9     10
    +-------------------------------------------------------------------------+
  1 |     .   1950      .      .      .      .      .      .      .      1  |
  2 |     .   1951      .      .      .      .      .      .      .      2  |
    +-------------------------------------------------------------------------+

: stata("li in 1/2")

     +-------------------------------------------------------------------+
     | countr~e   year   pop   cgdp   openk   kc   kg   ki   grgdpch   id |
     |-------------------------------------------------------------------|
  1. |      MRT   1950     .      .       .    .    .    .         .    1 |
  2. |      MRT   1951     .      .       .    .    .    .         .    2 |
     +-------------------------------------------------------------------+

: end
-------------------------------------------------------------------------------------------
```

What happened? We messed up our observations! The reason is that some of our variables are string variables, others are numeric. We only affect the sort order of the numerical variables when we permute the view, the string variables are not affected! So we need to be more careful when doing this. Let's see whether we can change the order differently.

But before we do this, note two things about our use of the permutation in the command line: `alld[.,.] = alld[invorder(order),.]`. The `invorder()` command changes the interpretation of the provided vector. While normally the number provided in the vector (e.g. 3) tells Stata which row (or column) of the matrix to display at the place of the number (e.g. the third row would be displayed), with `invorder()` the number tells Stata where to put the current observation (so the row would become the third row). The second peculiar aspect about the command line is that normally we ommit the subset when we want to refer to all rows and columns, but here we explicitly use them (`alld[.,.]`). The reason for this is that if we assign a view to another matrix in Mata (or even itself) it will result in Mata copying all the information in the view (i.e. all

the data) into Mata. The same thing we would have achieved when using `st-data`. But that has two implications, first the original data is not affected by the permutation and second we now need a lot more memory for the copy than we used for the view, exactly what we tried to avoid. So in order for the view to remain intact, we need to specify explicitly that we want to do something to its elements, i.e. the whole subset of its rows and columns. This way the view remains intact and the sort order of the original data is changed. Now let's see whether we can get the code right.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: stata("gen id = _n")

: stata("sort kc")

: stata("li in 1/2")

     +-------------------------------------------------------------------------+
  1. |    country | countr~e | year |     pop |     cgdp |    openk |       kc |
     | Mauritania |      MRT | 1962 | 1030.98 | 185.9104 | 64.56567 | 14.30394 |
     |-------------------------------------------------------------------------|
     |           kg     |           ki     |       grgdpch     |       id      |
     |     107.2764     |     10.16372     |     -3.424893     |     5419      |
     +-------------------------------------------------------------------------+

     +-------------------------------------------------------------------------+
  2. |    country | countr~e | year |     pop |     cgdp |    openk |       kc |
     | Mauritania |      MRT | 1961 | 1010.69 | 183.5449 | 56.23458 | 14.85863 |
     |-------------------------------------------------------------------------|
     |           kg     |           ki     |       grgdpch     |       id      |
     |      102.059     |     8.064412     |     -28.62559     |     5418      |
     +-------------------------------------------------------------------------+

: st_view(order=.,.,("id"))

: st_view(allv=.,.,.)

: st_view(allnum=.,.,select(st_viewvars(allv), st_viewvars(allv):!=st_varindex("id")))

: st_sview(allstr=.,.,.)

: allnum[(1\2),.]
                 1              2              3              4              5
    +-------------------------------------------------------------------------------
  1 |             .              .           1962     1030.97998    185.9104462
  2 |             .              .           1961    1010.690002    183.5448914
    +-------------------------------------------------------------------------------
                 6              7              8              9             10
     ---------------------------------------------------------------------------+
  1      64.5656662    14.30393887    107.2763977    10.16372299   -3.424893379  |
  2      56.23458099      14.858634    102.0589905    8.064412117   -28.62559128  |
     ---------------------------------------------------------------------------+

: allstr[(1\2),.]
                1            2            3            4            5            6
    +-------------------------------------------------------------------------------
  1 |  Mauritania          MRT
  2 |  Mauritania          MRT
    +-------------------------------------------------------------------------------
                7            8            9           10           11
     ------------------------------------------------------------------+
  1                                                                    |
  2                                                                    |
     ------------------------------------------------------------------+

: allnum[.,.] = allnum[invorder(order),.]

: allstr[.,.] = allstr[invorder(order),.]
```

```
: order[.,.] = order[invorder(order),.]

: stata("li in 1/2")

     +-------------------------------------------------------------------------------+
     | country   countr~e   year   pop   cgdp   openk   kc   kg   ki   grgdpch   id |
     |-------------------------------------------------------------------------------|
  1. |  Angola        AGO   1950     .      .       .    .    .    .         .    1 |
  2. |  Angola        AGO   1951     .      .       .    .    .    .         .    2 |
     +-------------------------------------------------------------------------------+

: end
--------------------------------------------------------------------------------
```

We use `st_view` and `st_sview` to load all the numerical and string variables into Mata. Though we have to be careful when loading the numerical data we would automatically include the variable we want to use for the permutation. But if we permute the `allnum` view first, this would change the order of the variable that we use for the permutation as well. So we exclude the variable when generating our `allnum` view. Note that it would „probably“ have been easier to just sort the string variables first and then the numerical variables.

To select all variables except the permutation indicator we use the select command. The arguments we provide are based on `st_viewvars` and `st_varindex`. The former generates a vector with the index of each variable in the view `allv` as elements, i.e. if the first variable `allv` points to is the first variable in our data set, the first element in the vector will be 1. Now `st_varindex` tells Stata what is the position of a certain variable, i.e. its index. And so the `select` command returns a row vector including all variable indizes except for the index of our order variable.

```
st_view(allnum=.,.,select(st_viewvars(allv), st_viewvars(allv):!=st_varindex("id")))
```

Notice that we need to use the elementwise logical operator `:!=` otherwise we will have a conformability error.

Mata functions

Before we define a proper function let us up the example we want to use interactively. As a matrix language Mata lends itself easily to estimating OLS coefficients. So this is our first example, a basic OLS regression without any extras.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: stata("cap gen cons = 1")

: st_view(allv=.,.,("grgdpch", "pop", "openk", "kc", "kg", "ki", "cons"), 0)

: st_subview(y=., allv, ., 1)

: st_subview(X=., allv, ., (2..7))

: mata describe

      # bytes   type                          name and extent
-------------------------------------------------------------------------------
       22,508   real matrix                   X[5621,6]
       22,512   real matrix                   allv[5621,7]
       22,488   real colvector                y[5621]
-------------------------------------------------------------------------------

: stata("li countryisocode year grgdpch pop openk kc kg in 1/13")

     +-------------------------------------------------------------------+
     | countr~e   year    grgdpch       pop      openk         kc      kg |
     |-------------------------------------------------------------------|
  1. |      AGO   1950          .         .          .          .       . |
  2. |      AGO   1951          .         .          .          .       . |
[...]
 10. |      AGO   1959          .         .          .          .       . |
     |-------------------------------------------------------------------|
```

```
 11. |      AGO   1960          .      4816   31.33579    72.8689    11.5463 |
 12. |      AGO   1961   5.143075   4884.19   32.50923   68.89259   11.47156 |
 13. |      AGO   1962   .6126294   4955.35   34.65629   69.76397   12.05302 |
     +-------------------------------------------------------------------------+

: X[1,.]
                 1             2             3             4             5
    +-------------------------------------------------------------------------+
  1 |  4884.189941   32.50923157   68.89259338   11.47156429   6.134659767   |
    +-------------------------------------------------------------------------+
                 6
     ---------------+
  1              1  |
     ---------------+

: y[1,.]
  5.143075466

: b = luinv(cross(X,X))*X'*y

: b
                  1
    +----------------+
  1 |   2.05561e-06  |
  2 |   .0028080242  |
  3 |   -.020235871  |
  4 |  -.0231073196  |
  5 |   .0918375783  |
  6 |   2.311029817  |
    +----------------+

: stata("reg grgdpch pop openk kc kg ki")

      Source |       SS       df       MS              Number of obs =    5621
-------------+------------------------------           F(  5,  5615) =   34.42
       Model |  7451.82653     5  1490.36531           Prob > F      =  0.0000
    Residual |   243152.41  5615  43.3040802           R-squared     =  0.0297
-------------+------------------------------           Adj R-squared =  0.0289
       Total |  250604.237  5620  44.5915012           Root MSE      =  6.5806

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         pop |   2.06e-06   8.19e-07     2.51   0.012     4.51e-07    3.66e-06
       openk |    .002808    .002011     1.40   0.163    -.0011343    .0067503
          kc |  -.0202359   .0051074    -3.96   0.000    -.0302483   -.0102235
          kg |  -.0231073   .0071682    -3.22   0.001    -.0371597   -.0090549
          ki |   .0918376   .0105844     8.68   0.000      .071088    .1125872
       _cons |    2.31103   .4853361     4.76   0.000     1.359583    3.262476
------------------------------------------------------------------------------

: end
--------------------------------------------------------------------------------
```

We use Stata to generate the variables we need (including an intercept) and use a view to select all the data. Notice that we do this in a two-step procedure, first we create a view of all the data we will need in our regression using an additional argument after selecting the variables we want to use. The "0" at the end tells Mata to omit those observations with at least one missing value for any of the observations. Only after we do this we create to views, based on the view that entails all our observations, which we will use for the regression. We have to be very careful when loading the data, if we created views for our dependent and independent variables separately, the number of rows (i.e. observations) that we use might differ, Here for example we have information on all variables for Angola in 1960 except for the growth rate, our independent variable. To avoid this problem and to keep the observation structure intact we need to use the option to omit missings on all variables at the same time.

To estimate the coefficient vector we can simply use matrix algebra or we can use some of Mata's matrix functions. The `cross()` command generates the inner product of two matrices, which we could easily achieve by just writing `X'*X`. The

advantage of using the `cross()` command is that it saves memory (especially in combination with views), it automatically takes account of missings (though we should do this beforehand!) and it accounts for the symmetric nature of the inner product of a matrix with itself (which again gives us an efficiency gain). So we should prefer `cross()` over a notation like `X'*y`.

Interactive use is one thing, but this is a typical example where we would be better suited with writing a function to perform the regression for us. Later in this section we will use this example to write our own regression command in Stata, but at this stage we will stick to a simple function.

```
mata:
stata("cap mata mata drop MyReg()")
real vector MyReg(string scalar varlist)
{
       real vector b   //coefficient vector
       real matrix y   //dependent variable
       real matrix X   //regressors
       (void) st_addvar("float", t_name=st_tempname())
       st_store(., t_name, J(c("N"),1,1))
       st_view(allv=., ., (tokens(varlist), t_name), 0)
       st_subview(y=., allv, ., 1)
       st_subview(X=., allv, ., (2..cols(allv)))
       b = cross(luinv(cross(X,X)),cross(X,y))
       return(b)
}
end
```

We first make sure that the function is not already defined and drop it (not very elegantly by calling upon Stata). We define a function by just typing a name, followed by parenthes and wiggly parentheses to start and finish the function. Now not specifying anything in front of your function's name is not good programming practice (though it works). What we should do is specify the return value of the function. If the function does not return anything it is of the type `void`, but here our function returns a vector (the coefficient estimates) and therefore its type is `real vector`. Within the parentheses we provide the definition of the input that the function needs, here we only use a string that contains the variables we will use in our regression. We define all the objects we will use and add a temporary variable to Stata (the constant). This is a two step procedure. First we define a new variable in Stata using `(void) st_addvar("float", t_name=st_tempname())`. The `(void)` preceding the command surpresses the output that the command would generate (it returns the index of the newly added variable). We specify the precision of the new variable and the variable name. The name we choose is generated by Stata using the `st_tempname()` function. This function generates a unique name for the temporary variable.

In the second step we store a one for each observation to construct our intercept. The `st_store` function replaces all values (the first argument is missing) of a variable which is named as second argument with the values of a vector (the third argument). We do not know the number of observations, but we can access it using the `c()` – lower case! – function, which gives us access to a variety of current Stata objects. To make a (row) vector of variable names out of our string, we use `tokens()` and add the newly created temporary variable at the end of the row vector. The final step in our program is to return the object of interest, our vector of coefficient estimates using `return(b)`.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: MyReg("grgdpch pop openk kc kg ki")
                  1
    +----------------+
  1 |   2.05561e-06  |
  2 |   .0028080242  |
  3 |   -.020235871  |
  4 |  -.0231073196  |
  5 |   .0918375783  |
  6 |   2.311029817  |
    +----------------+

: end
--------------------------------------------------------------------------------------
```

Looping and branching

To select certain sets of our Mata data we can (and should) use subsetting, but we often need programming routines like the ones introduced in the previous sections. We can use `do/while` loops similar to the `while` loops we used before. As an alternative

Mata offers the `for` loop.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: i=0

: do {
>         i++
> } while(i<=3)

: i
  4

: i=0

: while(i<=3) {
>         i++
> }

: i
  4

: i=0

: for(;i<=3;) {
>         i++
> }

: i
  4

: for(i=0;i<=3;i++){
> }

: i
  4

: end
--------------------------------------------------------------------------------
```

The `for` command can be used with only the logical condition as an argument, but then we have to initialize and change the argument by hand, or it can be called wit the initialization and change of the counter in the declaration.

For branching we have again a combination of if and else, but in Mata also a handy shorthand notation that combines the two. We provide a logical condition, and separate the behavior for a true and a false statement using a question mark and a colon.

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: if(isreal(i)){
>         printf("i is a real number")
> }
> else{
>         printf("i is not a real number")
> }
i is a real number
: (i < 5) ? "less than 5" : "5 or more"
  less than 5

:
: end
--------------------------------------------------------------------------------
```

Using structures or pointers

In Stata it is quite easy to loop over variables, especially when we construct them in clever ways (e.g. using the same prefix and

numbers as suffix). In Mata this is not so easy. There are two types of formats that can help us to emulate Stata's behavior (though their usefulness extends far beyond using them in loops). The two types are structures and pointers. As mentioned earlier, pointers save the address of an object and circumvent passing the whole object to functions.

```
mata
p = J(1,4, NULL)
A = (1,2 \ 3,4)
B = (5,6 \ 7,8)
C = (8,7 \ 6,5)
D = (4,3 \ 2,1)
p = (&A \ &B \ &C \ &D)
p
for(i=1; i<=4; i++) {
	*p[i]
}
end
```

We define a pointer (column) vector and fill it with the addresses of four matrices. The address of an object is given by an ampersand (`&`) followed by the name of the object. When we look at the content of `p`, we see the memory address where the four matrices are saved. To get the content back we need to use the asterisk (`*`) prefix. The pointer is a very efficient way to refer to an object, we can define pointers for basically all objects in Mata, even functions themselves.

Instead of using pointers we can define structures of objects. Structures are containers of objects summarizing them under one heading. We have to define a structure and its contents before we can use it. Here our structure only has one element, a matrix `X`, but in theory we could have it have as many elements as we want. After defining the content of our structure we define its size. For us this means simply how many matrices we want to have. To refer to the content of a structure we use a dot (`.`) to indicate that we want to refer to an element of our structure and then name the element of the structure we want to use.

```
struct MyMatrices {
	real matrix X
}
void MyFunction()
{
	struct MyMatrices vector MyS
	MyS = MyMatrices(4)
	MyS[1].X = (1,2 \ 3,4)
	MyS[2].X = (5,6 \ 7,8)
	MyS[3].X = (8,7 \ 6,5)
	MyS[4].X = (4,3 \ 2,1)
	i = 1
	while(i<=4) {
		MyS[i++].X
	}
}
MyFunction()
```

Mata's optimize command

We had a detailed look on optimization with Stata's `ml` command, with the introduction of Mata another maximization (or minimization for that matter) routine became available. We need to set up the problem similar to the `ml` command. The methods `d0`, `d1` and `d2` are available in Mata, but we also have three new methods (`v0` to `v2`). The new methods do not require us to aggregate the likelihood, gradient or Hessian ourselves, but it suffices to provide matrices in the optimization routine. To exemplify the use of the optimiziation routine in Mata we will program OLS by hand. OLS is equivalent to the maximum likelihood estimator of a linear model with normally distributed errors. The likelihood and log-likelihood are given by the following formulas.

$$L = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left(-\frac{(y_i - X_i\beta)^2}{2\sigma^2}\right)$$

$$\ln(\mathrm{L}) = l = \sum_{i=1}^{N} -\frac{1}{2}\ln(2\pi\sigma^2) + \sum_{i=1}^{N} -\frac{1}{2\sigma^2}(y_i - X_i\beta)^2$$

But before we call upon Mata for the optimization we should clean our sample in Stata.

```
. reg grgdpch pop openk kc kg ki

      Source |       SS       df       MS              Number of obs =    5621
-------------+------------------------------           F(  5,  5615) =   34.42
       Model |  7451.82653     5  1490.36531           Prob > F      =  0.0000
    Residual |   243152.41  5615  43.3040802           R-squared     =  0.0297
-------------+------------------------------           Adj R-squared =  0.0289
       Total |  250604.237  5620  44.5915012           Root MSE      =  6.5806

------------------------------------------------------------------------------
     grgdpch |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         pop |   2.06e-06   8.19e-07     2.51   0.012     4.51e-07    3.66e-06
       openk |    .002808    .002011     1.40   0.163    -.0011343    .0067503
          kc |  -.0202359   .0051074    -3.96   0.000    -.0302483   -.0102235
          kg |  -.0231073   .0071682    -3.22   0.001    -.0371597   -.0090549
          ki |   .0918376   .0105844     8.68   0.000      .071088    .1125872
       _cons |    2.31103   .4853361     4.76   0.000     1.359583    3.262476
------------------------------------------------------------------------------

. keep if e(sample)
(2947 observations deleted)

. keep grgdpch pop openk kc kg ki

. gen cons = 1
```

Now we have only observations with nonmissing information for all variables as well as only the variables of interest left in the sample. The actual optimization process involves similar steps as in Stata. First we need to define a function that takes on the value of the likelihood (or rather the criterion function, it does not have to be a likelihood). For `d/v0` this suffices (derivatives are evaluated numerically). Wwhen using `d/v1` we have to additionally specify the gradient and for `d/v2` also the matrix of second derivaties. The other no necessary ingredients are the `todo` argument, which indicates the maximization method we choose and the parameter vector (here we choose `b`, but it could basically be called anything, like `lnf` as well). What is important is the order of the arguments. If we want to introduce additional arguments, like a dependent and independent variable for example, we have to tell Mata about this (and put them at the right spot, right after the parameter vector). After defining the functional form we need to define a placeholder for our maximization problem. The command to do this is `optimize_init`. The placeholder is used in all following functions to refer to our maximization problem. The object `s` that we define is basically a structure that takes on the default values for a maximization problem in Mata, which we subsequently change to adapt them to our objective.

The first thing we add to `s` is the function that entails the functional form of our optimization problem. Notice that we do not pass the function directly but rather the address in memory. We then choose the optimization method, here `v0` to avoid having to sum the likelihood ourselves. We add the two data matrices to the maximization problem by telling Mata their position in the optimization problem and which matrices hold that information and lastly, before starting the optimization, we define the initial values for our parameter estimates (here I set them all equal to one).

```
. mata
------------------------------------------------- mata (type end to exit) -----------
: st_view(y=., ., "grgdpch")

: st_view(X=., ., ("pop", "openk", "kc", "kg", "ki", "cons"))

: void maxme(todo,b,y,X,lnf,g,H)
> {
>       lnf = -.5*ln(2*pi()*b[cols(X)+1]) :- .5*(1/b[cols(X)+1])*(y-X*b[1..cols(X)]'):^2
> }
note: argument todo unused
note: argument g unused
note: argument H unused

: s=optimize_init()

: optimize_init_evaluator(s,&maxme())

: optimize_init_evaluatortype(s,"v0")
```

```
: optimize_init_argument(s, 1, y)

: optimize_init_argument(s, 2, X)

: optimize_init_params(s, J(1,cols(X)+1,1))

: b = optimize(s)
Iteration 0:  f(p) = -3.683e+13  (not concave)
[...]
Iteration 23:  f(p) =  -18563.51

: b[1..cols(X)]
                  1              2              3              4              5
    +-------------------------------------------------------------------------------
  1 |   2.05561e-06    .0028080241   -.0202358709   -.0231073195    .0918375786
    +-------------------------------------------------------------------------------
                  6
     ----------------+
  1     2.311029812  |
     ----------------+

: sqrt(b[cols(X)+1])
  6.577070774

: end
--------------------------------------------------------------------------------------
```

Using Mata for command programming

Now we can get back to our OLS example and write our own command. The way we set this up is to use a Stata program to deal with the call of the code, missing values and other housekeeping and then pass the problem on to Mata which does the calculations and returns our results. To make the problem a bit more challenging, we will program an OLS estimator that reports heteroscedasticity robust standard errors by default and allows for clustered standard errors. First let's have a look at the Stata program part:

```
cap prog drop MyRegression
prog def MyRegression, eclass
//OLS regression with robust or clustered SEs
syntax varlist [if] [in][, cluster(varname)]
       marksample touse
       tempvar cons clustvar
       qui gen `cons' = 1
       if "`cluster'" != "" qui egen `clustvar' = group(`cluster') if `touse'
       else qui gen `clustvar' = 1 if `touse'
       mata: MyMataRegression("`varlist' `cons'", "`touse'", "`clustvar'")
       ereturn local cmd "MyRegression"
       ereturn display
end
```

We use the syntax command to check whether the command call provided all the necessary ingredients and make use of `marksample` to select the sample for our regression (remember `marksample` checks for missing values and accounts for subset qualifiers). We generate an intercept term and a cluster indicator and call the Mata function to run the estimation. The final step of our (or any) estimation command is to return the command that was issued, as explained before, postestimation commands in Stata check whether the `e(cmd)` local contains anything, if it doesn't the postestimation command will exit with an error.

We want our Mata function to estimate more than just the coefficient vector, so we choose to return results directly to Stata rather than using return values. The function takes three arguments, the variables we use (dependent variable first), a sample selection indicator and the cluster indicator. Again we define all the matrices we will use at the beginning and start by loading the data into a view. We create our dependent variable and our regressor matrix from the initial view and estimate our coefficient, as we did before.

The alterations come in when we estimate the variance matrix of our coefficient estimates. First we have to distinguish between robust and clustered standard error estimation. The difference is twofold, first, the "butter" part of the sandwich estimator differs and second the small sample correction is different as well. We estimate the variance as

$$\hat{V} = S_t(X'X)^{-1}B_t(X'X)^{-1} \quad t \in \{robust, cluster\}$$

$$B_r = \sum_{i=1}^{N} e_i^2 X_i' X_i \text{ and } S_r = \frac{N}{N-k}$$

$$B_c = \sum_{j=1}^{C} X_j' e_j e_j' X_j \text{ and } S_r = \frac{N-1}{N-k}\frac{C}{C-1}$$

Where N is the number of observations, C the number of clusters and k the number of estimated parameters (including the intercept). The last step is returning the estimation results to Stata, we post some of them directly and others using the `ereturn` command in Stata.

```
mata:
void MyMataRegression(string scalar varlist, ///
                      string scalar touse,   ///
                      string scalar cluster)
{
       real vector b   //coefficient vector
       real matrix V   //variance matrix
       real scalar i   //counter
       //load the data
       st_view(allv=., ., tokens(varlist), touse)
       varname=tokens(varlist)[1,1]
       st_subview(y=., allv, ., 1)
       st_subview(X=., allv, ., (2..cols(allv)))
       st_view(C=., ., cluster, touse)
       //coefficient estimate
       b = cross(luinv(cross(X,X)),cross(X,y))
       //robust butter
       if (colmax(C[.,1]) == 1) {
             V = cross(X,((y-X*b):^2), X)
       }
       //cluster butter
       else {
             V = J(rows(b), rows(b), 0)
             for(i=1; i<=colmax(C[.,1]); i++) {
                   st_subview(x_j=., select(X, C:==i), ., .)
                   st_subview(y_j=., select(y, C:==i), ., .)
                   V = V + cross(cross(x_j, (y_j-x_j*b))', cross(x_j, (y_j-x_j*b))')
             }
       }
       //Put bread and butter together with Stata ssc
       V = (colmax(C[.,1]) == 1 ? (rows(y))/(rows(y)-rows(b)) :                  ///
             (rows(y)-1)/(rows(y)-rows(b))*colmax(C[.,1])/(colmax(C[.,1])-1)) * ///
             invsym(cross(X,X)) * V * invsym(cross(X,X))
       //return everything in ereturn
       st_local("b", bname=st_tempname())
       st_matrix(bname, b')
       st_local("V", Vname=st_tempname())
       st_matrix(Vname, V)
       rcnames = ( J(rows(b), 1, ""), tokens(subinstr(varlist, varname, ""))' )
       st_matrixcolstripe(bname, rcnames)
       st_matrixcolstripe(Vname, rcnames)
       st_matrixrowstripe(Vname, rcnames)
       stata("ereturn post " + bname + " " + Vname + ", esample(`touse')  depname(" ///
                   + varname + ") dof(" + strofreal( i<. ? colmax(C[.,1])-1 :     ///
                   rows(y) - rows(b) ) + ") obs(" + strofreal(rows(y)) + ")")
       st_numscalar("e(df_m)", rows(b)-1)
}
end
```

Using `ereturn post` requires some additional work. First we need to create matrices of our coefficient estimates and the variance estimate to pass them to ereturn. So we imitate the process of creating a temporary variable by creating a local that holds the value of a temporary object, which we then define to be the coefficient vector (or rather the transpose thereof, since Stata expects the estimates vector to be a row vector). We repeat the procedure for the variance matrix. The last step before posting our

results is to change the row and column names of our matrices. This is done with the stripe commands which take a matrix as argument that has the equation name in the first row and the variable name in the second.

Note that this is not really good programming style, we should split the tasks into individual functions, i.e. estimating the parameters, calculating the variance etc. these are nice separate tasks that we could easily generalize to work with other types of commands.