A guide to Stata

Florian Chávez Juárez*

Version 1.7 - May 18, 2015

Contents

*Centro de Investigración y Docencia Económicas (CIDE), Mexico City, Mexico. florian@chavezjuarez.com

1 Introduction

This document has mainly two purposes: first, it should help new users to **get started** with Stata® and second, it should serve more experienced users as a **look-up document**. In order to comply with the first goal of the document, I start with a general introduction to the software package and introduce then chapter by chapter more complicated notions in order to familiarize the reader with the software first and then introduce him to the possibilities Stata® offers. The second goal should be achieved by the use of a clear structure and an extensively detailed index in the end of the document. Like software development, this document will never achieve a final version and comments and suggestions are always welcome. Even though I refer to some econometric models, this document is **NOT** a reference for econometric analysis. The reader is supposed to understand the models I present here and know how and when to use them.
The document is under constant review and subject to changes and extensions. Please check for updated versions frequently. Please report all errors to florian@chavezjuarez.com.

Check your version online

Click here or go to http://www.econ.chavezjuarez.com/vcheck.php?i=stata&v=1.7 to check if you have the newest version.
This document is **freeware**. You can find the newest version online at http://econ.chavezjuarez.com/stata.php for free.

1.1 Why to use Stata?

Stata® is one of the most used software packages in applied econometrics, since it is fast, flexible and relatively easy to understand. In addition to the very long list of included packages to run econometric analysis, Stata® benefits from a large user base and high quality user written commands that can be installed and used very easily. For almost any econometric application that can be found in the literature, there is a package. See section 10 for an explanation on how to add user written packages to Stata® .
Besides the wide-ranging possibilities in terms of econometric analysis, Stata® has, in my opinion, two very important advantages. First, it is not only a statistical package, but also a data-handling package. Merging, manipulating, extracting etc. of databases is very easy and very fast in Stata® . A second plus of the software is the possibility to interact with other software packages, especially with LaTeX , allowing to produce directly result displays without a need to copy them to the LaTeX .

1.2 Alternative tutorials

There are many tutorials about Stata® available in the internet. Definitely the most complete way to learn about Stata® is the Stata® -help, integrated in your program or online at http://www.stata.com/help.cgi?contents. Other good addresses are:

- http://www.econ.uiuc.edu/~econ472/tutorial1.html
- http://www.cpc.unc.edu/research/tools/data_analysis/statatutorial/index.html
- http://data.princeton.edu/stata/ (Very nice tutorial)
- http://www.ats.ucla.edu/stat/stata/

2 Interface, components and data structure

2.1 Interface

The Stata® interface might be somewhat scary at first glance, but it becomes very useful once the user understands it well. Figure 1 represents a typical setting of the interface, where the big black part is the output panel of Stata® , all results will be displayed there[1]. On the left side you have the list of variables of the currently loaded database (see 3.3). Below the output panel, you will find the command line, which should be used only under some circumstances I will present afterwards. Below

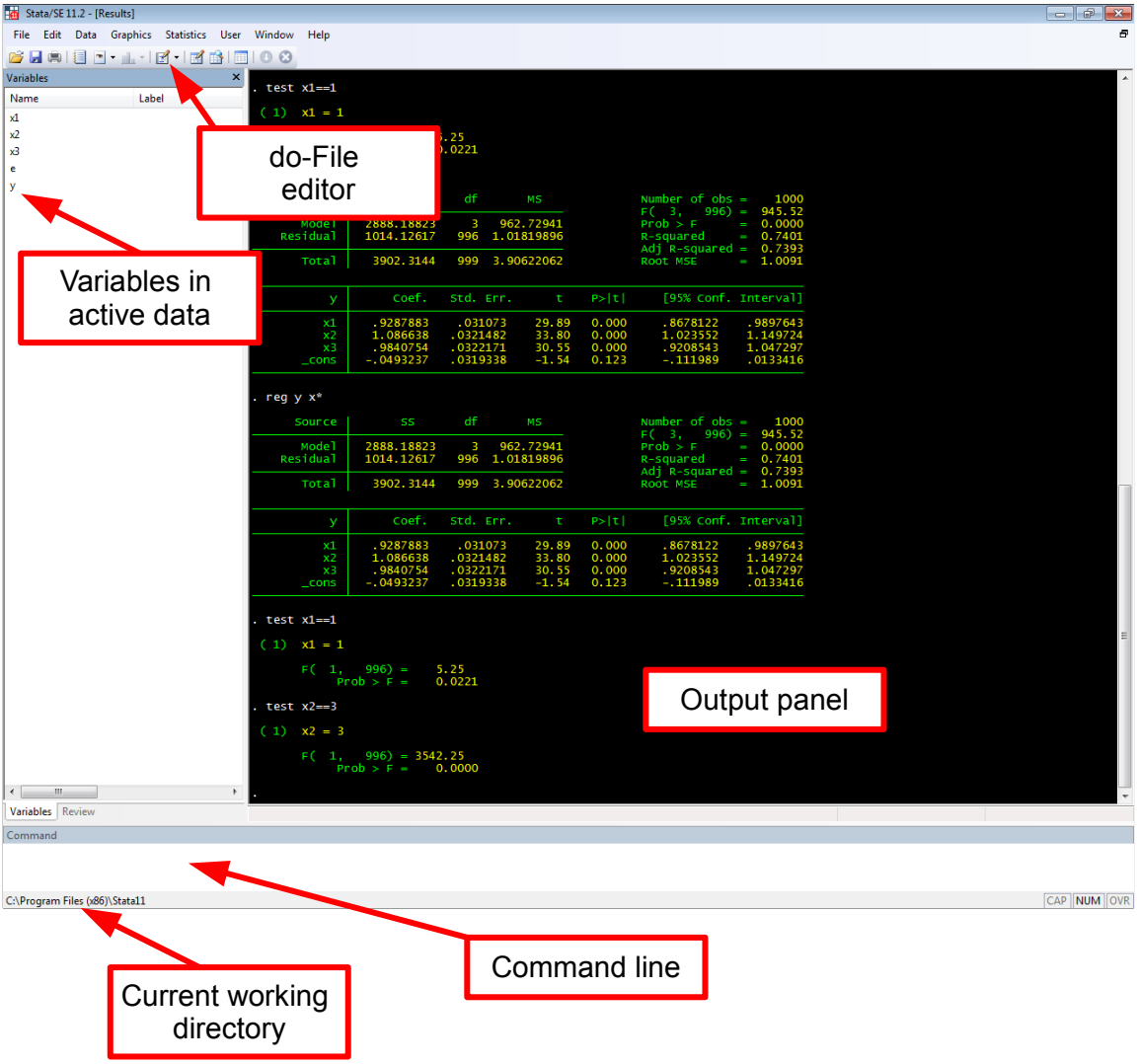


Figure 1: Stata Interface (Version 11.2)

the command line you see the current working directory, which is by default always the directory where Stata® is installed. Among the button in the menu, you find the do-file editor, which will probably be the most used button. The do-file editor is a simple text-editor with syntax highlighting

[1] In the new versions of Stata, the black screen is actually white and the results in black. Personally I prefer the black version (you can activate it under Edit → Preference → General preferences → Color scheme: Classic.) It looks a little bit more old-fashioned but highlights very well the different results.

(since Version 11). The remaining parts of the interface will be introduced later on, but they will not play a crucial role in the way I suggest to use Stata® .

2.2 Command based

Unlike other statistical packages, Stata® is mostly command based and the use of user interfaces is relatively limited. Generally, all commands could be initiated by the use of the mouse and user interfaces, however, it is not an efficient way to do things. My suggestion is to take the time to learn correctly the syntax-based use of Stata® and to work exclusively with the do-files. Do-files are like m-files in Matlab or *Syntax-files* is SPSS and allow to write down a series of commands and programming code. Using do-files to do all - starting from loading the database, carrying out the analysis and storing the results - permits to save time and avoid errors. Moreover, the results are easily reproducible by you and other researchers.
Normally, every line in the do-file corresponds to one command, which is generally what you want and avoids the need of finishing every line with a special character like in many programming languages. However, Stata® offers also the possibility to change this temporarily or permanently if you like it. This feature is useful when you have an extremely long command which does not fit the screen at all (see 3.1.1 for more details).

2.3 Help files

All official Stata® and most of the user written commands are accompanied by a so-called help file. These files generally include a description of what the command is performing, an explanation of the syntax including mandatory and optional arguments and some examples.
To display the help file just type into the command line

```
help commandname
```

where `commandname` is simply the name of the command you would like to see the corresponding help file.
At the beginning of a help file, the syntax is explained. The bold elements refer to the name of the command and in some cases to the options. The underlined part of the word is the part you have to write in the command, the remaining characters need not be written. For instance, the command for a simple OLS regression is `regress`, however it is sufficient to write `reg`. All elements in brackets "[ ]" refer to optional elements. The syntax is generally divided into two parts by a comma (,) where all behind the comma are options.
Some elements are present in almost all commands:

depvar	The dependent variable
indepvar	The independent variables
varlist	A list of variables, just write one after the other
`[if]`	*If* condition you might include to limit the command to a sample
`[in]`	*In* condition you might include to limit the command to a sample
`[weight]`	Possibility to include a variable with sample weights

2.4 File types

A Stata® -project is generally composed of at least two files. The database is where your data is stored and has the file type extension `.dta`. The do-files containing the commands to be performed

in the analysis end with the extension `.do` and is a simple text file you can also manipulate in other text editors like Notepad or Notepad++.
Besides these two main file types there exists the help files (ending with `.hlp`) and the ado-files ending with `.ado`. Both are used especially when you write your own commands, a topic I will discuss in section 11.
Finally Stata® is not limited to these file types, you can theoretically read and write any kind of files. I will discuss a couple of examples like the output of graphics, the automatic generation of LaTeX files or the import of SPSS and SAS data (section 9)
The following table displays the most common file extensions used in Stata® :

Extension	**Description**
.dta	database file in Stata format
.do	Executable Stata syntax file
.ado	Stata command file (each command is written in an ado-file)
.hlp	Stata help file (has the same name as command)

2.5 Data types: database and variables

When thinking about statistical analysis the main data type is obviously the database containing all the information you want to analyze. This kind of data is stored in the `.dta` files and can be loaded (see section 3.3) with the command `use`. However, there are other data types used in Stata® as well. When you write a do-file, you probably need from time to time saving some information in a variable to use it again later on. It would be very inefficient to save such information to the database, thus you can store them in other places.

type	**supported format**	Way to define it	Way to invoke value
local	numeric and string	`local name=55`	`` `name' ``
global	numeric and string	`global name=55`	`$name`
scalar	numeric	`scalar name=55`	`name`
matrix	numeric matrix	see `matrix`	

The first three types are generally interchangeable, however I suggest to use the *local* whenever possible. The *local* is preferable to the *global* since it remains set only until the end of the do-file, while the *globals* are not cleared automatically in the end. This feature of the globals can be useful in some cases, however, there is always a risk of having a global defined in a previous do-file and affecting later on when it should not. Moreover, it could be argued that the *locals* have a clearer invoking command than the scalars, since they need to be in between two very specific apostrophes always. On the other hand, scalars have the advantage that we can display easily all stored scalar values at once using the command `scalar list`.

Globals

To define a global and giving it the value of 888 you can just write

```
global myname=888
```

. If you would like to define the global based on the results of another command which stores the value for instance as *r-class*-scalars, you might use

```
global N=r(N)
```

for instance. To use them later on you have to use the dollar-symbol before the name:

```
display "The total number of observations is:  $N"
```

Locals

The use of *locals* is essentially identical to the use of *globals* with a small difference in the way you invoke the variable. Before the name, you have to write a single opening apostrophe (ASCII symbol 96) and right after the name a simple apostrophe (ASCII symbol 39). The following example defines first a numeric local containing the age of a person and then a string local with the name. In the third line, a small text with the information will be displayed:

```
local age=39
local name="Peter"
di "The age of ‘name’ is ‘age’ years"
```

This will produce the text *The age of Peter is 39 years.*

Scalars

The *scalars* can only take numerical values and the way to invoke them is at a first glance easier, since you just write their name. However, this might lead to confusions with the variable names, a potentially problematic issue.

2.6 Data type and format

The variable format is an important issue in Stata® like in all application dealing with data of this type. The goal is to use the least possible amount of memory for the variables. A particular issue in Stata® is that we have to distinguish between the actual format of the variable when it is saved and the output format, thus the way values are displayed. Hence, the *data type* refers to the way the information is stored in the database and *data format* is the way the information is displayed.

2.6.1 Data type

Data can be either numerical or string, which is probably the most basic distinction.

Numerical types

The following table gives a short overview of the different numerical data types in Stata® :

Storage type	Minimum	Maximum	Closest to 0 (but $\neq 0$)	bytes
byte	-127	100	± 1	1
int	-32’767	32’740	± 1	2
long	-2’147’483’647	2’147’483’620	± 1	4
float	-1.701×10^{38}	1.701×10^{38}	$\pm 10^{-38}$	4
double	-8.988×10^{307}	8.988×10^{307}	$\pm 10^{-323}$	8

String types

Regarding string variables it becomes very easy, since they are simply going from *str1* being one letter up to *str244* containing a maximum of 244 letters. The format is always at least as high as the largest entry in the database. Assume that in all but one observation, we do not have a string and in the one observation, we have 66 symbols. In this case, 66 bytes of memory are needed for every observation, hence it is worthy to avoid unnecessary string variables in a database.

How to choose the best type? This relative technical information on the storage type and used bytes might frighten a bit, but in practice this is hardly a topic, since there is a wonderful command called `compress` which analyzes every variable and converts it to the best storage format.

2.6.2 `compress` - Optimize the memory use of your database

The command `compress` is a very powerful tool to reduce the size of your database to the minimum needed. The command simply analyzes each variable and sets its format to the best fitting.

Hint 1. When you work with large datasets, combine several of them and create new variables, it is very recommended to include the command `compress` just before saving your database in order to avoid wasting memory for nothing.

2.6.3 Further compressing by using a ZIP-file

Especially when working with micro data the databases can become very large. If you have to send them to other people, you might want to consider putting them into a zip-file, which you can do under Windows in the Explorer for instance. The size reduction you can achieve depends a lot on your data, but it is not uncommon to reduce the size by up to 90%!! Such high compressions are especially possible when you have a lot of missing values or even more when many long string variables are empty.

2.6.4 Display format

Besides the format in which a value is stored on your disk, Stata® has a display format, which can differ from the first. The percentage sign is used to declare formats. For numerical values, there are mainly two types: the general (g) and the fixed (f) format. The general format depends on the number, while the fixed format has for instance a fixed number of decimals, no matter what the number is. For instance:

Complete number	%6.2g	%6.2f
99.1234	99	99.12
.0159	.016	0.02
500	500	500.00
.001	.001	0.00

Stata® supports also date and time formats. See the help file for details.
Knowing these format types can be very useful in commands like `est table`, `estout` or `tabout`

2.6.5 `format` - Change the display format of a variable

You can also assign to a variable a certain display format. For instance, if you wish to see the income always with two decimals, you can use the command

```
format income %6.2f
```

2.7 Missing values

Like in any other statistical package, missing values have a special code in Stata® . The normal coding of a missing variable is a dot (.), for instance you might want to set a value to missing

```
replace age=.  if age<0
```

if the indicated value is not plausible. Alternatively, you can also condition your commands on missing or non-missing values[2]:

```
tab education if age==.
```

This example would give you the frequency tables of the variable *education* for all people whose age we do not know.

In some cases it can be interesting to distinguish different cases of missing values (e.g. refusal, not applicable, etc), where you can extend the *dot* by a *letter*. Imagine that in the original data *refusals* are coded as -999 and *not applicable* as -998. We can then use for instance the `recode` command to set different types of missing values:

```
recode myvariable (-999=.a)(-998=.b)
```

You can even `label` these types of missing values like any other value `label`. When running econometric analyses, these missing values are considered like the standard missing values, hence they are not taken into account.

[2]Normally non-missing values are excluded by default for obvious reasons

3 Basic commands

Before starting with commands allowing to perform econometric analysis, it is important to understand the generic Stata® -command and to know several relevant commands to customize the Stata® environment and to load and save the data.

3.1 do-files

Commands should be written in a do-file, even though you could also write them directly in the command line and then pressing ENTER. The problem by doing so would be that you could hardly reproduce what you did before. Using a do-file is like writing a programming script, Stata® goes through it and performs each command you write in the do-file. An important point is that Stata® stops the execution on error, meaning that you can be sure that everything went well if the do file is executed until the end. To start a new do-file simply click on the button for the do-file editor (see figure 1) and start writing.

To execute the file you can click on the corresponding symbol () you find in the editor, go through the menu or click `[CTRL]+[D]` on your keyboard, which is obviously the most convenient way to do it.

Hint 2. The combination of pressing first `[CTRL]+[S]` and then `[CTRL]+[D]` might help you to save a lot of time and nerves. The first simply saves your do-file and the second executes. By doing it always like that, you can be sure your do-file is saved on the hard drive and even if Stata® crashes once, you will have your file

3.1.1 Line breaks

If you have very long commands that do not fit on the screen and you do not want them to be in one line, you can generate a line break with three slashes ///:

```
mycommand depvar indepvar1 indepvar2 indepvar3 indepvar4 indepvar5 indepvar6 ///
 indepvar7 indepvar8 indepvar9 indepvar10
```

alternatively, you can temporarily activate the active line break, meaning that you have to break the lines manually by the semicolon symbol (;) like in many other programming languages. To activate the manual line break type

```
# delimit ;
```

and to come back to the normal line break

```
# delimit cr;
```

3.1.2 Comments

You can and should comment your code which can be done with a double slash // or a star *. The double slash works at the beginning of the line or in the middle, while the star only works to declare a whole line as comment. For longer comments you can use the combination /* my comment */

```
* This is a comment on the whole line

reg y x //here I can comment the command

/* Here you can write a
comment on several
lines */
est store myreg //and yet another in-line comment
```

Hint 3. This is probably the most common hint: do comment your code as much and as clear as possible! This is not only useful when working with colleagues, but also when coming back to your do-file after a while. It might be difficult to understand your own code when it is not commented!!

3.2 Prepare Stata for your analysis

In a do-file the first things to do, even before loading the data, is to prepare the memory and the Stata® environment for it. Typically, we want to clean the memory before starting the analysis in order to have a common starting point every time the do-file is used. Using `clear all` erases all Stata® -relevant information from your memory. Once you did that you can change the memory size at the disposal of Stata® by using the command `set mem 100m` to give Stata® 100 megabytes of memory. Generally, it is advised to provide Stata® with sufficient memory, at least 100 megabytes more than the size of the database.
A next command you might want to use is the `cd` to change the working directory. The working directory is the place Stata® looks for databases if you do not indicate the whole path. Hence, it might be useful to change the working directory to the place where your data is stored (or where you want to store the output). This helps you to avoid writing every time the whole path. For instance, if you want to change the working directory to the folder *data* in your *C* drive you write:

```
cd C:/data
```

An alternative to change the directory is to define variables containing the path of your data and to use then these variables. Both methods have the enormous advantage over writing the whole path every time you load or save something, that when running the file on a different computer, you simply adapt one line of your code and you do not have to change it all over in your file.
Another command I suggest to use in the beginning of the do-file is

```
set more off
```

which avoids breaks in the execution of the do-file when the output of one single command is longer than the screen size. Normally Stata® stops allowing you to see all the results, but this is not very convenient when working with do-files. Rather than indicating this command in all do-files, you can execute once the command

```
set more off, permanently
```

in order to change it permanently (this changes the default value from on to off).
Hence, the head of a do-file could look like:

```
1:  clear all
```

```
2:  set more off
3:  set mem 250m
4:  cd C://data
5:  global source="C:/data/analysis1"
6:  global output="C:/data/analysis1/results"
```

where line one deletes all from your memory, line 2 disables the break in the output, line 3 increases or decreases the memory to 250 megabytes, line 4 changes the working directory to `C://data` and the lines 5 and 6 define two variables (source and output) containing the information of the source and output folder you will use.

3.3 Loading and saving data

Loading data in Stata® means to load the database into the RAM-memory of your computer. Only one database can be active at a time. The command is extremely simple

```
use C:/data/analysis1/mydata
```

to load for instance the database `mydata.dta` located in folder `C://data/analysis1`. Note that with the global variable defined in the short example before, we could simply write

```
use $source/mydata
```

since in the variable `source` we stored the whole path. Yet another possibility is to use only the relative path from the current working directory. As we are in the working directory `C://data` (indicated in line 4 of the example before) we can simply write

```
use analysis1/mydata
```

Hint 4. Always load and save databases with a do-file to avoid overwriting a database or to work on the wrong one.

Hint 5. Always save the database in a do-file with another name than the database you open at the beginning. Otherwise, you could not run the do-file twice, since the changes would now be in the initial database.

If needed, you can only load a part of a database, say variables x1 x2 and x3. The command becomes simply

```
use x1 x2 x3 using $source/mydata
```

Moreover, you can use conditions on the data, for instance an `if` (see section 3.6)

3.4 The generic Stata command

All Stata® commands have a common structure:

```
name varlist conditions, options
```

The command starts with the name of the command in order to tell Stata® what to do. The name is followed by the list of variables you would like to use in the command. Depending on the command, this list might include dependent and independent variables individually or together. The list of variables is followed by the sample conditions allowing you to perform the command only on a subsample of your

data. All you find after the comma are options, some of them might be mandatory. Understanding the logic of a Stata® command is crucial and it allows to fully understand the Stata® -help I present in section 2.3

3.5 Types of commands

Stata® commands are classified into different classes, depending on their content. Knowing about the exact definition of the classes is not absolutely needed, but it might be important to know the two most common for posterior use of stored results. The two main classes are:

r-class This is the most general class, including most of the commands for descriptive statistics such as `summarize` or `tabstat`. Results of this type of commands are stored in `r()`. To display the whole set of results, type `return list` after the command. Note that these results remain active until the next r-class command.

e-class The e-class commands are normally econometric estimations such as `regress`. The results are stored in `e()` and can be displayed typing `ereturn list`. As for the r-class, the values in `e()` are stored until the next e-class command is executed.

Example

For instance, if you would like to use the mean of a variable as a local you could use the following code:

```
summarize income
local meanincome=r(mean)
di ''The mean income is: `income' $''
```

where the first line is the r-class command `summarize` displays a series of summary statistics such as the mean, the standard deviation and others. The second line recovers the stored data[3] in `r()` and saves it in the local *meanincome*. Finally, in the third line of this not very practical example, a short text indicating the mean income is displayed. The following example recovers the adjusted R^2 statistic from a simple OLS regression and stores it in a scalar:

```
regress income education experience
scalar rsquared=e(r2_a)
```

Besides the e-class and r-class there exist also n-class and s-class commands, however, they are used very rarely. An easy way to find out if a command is r-class or e-class is to look in the help file how the results are stored. Normally towards the end of the help file a list of stored results are indicated.

3.6 Conditions

Generally, Stata® commands can be conditioned on a subsample of the dataset. The condition might take different forms and is normally introduced after the `varlist` and before the comma separating the command from its options. The most important condition is the `if`-condition starting simply with the word `if` followed by a logical condition. For instance, if you want to run a regression only for women and you have a dummy variable `female` taking the value of 1 if the person is a woman, the simple regression command becomes

[3] To see all the stored values, type `return list` just after the `summarize` command

```
regress y x1 x2 x3 if female==1
```

A second way to limit the sample is the `in`-condition, where you can run the command for instance for the first 100 observations (`regress y x1 x2 x3 in 1/100`).

Hint 6. The `in`-condition might be very helpful when you write a do-file containing a very computation intensive command and you would like to run the do-file in order to check if it works. Limiting the command to a small number of observations avoids loosing time when checking the do-file.

Hint 7. A nice and sometimes more elegant alternative to the `if` condition is to multiply your expression with a logical statement. Imagine you want to compute the value of a variable only for $|Z| < 1$, you can use the following command

```
gen kernel=(1-abs(Z)) * (abs(Z)<1)
```

where `(abs(Z)<1)` is a logical element returning 1 if the condition is satisfied and 0 otherwise, hence the variable `kernel` will take the value 0 (not missing!) whenever the condition is not satisfied.

3.7 Relational and logical operators

Like in every programming language, logic operators are crucial for a correct programming.

+	Addition	>	greater than
-	Subtraction	>=	greater or equal than
*	Multiplication	<	smaller than
/	Division	<=	smaller or equal than
^	Power	==	equal than
&	and	!=	not equal
|	or		

Stata® is very liberal in the use of logical operators and allows much more than other languages[4]. For instance "=" and "==" are not supposed to mean the same thing: `gen v1=1` generates a variable containing the value 1 and `ttest v1==1` performs a t-Test if the variable v1 has mean 1. However, you could also write `ttest v1=1` and Stata® automatically considers "=" as "==". This is sometimes useful, however I strongly suggest the correct distinction between the two equal signs, since some commands do not support both notations!

3.8 Placeholders and selection of multiple variables

Placeholders can become very important when you work a lot with Stata® . Imagine the quite academic case in which you have a database with one dependent variable `y` and 25 independent variables named `x1` to `x25`. Without placeholder you would have to write each `x`-variable individually in a command, however, using the * placeholder you can shortcut your command by a lot:

```
reg y x*
```

would be a simple OLS regression (see `regress`) of the variable `y` on all `x`-variables. Now assume that for some reason you only want to use `x12` to `x18`. In this case the * placeholder is not very useful, but

[4] One could also say that Stata® is somewhat sloppy with the logical operators

you can use the - sign:

```
reg y x12-x18
```

will include all the variables starting from `x12` up to `x18`. However, this gives you only the correct result if the variable order is alphabetical or if at least only the desired variables are in between `x12` up to `x18`. Hence, the - sign simply tells Stata® to take all variables between the two indicated variables.

Hint 8. The order of variables in Stata® has generally no specific logic and new variables are simply added in the end of the table. Using the command `order` you can order the variables according to your needs and `aorder` can be used to order all variables alphabetically.

3.9 Matrices

Stata® has two matrix systems, one that is implemented directly in the Stata® environment and Mata, which is to some extend a language apart. In this section, I deal only with the standard matrix package of Stata® . It has to be noticed that for matrix algebra Stata® is probably not the best software, so do not expect the quality of MATLAB or R when dealing with matrices.

3.9.1 `matrix define` - Generic command for matrices

Even though it is rather unusual to input a matrix by hand, in some cases it might be necessary (e.g. for very simple user written command).

```
matrix input A=(1,2,3\4,5,6)
```

defines

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Combining matrices

You can also create a matrix as a function of other matrices (as long as the dimensions match):

```
matrix define C = A + B
```

where `A` and `B` are two matrices of the same dimensions. The following table provides an overview of matrix operations available:

Operator	Symbol
transpose	'
negation	-
Kronecker product	#
division by scalar	/
multiplication	*
subtraction	-
addition	+
column join	,
row join	\

3.9.2 Some basic matrix functions

Some basic matrix functions include the following:

Function	Description
colsof(M)	Returns the number of columns in matrix M
rowsof(M)	Returns the number of rows in matrix M
issymmetric(M)	Returns 1 if M is symmetric, otherwise 0
det(M)	Returns the determinant of matrix M
trace(M)	Returns the trace of matrix M
I(n)	Returns the entity matrix of dimension n
inv(M)	Returns the inverse matrix of M

3.9.3 `matrix list` - Displaying matrices

To display matrix `mymat` you simply use

```
matrix list mymat
```

3.9.4 `matrix rownames/colnames` - Renaming matrix columns and rows

To rename the column and row names you can use the following commands:

```
matrix colnames A="Column 1" "Column 2" "Column3"
```

for the columns, and

```
matrix rownames A=A B C
```

for rows. As you can see, whenever the name consists only of one word, you do not need the quotes.

3.9.5 `matrix get` - Getting system matrices

Many estimation commands store important information in matrices, for instance the estimated coefficients of a regression are generally stored in the matrix `e(b)` and the corresponding variance-covariance matrix in `e(V)`. For further use of this information it is needed or at least recommended to extract these matrices to a user-defined matrix with the command

```
matrix b = get(_b)
```

for the betas for instance. Learn more about extracting results from estimations in section 7.6

3.9.6 `mkmat` - Data to Matrix conversion

It might be necessary in some cases to convert your data into a matrix or vice versa. This can be done very easily with

```
mkmat x1 x2 x3 x4
```

to convert the variables *x1*,*x2*,*x3* and *x4* into four column vectors with the same names or with

```
mkmat x1 x2 x3 x4, matrix(mymat)
```

to convert the four variables into a $x \times 4$ matrix called *mymat*.

3.9.7 `svmat` - Matrix to Data conversion

To perform the reverse operation, you use

```
svmat mymat
```

which will create four new variables named mymat1, mymat2, mymat3 and mymat4 respectively.

3.10 Factor variables and time series operators

Many times, you do not want to enter variables in regressions just like they are, but in some specific form. This is when the so-called *factor variables* enter the game. Basically, we are talking about prefixes for the variables to tell Stata® what to do with it.

3.10.1 `i.` - Categorical variables in the regression

Imagine you have a categorical variable and would like to include each category as a dummy variables (for category fixed effects). Defining for each outcome a dummy variable (using `tabulate` might be one solution, but probably not the most efficient one. You can simply write `i.` before the variable and Stata® will understand that all but one of these values should be used as dummy variables. This is not simply saving time in coding dummy variables but also reducing memory needs by a lot.

3.10.2 `l.` - Lagged variables

In time series or panel analysis, you might want to use lagged variables. This is easily done with the prefix `l.`, for instance

```
reg y l.y x1 x2
```

will perform an OLS estimation (`regress`) of y on the lagged value of y and two control variables x_1 and x_2. If you need more than one lag you can different choices:

Syntax	Variables	Description
`ll.x`	x_{t-2}	Double lagged variable used
`l(1/2).x`	x_{t-1},x_{t-2}	Lagged and double lagged variables
`l(0/5).x`	x_t,x_{t-1}, ... x_{t-5}	From non lagged to 5 periods lagged variables

3.10.3 `f.` - Forwarded variables

Just like lagged variables, it might be of need to use forwarded variables using the `f.` prefix. It works exactly the same way as `l.`.

3.10.4 `d.` - Difference

Just like lagged variables, you can create difference variables, for instance $\delta y_t = y_t - y_{t-1}$ is created by

```
gen dy=d.y
```

. Higher interval differences can be created with the same logic seen for `l.`.

3.10.5 `c.` - Continuous variables

Sometimes you have to specifically tell Stata® that a variable should be considered to be continuous, e.g. when you use interaction terms. However, normally this does not have any impact:

```
reg y c.x1
```

is exactly the same as

```
reg y x1
```

3.10.6 # - Interaction terms

Instead of defining interaction terms in a new variable using the product of the two in case of having continuous variables or combinations for categorical data, you can use the symbol #. However simply writing

```
reg y x z x#z
```

works only if x and z are categorical variables. In this case, all possible combinations are included as a dummy variable. To be clear about what Stata® does, I would however always write

```
reg y x z i.x#i.z
```

which is not absolutely needed but recommendable. This is especially the case because the #-symbol can also be used to create squared values and continues interaction terms. Assume now that both, x and z are continues and you would like to estimate the model

$$y = \alpha + \beta_1 x + \beta_2 z + \beta_3 xz + \epsilon$$

This can be done by writing

```
reg y x z c.x#c.z
```

where the `c.` tells Stata® that you would like to create an interaction by multiplication of two variables and not with all possible combination dummies.

3.11 Loops and programming conditions

A very important notion in any programming language or syntax are the loops and conditions. It is essential to understand them and to use them wherever it is possible. To understand this section on conditions and loops, it is necessary to be familiar with the data type local explained in section 2.5

Hint 9. Using loops instead of writing several times the same code with only slight differences helps reducing errors. Generally the shorter your code is, the better you did your job!

3.11.1 if/else - if/else/elseif clause

The most basic clause is the `if` clause. In Stata® it is implemented in a rather standard way:

```
local random=uniform()
if(‘random’>0.99){
display "This is rather an unlikely event"
}
else if(‘random’<0.01){
display "This is also an unlikely event"
}
else{
display "This is pretty normal"
}
```

The example first defines a local variable drawn form a uniform distribution. Then it displays a text in function of the value. The first *if* condition is TRUE when the value is higher than 0.99, the second element is a *else if* (with a space) condition being TRUE when the value is below 0.01 and the last element is the *else*-statement if the two conditions before returned FALSE.

3.11.2 `while` - while loop

The while condition is implemented in a very similar way to the *if* statement:

```
local i=1
while(‘i’<10){
di "‘i’"
local i=‘i’+1
}
```

This example simply displays all numbers from 1 to 10 on the output screen.

3.11.3 `foreach` - Looping through elements

Things are getting more interesting with the *foreach* statement, which loops through a predefined set of elements. For instance you can loop through a local variable:

```
local text="Hello Stata users"
foreach word of local text{
di "‘word’"
}
```

or

```
local myarray="1 2 5 9 7 15"
foreach i of local myarray{
di "‘i’"
}
```

The example takes every element of the local, stores it temporarily in the new local called *word* and uses it in the commands. You can also loop through variables:

```
foreach var of varlist x*{
sum ‘var’
}
```

or

```
foreach var of varlist x1 x2 x3{
sum ‘var’
}
```

where the loop performs the `summarize` command for each variable starting with the letter x. The general syntax of foreach is

`foreach` *runner* `of` *arraytype array*

where the italic elements can be changed. The *runner* refers to the local variable that changes in every loop, the *arraytype* indicates what kind of array the loop should go through (for instance local, global, `varlist`) and finally the *array* is the array containing the elements to loop through.

3.11.4 `forvalues` - Looping through numerical values

The last loop type is the `forvalues` which is contrary to the `foreach` limited to numerical values. Here are two examples:

```
forvalues i=1/4{
display "‘i’"
}
forvalues i=4(0.5)5{
display "‘i’"
}
```

where the first example displays the series 1,2,3,4 and the second 4,4.5,5

4 System commands

4.1 System commands

There are some Stata® commands, which you will not use very frequently, however, they might be of interest in some situations. The system commands are related neither directly to the data nor to the econometric analysis, but they allow you to adapt Stata® to your needs or update it.

4.1.1 `exit` - Leaving the do-file or Stata

The command `exit` has mainly two purposes. When running a do-file you might want to stop it at a given point (breaking point). By including the command `exit` the do-file will stop executing without an error message. If you use `exit` in the command line, Stata® will be closed if you do not have unsaved data in your memory. If you do have unsaved data, then you can add the option `clear` in order to force Stata® to shut down without saving the data.

4.1.2 `query` - Displaying Stata options

Stata® has many default settings you can change temporarily or permanently. To display all the settings the user can change just type `query` or `query topic` where `topic` is the topic for which you would like to the see the settings. For instance, `query memory` displays all the settings related to the memory. To change the settings, use the command `set`.

4.1.3 `set` - Change settings

Use `set` to change the default settings of Stata® , with or without the option `permanently` to keep the new settings for all future uses of Stata® . For instance, to change the memory allocated to Stata® to 300mb you can simply write

```
set mem 300m
```

Hint 10. The settings are saved locally on the machine, thus changing them permanently on a server version of Stata® might be useless.

4.1.4 `about` - Getting information of the version and license

By typing `about` Stata® displays information on your license and the version of your Stata® . Additionally information on the available memory of the computer is given.

4.1.5 `update` - Update your Stata

Like every software package Stata® needs to be updated from time to time. By typing `update all` Stata® will update all possible files where an update is available. Updates are made on executables, utilities and especially ado-files. If you simply wish to see if there are updates available type `update query`. Normally Stata® performs checks for updates every week, you can get the information by typing `query update`.

Hint 11. Updating a server based Stata® must be done on the server and not on the local machine

4.1.6 `findit` - Find ado-files online

`findit` is a very powerful command permitting you to search for ado-files (commands) on the internet. If you know the name of an ado-file then type it, otherwise use keywords to describe the command you are looking for. The search is performed in well-known Stata® -repositories and by clicking on `install` Stata® will automatically download and install the package. For instance, knowing that the package *nldecompose* performs Oaxaca decomposition for non-linear models, you can find and download using

```
findit nldecompose
```

Otherwise, if you want to find this package but you do not know its name, just write

```
findit oaxaca
```

and a list of potentially interesting packages will be displayed.

4.1.7 `search` - Search ado-files online

see `findit`

4.1.8 `help` - Display help

The Stata® -help is very useful for new and experienced users. I guess nobody knows all the commands and especially the options of Stata® -commands by heart, therefore the help is your best friend. Just type

```
help commandname
```

to display the help file of the command `commandname`. See section 2.3 for more information on the Stata-Help.

5 Data handling

5.1 Variable manipulation

In Stata® it is very easy to create, label and rename variables allowing you to understand your data afterwards much better. In this section, I present some useful commands.

5.1.1 `generate` - Generate a new variable

To create a new variable, you use generally the command `generate` (gen). For instance you can generate a variable by summing up two other variables

```
gen v3 = v1 + v2
```

To generate variables that contain summary statistics of the sample, for instance the sample mean, you have to use oftentimes `egenerate (egen)`

Generating dummy variables

To generate dummy variables you can use the `generate` command together with some conditions, for instance:

```
gen teenager=0
replace teenager=1 if age>10 & age <=20
```

generates a dummy variables taking the value of 1 if a person's age is bigger than 10 and smaller or equal 20. A simpler way to define this variable is just by including the condition in the first statement

```
gen teenager=(age>10 & age<=20)
```

5.1.2 `replace` - Replacing a variable

When a variable already exists you cannot use the command `generate`, but you might use `replace`, which works essentially the same way. For instance, if you want to replace negative income of people by a zero (or missing value) you could do

```
replace income=0 if income<0
```

or

```
replace income=.  if income<0
```

5.1.3 `egenerate` - Generate a new variable with summary statistics

The command `egenerate` or simply `egen` is similar to `generate` but is used normally to generate a new variable out of statistical operations. For instance, if you want to compute the average of several variables, let's say x1 x2 x3, you could use

```
gen xmean=(x1+x2+x3)/3
```

however this gives you a missing value if in any of the three variables you have a missing value, which might be exactly what you are looking for. If this is not the case and you would like to get a value even if one variable has missing values, then you should use

```
egen xmean=rowmean(x1 x2 x3)
```

Experience working with Stata® will help you to understand in which situation to use `egen` rather than `gen`.

5.1.4 `recode` - Recoding a variable

Recode is a powerful command to change the values of a categorical variable in particular. Imagine we get a database where the variable indicating the gender is coded as follows: 1=man and 2=woman and 3=unknown. We would like to have a variable where man=0 and woman=1 and .=unknown (see 2.7 for a discussion of missing values). We could use three times the `replace` command:

```
replace gender=0 if gender==1
replace gender=1 if gender==2
replace gender=. if gender==3
```

This is probably even a good solution in this case, but imagine you want to change the values of a categorical variable with 10 values. This is when `recode` becomes more efficient. To make the before mentioned change, the command would be

```
recode gender (3=.)(2=1)(1=0)
```

Actually, the command `recode` can much more than that:

```
recode x1 x2 x3 (-999 -998 -997 = .)(1 2 3 = 1)(5 6 7 = 2)
```

recodes the three variable x_1, x_2 and x_3 simultaneously and put the negative values -999, -998 and -997 to missing, 1,2 and 3 to 1 and 4, 5 and 6 to 2.
It is also possible to generate new variables (in this example `x_new` and `y_new`) out of the recoding and holding the original variables at their initial values:

```
recode x y (1=0)(2=1), gen(x_new y_new)
```

Note that the order of the `varlist` in the option `gen` must correspond to the order in the main `varlist` of the command.

5.1.5 `label` - Label your variables

Stata® know two types of labels: value labels and variable labels.

Variable labels The variable label is simply a text describing a variable in general. It appears in the variable overview and helps you to understand the content of a variable. To define or redefine (overwrite the old) a variable label simply write

```
label variable income "Income of the person in US$"
```

to label the variable `income` with the text between the two quotes.

Hint 12. Try to keep variable names short and describe the content in the variable label. For instance if you have a variable with the log annual income per capita avoid variable names like `log_annual_income_per_capita` and use rather `lincpc` with a corresponding variable label *Log annual income per capita*

Value labels

Besides the variable label Stata® is also capable to assign value label, meaning that each value of a variable is labeled and in the database the label is shown instead of the actual value. This is useful in the case of categorical data. Let us illustrate the value labels with a small example. Imagine a variable `lstatus` taking three values: 1 for people still in education, 2 for people in the active labor force and 3 for retired people. Using for instance the command `tab` we could display the frequencies, but the table would not be very self-explaining:

```
    lstatus |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |        357       35.70       35.70
          2 |        324       32.40       68.10
          3 |        319       31.90      100.00
------------+-----------------------------------
```

Therefore it is useful to label the values. The first step is to define the label. This would be

```
label define lstatlabel 1 "In education" 2 "Labor force" 3 "Retired"
```

where we name the label *lstatlabel* and then indicate the value and in quotes the corresponding label. This command only saves the label, we still have to assign it to the variable.
To assign the variable we use again the command `label`:

```
label value lstatus lstatlabel
```

Now the table with the frequencies becomes:

```
     lstatus |      Freq.     Percent        Cum.
-------------+-----------------------------------
In education |        357       35.70       35.70
 Labor force |        324       32.40       68.10
     Retired |        319       31.90      100.00
-------------+-----------------------------------
```

which looks already much nicer.

Hint 13. It is possible to assign the same label to different variables, for instance you can define a label called *dummy* and assign it to all the dichotomous variables. Let us say we want to label the variables d1, d2 and d3 with the label *dummy* we would use `label value d1 d2 d3 dummy`

Once the variables are labeled, it is also possible to automatically extract these labels. For more details see section 11.2.6.

5.1.6 `rename` - Rename a variable

Variables in micro data have oftentimes quite reasonable but not very intuitive names, thus it is important to rename variables in order to have intuitive names. Renaming in Stata® is extremely easy, let us say we want to rename the variable *st98q01* in *indigenous*. The command is simply given by:

```
rename st98q01 indigenous
```

Note that this does not alter the content of the variable nor any kind of labels.

5.1.7 `drop` - Deleting variables

To delete one or various variables use

```
drop id gender age
```

The example above simply drops the three variables `id`, `gender` and `age`.

5.2 Describing and sorting the data

5.2.1 `describe` - Describe the dataset

`describe` is used to display a description of the dataset, not descriptive statistics of the data! By typing `desc` without a variable all the variables will be described, while the inclusion of a variable list will limit the description to those variables. Here is the example for only one variable:

```
Contains data
  obs:         1,000
 vars:             1
 size:         8,000 (99.9% of memory free)
------------------------------------------------------------------
              storage  display     value
variable name   type   format      label      variable label
------------------------------------------------------------------
lstatus         float  %12.0g      lstatlabel Labor market status
------------------------------------------------------------------
```

where Stata® indicates you first how many observations are available in the dataset (independent of missing values!) and how many variables you have. The value *size* refers to the memory use of the database (in this case almost nothing is used). After this information on the whole dataset, the information of each variable is displayed. First the name, then the format and the display format, followed by the assigned value label and the variable label.

5.2.2 `codebook` - Display the codebook

`codebook` is similar to the variable related part of describe, however, with much more details. Besides the format and some information on the labels, the units, the range, frequencies etc. are displayed.

5.2.3 `sort` - Sorting the data

You can easily sort your data according to one or more variables with the command `sort`. For instance

```
sort IDhousehold IDindividual
```

would make the following change in the database:

IDhousehold	IDindividual	Age
2	1	56
1	2	34
1	1	33
2	2	61

becomes

IDhousehold	IDindividual	Age
1	1	33
1	2	34
2	1	56
2	2	61

5.2.4 `gsort` - Sorting the data

`gsort` works basically like `sort`, but you can sort the data in the opposite direction by including a minus sign (-):

```
gsort -year
```

will sort the data according the variable `year` starting with the highest value.

5.2.5 `order` - Sorting the variables

While `sort` sorts your data vertically, the command `order` allows you to sort the data horizontally, meaning you can change the order of the variables. Normally this is not a very important feature, but there are situations when it might be necessary (e.g. to have the identifier of the observations at the beginning for easier use or to use the placeholder symbol "-" in estimation commands). By writing

```
order id name country
```

the variables *id*, *name* and *country* will be put in the beginning of the dataset, while all the remaining variables remain unchanged!

5.2.6 `aorder` - Sorting the variables alphabetically

Like the command `order`, `aorder` allows you to change the order of the variables in your dataset, however, in the alphabetical order. By simply writing

```
aorder
```

without any `varlist` , all the variables will be ordered alphabetically, where special symbols like underlines (_) come first, followed by capital letters and lower case letters.
If you indicate a `varlist` like

```
aorder ID name country
```

then the variables *id*, *name* and *country* will come first, followed by all the remaining variables in alphabetical order.

5.3 Joining several databases

Especially when working with micro data, it is oftentimes needed to merge several databases into one. Stata® is very efficient in this kind of data handling. Two main ways of merging/appending two or more datasets into one are to be considered. The first situation is when we have two datasets with different variables for the same individual, firm or household. In this case we use the command `merge`. The second situation is when we have two databases with the same variables but not for the same people, thus we would like to add one to the other and we use `append`

5.3.1 `merge` - Merging/appending two databases (horizontally)

Before starting to describe the command `merge` there must be made a difference between the version 11 and higher and the previous versions, since the command changed slightly. Both versions perform the same action in the end, but the syntax became more explicit in the newer version. In what follows I refer to both versions.

Prepare the data for `merge`

Before being able to merge we have to prepare the data. A necessary condition is that in both databases we have an identifier of the observation, which might consist of a single variable (e.g. individual ID number), or a series of variables (e.g. family ID number + intra family rank). In both databases the variables must be coded the same way and should have the same format. We have to `sort` the databases by these identifiers first. Since only one database can be active at a time, there is always the *master*-database (the active one) and the *using*-database, the one we integrate in the *master*.

Merging under version 10 or lower

In version 10 or lower of Stata® the command to use when ready is:

```
merge id1 id2 using myusingdata
```

where id1 and id2 are two variables that identify the observation and that are present in both databases. The word *using* is used to indicate Stata® that the following part will be the database(s) to be joined. The last part refers to our *using*-data called myusingdata in this example.

Merging under version 11 and higher

Basically the syntax in the newer version is the same, with the small exception that we have to indicate now where we can find multiple entries. Imagine that you have individuals in the *master* data and household information in the *using* data, thus the same household data must be added to all members of the same household. In the earlier versions Stata® just "tried to understand" it at the risk to misunderstand it. Now, you must indicate it here. The example would then be for instance:

```
merge [n:1] householdID using householdinfo
```

In this example the *master* data contains the individuals and the *using*-data the household information. The term `n:1` means that multiple observations in the *master*-data are to be merged with one observation in the *using*-data. If you have individual data on both sides, you would use `1:1` and in case of merging multiple to multiple the term becomes `n:n`.

Merging result

Once the merge of the databases performed, Stata® displays the results and stores information in the new variable `_merge` by default[5]. The generated variable is coded in the following way:

1	the observation was only found in the master data (no merge)
2	the observation was only found in the using data (no merge)
3	merge successful, observation found in both
4	observation found in both, missing values updates
5	observation found in both, conflict in some variables

5.3.2 `append` - Appending/merging two databases (vertically)

Appending data is generally easier than merging, since you don't need identifiers. The command `append` simply adds the rows `using`-database (see `merge`) to the current database. In case of having the same variable in both databases (the normal case), the data is just added. If in one of the two databases the variable was not present, the column is added and in the database where it was not present, missing values are added. The command is simply

```
append using myusingdata
```

where `myusingdata` is the database to add to the active database.

In case of not having the same format in the variables (same variable in the two databases), you should use the option `force` in order to enable Stata® to convert the formats where necessary. This might be especially needed for string variables, since they are generally defined to be as long as the longest

[5] with the option `generate(myvarname)` you can change `_merge` to `myvarname`

entry (to save memory) and these values might differ from one the other database.

5.4 Changing structure of a database (panel data)

5.4.1 `reshape` - Reshape your panel data

There are two ways to save panel data in a database, the *long* and the *wide* form. Depending on the amount of variables that are constant over time (e.g. gender) the one or the other form are more suitable.

long:

id	year	income
1	2000	5.6
1	2005	7.2
2	2000	8.3
2	2005	9.1

wide:

id	income2000	income2005
1	5.6	7.2
2	8.3	9.1

The command `reshape` can be used to change your data easily from one to the other format. If you want to reshape from wide to long, then use

```
reshape long income, i(ID)
```

where a new variable `_j` will be created with the years. If you want to call it *years* right away, you can include additionally the option `j(year)`.
To get back to the wide form, write

```
reshape wide income,i(id) j(year)
```

6 Summary statistics and graphics

6.1 Descriptive statistics

6.1.1 `summarize` - Summary statistics

The command `summarize` or just `sum` provides you with the essential summary statistics of the variable. Without options or `varlist` the command provides the number of non-missing observations, the average, the standard deviation and the minimum and maximum value. By using the option `detail` additional summary statistics like the percentiles, skewness and kurtosis are displayed. `summarize` is a r-class function and stores all the displayed valued in *r(name)* which you can consult typing `return list`

6.1.2 `tabstat` - More flexible summary statistics

An alternative and oftentimes more flexible way to get summary statistics is the command `tabstat`. Its basic syntax is

```
tabstat varlist [if] [in],[ stats(statisitics) by(byvar)]
```

where the `varlist` contains all variables you would like to analyze. The `if` and `in` conditions are straightforward. The true utility of the `tabstat` command comes with the two non-mandatory options `stats(statisitics)` and `by(byvar)`. The first allows you to specify the list of statistics you would like to compute. The most common options include the mean (`mean`), median (`median` or `p50`), standard deviation (`sd`), maximum (`max`), minimum (`min`) or the number of observations (`n` or `count`). The whole list of available statistics can be found in the helpfile. The `by(byvar)` allows you to compute these statistics by subpopulations defined by the variable `byvar`.
Let us consider two examples. First,

```
tabstat income, stats(mean median min max) by(region)
```

will give you the mean, median, minimum and maximum of the variable `income` for each region as defined in the variable `region`. For each region, Stata® will display a row with the four specified statistics. Adding the option `column(variable)` would change the direction of the table, putting the variable in the columns and the statistics in the rows.

```
tabstat income wage consumption, stats(mean median min max) column(stats)
```

will do the same but instead of displaying a row for each region, you will get a row for each variable specified. You can also combine the two! Without the option `column(stats)` Stata® will display the variables in the columns and the statistics in the row.

6.1.3 `tabulate` - One- and twoway tables of frequencies

While `summarize` is very useful for continuous variables, `tabulate` is the most useful command for categorical data. It provides you with a table of frequencies of each possible outcome. You can also create a 2-way table where every possible combination of two variables are presented. For instance

```
tab gender indigenous, mi
```

gives something like

```
           |           indigenous
    gender |         0          1          . |     Total
-----------+---------------------------------+----------
         0 |     2,608        669         54 |     3,331
         1 |     2,627        635         74 |     3,336
         . |        48         12          2 |        62
```

```
-----------+---------------------------------+----------
     Total |     5,283      1,316        130 |     6,729
```

where the option `mi` (missing) indicates that you also want the missing values to be considered as a category. The total number of observations is 6729, 2608 non-indigenous men and for instance 12 indigenous people who's gender we don't know. About 2 people we do not have any information.

Generate dummies of each category

The *tabulate* command has an interesting side-feature that allows you to create easily a dummy variable for each possible outcome, by simply adding the option `gen(name)`. For example

```
tab gender, gen(d)
```

creates a variable *d1* for men and *d2* for women. If we would have specified the option `missing` an additional variable *d3* for the missing values would have been created. The naming of the new variables is simply the name indicated and an increasing integer at the end.

6.1.4 `correlate` - Correlation and Covariance

The simple correlation of two (continuous) variables is probably one of the most useful descriptive statistics. To display the correlation matrix of a set of variables simply use

```
corr x1 x2 x3
```

and Stata® will provide you with a result like

```
             |       x1       x2       x3
-------------+---------------------------
          x1 |   1.0000
          x2 |  -0.0185   1.0000
          x3 |   0.7128   0.6881   1.0000
```

In case of preferring the covariance matrix instead of the correlation matrix, add the option `covariance` to the command:

```
corr x1 x2 x3, cov
```

which will display you the following matrix:

```
             |       x1       x2       x3
-------------+---------------------------
          x1 |  1.05657
          x2 | -.018846  .987245
          x3 |  1.03772    .9684  2.00612
```

6.2 Graphs and plots

6.2.1 `plot` - Easy scatter plot

A very easy, even though somewhat outdated, way to visualize your data is the `plot` command, which provides you in the output window an approximate graph of your data. For instance,

```
plot x1 x3
```

would display something like:

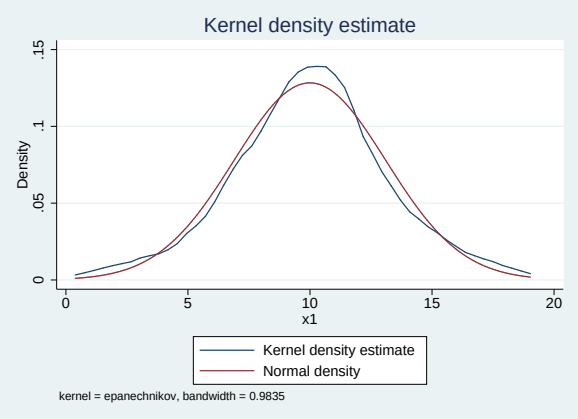
```
 5.10264 +
         |                                                      *
         |                                                          *
         |                                              *     *
         |                                              *
         |                                    ** * *
         |                                 * ** *
         |                             * **   *
         |                          * ** * *
    x    |                    *  ****** *
```

```
   1    |                   * * *
        |                ******* *
        |           * *   ***
        |            ** *      *
        |       * * **
        |   * ** ***
        |    **
        |      * *
        | **   *
        |
.057795 +    *
         +------------------------------------------------------------+
         4.16933                        x3                        31.902
```

I agree that this way of visualizing data is probably not the state-of-the art in the century of vectorial graphics, however, it is a fast way to get a first impression. A newer - and much more advanced - way to visualize data is the `graphics` command, which I will explain in the next section.

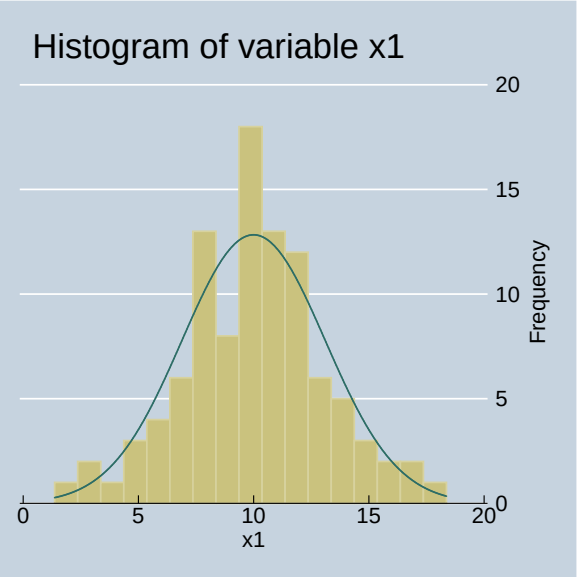
6.2.2 `graphics` - General graphics

Stata has an enormous amount of possible graphics you can create, however, the precise syntax is not always easy to obtain. The Stata® graph website provides a very useful visual overview of the different graphs that can be produced with corresponding commands. The following table presents some example, hoping that the reader can extract from the proposed syntax some useful information and tricks.

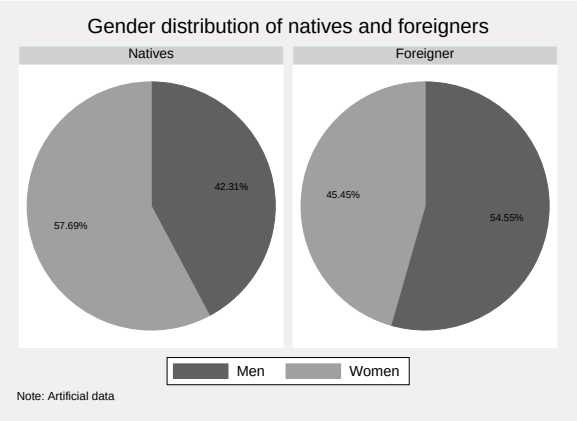
Output	Syntax
	`kdensity x1, normal`

Output

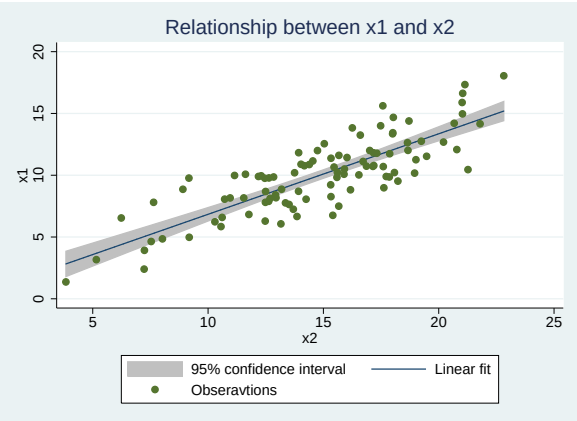
Syntax




```
histogram x1, normal width(1)
frequency title(Histogram of variable
x1) scheme(economist)
```

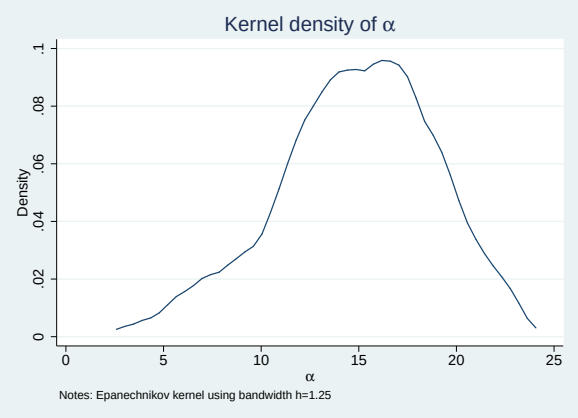



```
graph pie , over(female)
by(foreigner,title(Gender
distribution of natives and
foreigners, span) note(Note:
Artificial data, span)) plabel(_all
percent) scheme(sj)
```




```
twoway (lfitci x1 x2)(scatter x1 x2),
title(Relationship between x1 and x2)
ytitle(x1) xtitle(x2) legend(order(1
"95% confidence interval" 2 "Linear
fit" 3 "Obseravtions"))
```

Output



Syntax

```
kdensity alpha, title(Kernel density
of {&alpha}) xtitle({&alpha})
ytitle(Density) bw(1.25) note(Notes:
Epanechnikov kernel using bandwidth
h=1.25)
```

Hint 14. Using the combination of curly brackets "{}" and the &-symbol, you can use greek letters in the text you add to graphics. In the last example, α is written in greek letters. Here are some examples:

Symbol	Stata-Code
γ	{&gamma}
ϕ	{&phi}
Φ	{&Phi}

6.2.3 `spmap` - Build vectorial maps with shapefiles (shp)

`spmap` is a user written package to generate very nice looking vectorial maps based on the data in your Stata® database. You need a shapefile[6] (shp) containing the GIS data of the country of interest with the same identifier for entities (e.g. states) as you have in your data[7]. First at all, you have to convert the shp-file to a Stata® database using the command `shp2dta`:

```
shp2dta using CHE_adm, data(CHE-d) coor(CHE-c) genid(id) replace
```

This is an example to convert the file *CHE_adm.shp* into two databases with names you can freely choose. Here I chose CHE-d for the database containing the data (all type of information from the shp file) and CHE-c for the database with the coordinates. Moreover, the option `genid(id)` generates a new variable `id` with the identifier of the unit.

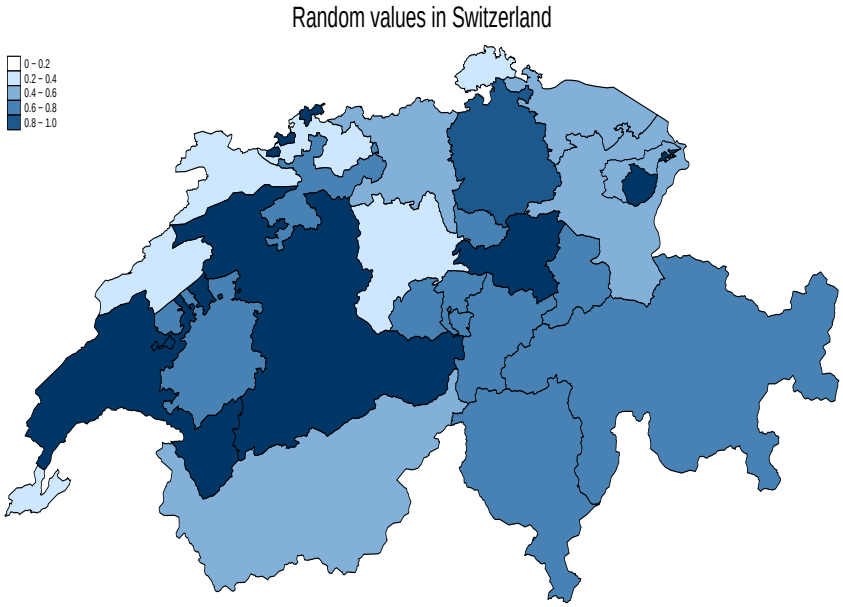
In a second step we have to `merge` the CHE-d (data database) with our database of the analysis, e.g. your working database on any economic phenomenon. It is important to check if the identifier used in the shp-file corresponds to the one you use in your file, otherwise you have to change the it in your file, as it would become very hard to change in the shp-file. Once you have merged your data, it is very simple to create a map, using for instance the following code

```
1:    spmap myvalue using "CHE-c.dta", id(id) clmethod(custom) ///
2:    clbreaks(0 0.2 0.4 0.6 0.8 1) fcolor(Blues)  ///
3:    ocolor(black ..) plotregion(icolor(none)) legenda(on) legstyle(0) ///
4:    legend(order(1 "0 - 0.2 " 2 "0.2 - 0.4" 3 "0.4 - 0.6"  4 "0.6 - 0.8" 5 "0.8 - 1.0") position(11) )
```

[6] Free shapefiles can be found under http://www.mapcruzin.com and on the websites of some government agencies depending on the country

[7] If the numbers are different, then change the identifier in your data, this is easier than in the shp-files

produces the graphic hereafter. The first line indicates on which variable we want to perform the graphic (myvalue) and which database contains the coordinates. The `id(id)` indicates the identifier of the unit (here Swiss cantons) and `clmethod(custom)` is used to customize the thresholds between categories. These are indicated in `clbreaks(....)` and the option `fcolor` selects the color set to be used. An overview of the color sets can be found in the help file. `ocolor` is used to define the color of the border and `plotregion(icolor(none))` defines the background (here empty). The following commands are used to customize the legend: first its set to be displayed, then the style is selected and finally the values are changed to whatever text you want.



This fully vectorial map was exported from Stata® with `graph export` as explained in sections 6.2.4 and 8.1.1.

6.2.4 Export graphs

When you produce a graphic in Stata it is generally displayed in a new window. You can easily save graphics in various formats using the command `graph export`, followed by the name of the file (with extension!). Use the option `as(`*`format`*`)` to indicate the format to export and `replace` to overwrite an old graphic if you wish to do it. If you don't use the `as()` option, the file extension will be used to determine the format. The supported graphic formats under windows are:

.ps	PostScript
.eps	EPS (Encapsulated PostScript)
.wmf	Windows Metafile
.emf	Windows Enhanced Metafile
.png	PNG (Portable Network Graphics)
.tif	TIFF

I suggest the use of PNG for the standard use, since the graphics are relatively small and all standard programs can read them.

Hint 15. If you work with LaTeX and would like to use Stata® -graphics, you should export the graphics as vector graphics in order to get the best possible quality. See section 8 on how to export graphs to LaTeX and how to use them in LaTeX without loss of quality.

7 Econometric analysis

The goal of this section is not to describe all possible estimation commands in detail, but rather to give a short overview of commands, helping to find the needed routine. If you wish to learn more about the command and its options, you should refer to the help file, which includes in many cases examples. Type

```
help mycommand
```

to display the help file of the command *mycommand*.

7.1 Continuous outcome

7.1.1 `regress` - OLS estimation

The most basic regression is econometrics is the OLS estimation and the `regress` command might be the most used. To perform an OLS estimation of y_1 on x_1, x_2 and x_3 you simply write

```
reg y x1 x2 x3
```

or using placeholders you can even reduce it to

```
reg y x*
```

if there are no other variables starting with the letter "x".
All regression commands include a variety of possible options, starting from more technical settings for Stata® and going to relatively complicated versions of the estimated model. A commonly used option is used to change the way standard errors are computed. In a general way, the option is called `vce(vcetype)`, where *vcetype* can take many different values[8]. Common options include:

option	Description
`robust`	Computes heteroskedasticity-consistent standard errors according to White (1980). This option is available in many estimation commands and can be invoked directly by typing for instance `reg y x1 x2, robust` instead of `reg y x1 x2,vce(robust)`.
`noconstant`	Performs the OLS estimation without constant term.
`beta`	Provides standardized coefficients β^* defined as $$\beta^* = \beta \frac{\sigma_x}{\sigma_y}$$ where β is the estimated coefficient,σ_x and σ_y the standard deviation of the independent and dependent variable respectively.

A typical output of a simple regression analysis looks like:

```
      Source |       SS       df       MS              Number of obs =    1000
-------------+------------------------------           F(  2,   997) =  996.87
       Model |  2034.95712     2  1017.47856            Prob > F      =  0.0000
    Residual |  1017.61138   997   1.0206734           R-squared     =  0.6666
-------------+------------------------------           Adj R-squared =  0.6660
       Total |  3052.56851   999  3.05562413           Root MSE      =  1.0103
```

[8] see `help vce_option` for more details

```
------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
          x1 |   1.058603   .0324676    32.60   0.000     .9948909    1.122316
          x2 |   .9816983   .0326528    30.06   0.000     .9176222    1.045774
       _cons |   .9089836    .031962    28.44   0.000     .8462631     .971704
------------------------------------------------------------------------------
```

where the upper left panel provides ANOVA-like information of the sum of squares, degrees of freedom and mean sum of squares. The upper right panel provides some statistics of the model fit and the main panel thereafter is the actual estimation. Each row refers to one regressor, starting with the coefficient (coef.), followed by the standard errors (Std. Err.), the t-statistic (t), the p-value ($P > |t|$) and the 95% confidence interval.

7.1.2 Other estimators of continuous outcome

The following table provides a short overview of other estimators used for continuous variables.

Command	Description

`cnsreg`	Constrained linear regression model. Define first a constraint using the command *constraint*:
	`constraint 1 x1=x2`
	and then use the command for the constrained linear regression model:
	`cnsreg y x1 x2 x3,constraints(1)`

`gmm`	Generalized method of moments estimation. See the help file for details.
`heckman`	This command allows you to perform the OLS estimation with selection following Heckman (1976). The command is used similar to `regress`, but you include the option `select(`*model*`)` where *model* refers to the selection model, having a dummy variable as dependent variable, taking the value of 1 when the observation is selected in the main equation and 0 otherwise.
`ivreg`	Instrumental variable (IV) estimator. The syntax is relatively easy, assume that you regress y on x_1, x_2 and x_3, where x_1 and x_2 are assumed to be endogenous and therefore will be instrumented by z_1,z_2 and z_3. Then the command is simply `ivreg y (x1 x2=z1 z2 z3) x3` Using the options `2sls`, `liml` and `gmm` you can choose the estimator to be the two-stage least squares, the limited-information maximum likelihood or the generalized method of moments estimator respectively.
`reg3`	Three-stage least squares (3SLS) estimator for simultaneous equations. See the help file for details.
`sureg`	Zellner's seemingly unrelated regression. See the help file for details.
`tobit`	Tobit estimation for censored dependent variable. The syntax is similar to the `regress` where you add two options `ul(#)` and/or `ll(#)` to indicate the upper and the lower limit respectively. For example if you have data on income that is censored due to data collection at 999999 then the command would be `tobit income educ exper, ul(999999)`
`treatreg`	Treatment-effects model
`truncreg`	Truncated regression model. Similar to the `tobit` model but for truncated variables. For instance the income is truncated at zero, since we cannot have negative incomes: `truncreg income educ exper, ll(0)`

- Continued on next page -

`selmlog`	This is a user written command to perform OLS estimation with Selection bias correction based on the multinomial logit model. The command features correction terms according to Lee (1983), Dubin and McFadden (1984) and Dahl (2003).

7.2 Categorical outcome

7.2.1 `probit` - Probit estimation

`probit` estimates the standard probit model for dichotomous dependent variables. The syntax is exactly the same as for `regress`. If you want to obtain rather the marginal effects than the raw coefficients, the command `dprobit` is more suitable. See also section 7.9 for more details on marginal effects using the newly introduced command `margins`.

7.2.2 `dprobit` - Probit estimation with marginal effects

`dprobit` is based on `probit` but reports the marginal effects. If you want to publish marginal effects in your work and use `estout` to produce your estimation table, then you have to use `dprobit` instead of `probit`, since otherwise `estout` cannot report the marginal effects.

7.2.3 `logit` - Logit regression

The `logit` command is used exactly the same way as the `probit` command. See the help file for details.

7.2.4 `mlogit` - Multinominal logit

To perform a multinominal logit model, you can use the command `mlogit` where the syntax is very easy as well:

```
mlogit y x1 x2 x3, base(2)
```

where y is a categorical variable with integer values. The option `base(2)` in the example is not necessary but permits you to define the base outcome, here the value 2. See the help file for details.

7.2.5 `oprobit` - Ordered probit model

`oprobit` performs an ordered probit model (Greene, 2008, Ch. 23.10.1, page 831). The dependent variable must be an ordered categorical variable, where higher values refer to better outcomes.

7.2.6 `ologit` - Ordered logit model

`ologit` performs an ordered logit model, see `oprobit` for more details.

7.3 Count data

7.3.1 `poisson` - Poisson regression

`poisson` performs a Poisson regression on count data. The dependent variable must contain positive integer values only. See the help file for details.

7.3.2 nbreg - Negative binomial regression

`nbreg` performs a negative binomial regression on count data. The dependent variable must contain positive integer values only. See the help file for details.

7.4 Panel data

7.4.1 xtset - Set-up the panel data

Before being able to perform panel data analysis, you have to tell Stata® about the panel structure of the data. Normally the data should have the long form (see `reshape` for details on the form and how to change it). Typically you have a variable identifying the individual (the unit more generally) and a variable describing the time dimensions (e.g. the year). Assume now that these two variables are *id* and *year*, then the command to define the panel data is given by

```
xtset id year
```

which provides you directly with some information on the data enabling you to verify if everything went well:

```
  panel variable:  id (unbalanced)
   time variable:  year, 2000 to 2004, but with gaps
           delta:  1 unit
```

To get a more detailed description of the data's pattern, consider the command `xtdescribe`

7.4.2 xtdescribe - Describe the pattern of the panel data

`xtdescribe` allows you to get some more information on the structure of your panel data. You must have used `xtset` before. The just type

```
xtdescribe
```

and you will get a detailed description like the following:

```
      id:  0, 1, ..., 99                                     n =        100
    year:  2000, 2001, ..., 2004                             T =          5
           Delta(year) = 1 unit
           Span(year)  = 5 periods
           (id*year uniquely identifies each observation)

Distribution of T_i:   min      5%     25%       50%       75%     95%     max
                         2       3       4         4         5       5       5

     Freq.  Percent    Cum. |  Pattern
 ---------------------------+---------
       49     49.00   49.00 |  11111
        8      8.00   57.00 |  11.11
        7      7.00   64.00 |  111.1
        6      6.00   70.00 |  .1111
        6      6.00   76.00 |  1.111
        6      6.00   82.00 |  1111.
        4      4.00   86.00 |  1.11.
        3      3.00   89.00 |  ..111
        2      2.00   91.00 |  .1.1.
        9      9.00  100.00 | (other patterns)
 ---------------------------+---------
      100    100.00         |  XXXXX
```

First general information on the panel structure is indicated, for instance the variables identifying both dimensions with their respective value pattern. You will also be informed if the combination of dimension identifiers identifies each observation uniquely, a very important issue for further analysis. Finally the most common patterns in the data are displayed, where the `1` indicates that the value is present and the dot (.) refers to missing data. In our example we have 49 observations where information is available for all 5 years in the sample, followed by the second most common pattern where in all but the third year information is available.

7.4.3 `xtreg` - Panel regression: fixed and random effects

Once your data is set up, you can start running regression models. In this section I only present the command `xtreg` which is able to perform fixed-, between- and random-effects and population-average linear models.
The syntax is similar to the syntax used in `regress` by simply putting

```
xtreg depvar indepvars, model
```

where *depvar* and *indepvar* are the dependent and independent variables respectively and *model* refers to the type of model that you want to estimate:

`re`	Random effects model (default), GLS estimation
`fe`	Fixed effects model
`be`	Between effects model
`mle`	Maximum likelihood random effects model
`pa`	Population average model

See the help file for details.

Hint 16. See the section 3.10 on factor variables and timeseries operators to learn more about the use of lagged, forwarded and differenciated variables in panel models

7.5 Time series

In general you can refer to what was written about panel data in section 7.4 and apply it directly to time series data. Instead of the prefix `xt` you should now use `ts`. For instance to set up your data, you use

```
tsset year
```

For the moment, no more time series specific information is available in this document. For further details refer to section 7.4 or check out http://dss.princeton.edu/online_help/stats_packages/stata/time_series_data.htm

7.6 Extracting estimation results

Almost every Stata® routine saves results in macros names `r(`*name*`)`, `e(`*name*`)`, `s(`*name*`)` or `c(`*name*`)`. You can easily access this data either directly of saving them to your own macros. Look at the following example:

```
sum x1
```

```
local mean=r(mean)
local sd=r(sd)
local t=‘mean’/‘sd’
display "The t-statistic is: ‘t’"
```

First, the summary statistics of the variable $x1$ are computed with the command `summarize`. This command is a so-called r-class command, meaning that the data is stored in `r(`*name*`)`. To see all saved values in r() type `return list`[9]. The second and third line simply copies the values into used defined local variables. The fourth line then computes a new value out of the stored data and the last line displays the result.
Some results are stored in matrices (e.g. estimation coefficients and variance-covariance matrices. See section 3.9.5).

7.7 Post estimation

Once a regression is performed, it is oftentimes needed to perform some post-estimation tests. In this section I present a few of them, however, in general you find in the help files of the command a link to the post-estimation commands of this specific command. This is useful, since not all estimations allow for the same post-estimation commands.

7.7.1 `hettest` - Breusch-Pagan / Cook-Weisberg test for heteroskedasticity

After `regress` you can simply type `hettest` to perform a Breusch-Pagan test on heteroskedastic error terms. In the output the H0 is explained, being a constant variance. Hence, finding a p-value lower than the threshold would indicate heteroskedastic error terms and you might want to consider the White (1980) correction using the option `robust` (see section 7.1 on page 39)

7.7.2 `test` - Test linear hypotheses after estimation

`test` allows you to make tests on the coefficients of a regression. For instance, to test whether the coefficient of the variable *x1* is equal to 2, type

```
test x1==2
```

or to test whether the coefficients of x_1 and x_2 are equal put

```
test x1==x2
```

This command will be performed on the active estimation, which is in general the last estimation performed.

7.8 Saving and reusing estimations

7.8.1 `est store` - Save an estimation

Like for the database, you can always have at most one estimation active, but estimations can be stored and used later on as well. To store an estimation simply write

```
est store myestimation1
```

[9] Type `ereturn list` for e-class commands, `sreturn list` for s-class commands and `creturn list` for c-class commands

to save the current estimation under the name *myestimation1*. This is very useful to produce estimation tables afterwards (see `est table` (section 7.8.4) and `estout` (section 8.2.1)).

7.8.2 `est restore` - Restore an estimation

You can restore (re-activate) an estimation previously saved at any moment by simply writing

```
est restore myestimation1
```

This will not display the regression again, but the data off the regression will be available again in the e-class-scalars (type `ereturn list` to see them).s

7.8.3 `est replay` - Replay an estimation

If you want to display an earlier regression again, then you can use

```
est replay myestimation
```

Important: this command does not reactivate the estimation, it simply displays it. To reactive an earlier regression, use the command `est restore`.

7.8.4 `est table` - Display an estimation table of several regressions

A very nice feature of the general command `estimates` is the sub-command estimates table, which allows you to display several regressions at a time in order to compare them. I illustrate the use of this command with a short example using the system-database *auto.dta*. First, we perform and save to regressions:

```
sysuse auto, clear
reg price weight length
est store reg1
reg price weight length trunk
est store reg2
```

This example simply performs two OLS regressions with the command `regress` on two and three independent variables respectively and saves the results in *reg1* and *reg2*. Now we would like to display the two regressions with

```
est table reg1 reg2
```

which yields to the following display:

```
-------------------------------------
    Variable |    reg1         reg2
-------------+-----------------------
      weight |  4.6990649    4.7215989
      length | -97.960312   -102.66518
       trunk |               28.376437
       _cons |  10386.541    10812.329
-------------------------------------
```

This is not yet the nicest estimation table ever seen! With a few additional options, you can make it much better:

```
est table reg1 reg2, stats(N r2_a) b(%6.3f) star(0.1 0.05 0.01)
```

you already get a useful display like:

```
-------------------------------------
    Variable |    reg1         reg2
-------------+-----------------------
      weight |   4.699***     4.722***
      length | -97.960**    -102.665**
       trunk |               28.376
       _cons | 1.0e+04**    1.1e+04**
```

```
-------------+--------------------------
           N |       74           74
      r2_a |    0.329        0.320
-----------------------------------------
     legend: * p<.1; ** p<.05; *** p<.01
```

where you immediately see the significance levels of the coefficients and some statistics of the estimation. Let us have a look at the different options:

`star(n1 n2 n3)`	Allows you to define significance levels of the stars. In our example the standard 10%, 5% and 1% were chosen.
`b(`*format*`)`	Allows you to format the output of the coefficients and to limit the amount of digits after the coma. See 2.6.4
`stats(`*args*`)`	Enables you to display some estimation relevant statistics: `N` Number of observations `aic` Akaike's information criterion `bic` Schwarz's Bayesian info. criterion `chi2` Value of χ^2 `ll` log likelihood
`se`, `t`, `p`	You can display in addition to the coefficients the standard errors (se), the t-statistics (t) and the p-value (p). Simply indicate the desired statistic together with its format, e.g. `se(%6.3g)`.
`keep(`*coeflist*`)`	Only the coefficients indicated in *coeflist* will be shown
`drop(`*coeflist*`)`	All coefficients in *coeflist* will be dropped from the table

7.9 Marginal effects

In many model the estimated coefficients of a model cannot be interpreted directly and one needs to compute the marginal effects first. In the case of a probit, the usual way to do this was to use `dprobit` instead of `probit`. However, Stata® introduced recently a completely new command `margins` that is intended to compute marginal effects for a very wide range of models. Since I believe that this will be the command to be used in the future, I will discuss it with some more details hereafter.

7.9.1 `margins` - Compute marginal effects (post estimation)

`margins` is a relatively new command to compute marginal effects for a large range of models. It is a somewhat complicated command with a rather difficult syntax at first sight. On the other hand, it offers many very nice options to estimate not only marginal effects at the mean, but basically wherever you want. In this explanation I will focus on the very basic capabilities of `margins` and I encourage the reader to have a look at the official help file of the command for more advanced uses of the command. I will explain the command using a simple probit model, since most of the reader are familiar with it and we will be able to compare it easily to the older `dprobit` command. For the following discussion I use the freely available dataset *margex* which you can load by typing

```
webuse margex
```

Let us consider the simple probit model where `outcome` is explained by two continuous variables `age` and `distance` and the gender dummy called `sex`. The simple estimation of

```
probit y sex age distance
```

will provide us with the estimated coefficients that are not directly interpretable. An easy way to get the marginal effects instead would be to use

```
dprobit y sex age distance
```

Unfortunately, this solution does not store the marginal effects in the system matrices. Moreover, the newly introduced command `margins` is more powerful, so let us discuss how it works.
Since it is introduced as a post-estimation command, it has to be always in combination with a prior estimation command. To get the same marginal effects as in the example with `dprobit` one would have to type:

```
probit outcome i.sex age distance
margins , dydx(*) atmeans
```

where it is important to use the factor variable (`i.sex` instead of `sex`) in the probit. Otherwise, the command `margins` does not recognize `sex` as being a dichotomous variable and uses the procedure for continuous variables. The option `dydx(*)` is used to indicate Stata® that we want to estimate the marginal effect for all regressors. By replacing the `*` by some regressors, `margins` computes the marginal effects only for the indicated variables. The option `atmeans` permits to compute the marginal effect at the mean instead of the average marginal effect (see next paragraph).

ME at the mean vs. mean ME

Marginal effects are not uniquely defined in non-linear models. One option is to compute the marginal effect for an average person (this is what the example above does) or the marginal effect for each observation at their observed values and then average the different effects. Formally and with some slight abuse of notation, the average marginal effect (AME) and the marginal effect at the mean (MEAM) can be defined as follows for the probit model:

$$AME = E\left[\frac{\partial \Phi(X\hat{\beta})}{\partial x_j}\right] \tag{1}$$

$$MEAM = \frac{\partial \Phi(E[X]\hat{\beta})}{\partial x_j} \tag{2}$$

where $\Phi(.)$ is the cumulative normal distribution, X is the whole matrix of covariates and x_j indicates the variable for which we are computing the marginal effect.
It is also possible to compute the marginal effect at different points of the distribution. Let us consider some examples, always assuming that the model we estimated before the `margins` command was

```
probit outcome i.sex age
```

In this case

```
margins, dydx(*)
```

computes the marginal effects for each observation at its observed levels of the regressors. In a second step the average of all these marginal effects is computed giving you the *average marginal effect*. In contrast,

```
margins, dydx(*) atmeans
```

defines a person that has the average characteristics and computes for this person the marginal effect giving you the *marginal effect at the mean*.

Marginal effects at different points of the distribution

Sometimes it is interesting to look at the marginal effects at different points of the distribution. For instance, if we would like to compute the marginal effect for people at the age of 50, we would use

```
margins , dydx(*) at(age=50)
```

To limit it further to men at the age of 50, we would use

```
margins , dydx(*) at(age=50 sex=1)
```

It is also possible to compute the marginal effects for several values. For instance, to compute them simultaneously for the age of 20, 40 and 60, we would use

```
margins , dydx(*) at(age=(20 40 60))
```

Finally, we can automate these intervals using the following notation

```
margins, dydx(*) at(age=(20(5)60)
```

where the marginal effects are computed for all multiple of 5 starting from 20 and going up to 60.

Marginal effects for different groups/populations

It could also be interesting to compute the marginal effects for different populations. These groups can be defined by the covariates of the model or variables that were not used in the model. To activate this computation by groups you can use the `over(varlist)`, where the whole computation is done for each value of the variables in *varlist*[10]. It is important not to confound this option with the previously explained `at()` option. While the `at()` option refers to the position in the distribution of a covariate for which the marginal effect should be computed, the option `over(varlist)` computes the marginal effect at a given position (can be specified as well) in the distribution for different groups. For instance,

```
margins , dxdy(*) over(sex) atmeans
```

computes the marginal effect at the mean for men and women separately.

Computational issues

Sometimes the computation of marginal effects can be very time consuming. If the computation takes too much time, one might consider not to compute the standard errors (if they are not absolutely needed), which should make the computation much faster. This can be done by adding `nose` as an option.

Export marginal effects

Most of the time, we do not want to limit the computation to the display on the screen but would like to export them. `margins` might surprise you negatively when you first try to do this. For instance, if you use the following code

```
probit outcome i.sex age distance
margins , dydx(*) atmeans
est store reg1
estout reg1, margin
```

you will get the estimated coefficients of the probit and not the marginal effects. This is because margins does not overwrite the e-values of the probit estimation. If you want `margins` to do so, you

[10] Alternatively, you can also use the option `by(varlist)`, giving you exactly the result

just have to add the option `post`. Hence, the correct syntax for the above example would be

```
probit outcome i.sex age distance
margins , dydx(*) atmeans post
est store reg1
estout reg1, margin
```

Actually, the option `margin` is no longer needed in the command `estout`, however, I leave it here, because you might have other regressions in the `estout` that still need the option.

Predicted values at different levels of covariates

An additional nice feature of `margins` is the ability to compute predicted values (or expected values) conditioned on some specific values of covariates. Imagine for instance that we would like to have the marginal effect of an interaction of two dummy variables. For non-linear models the standard procedure cannot be used. However, we can simply compute the predicted outcome conditional on all possible combinations of the two dummy variables. Then we can compute the marginal effect and using the `test` command we can test its significance. Here a very simple example:

```
sysuse nlsw88, clear
probit union i.south##i.married age
margins, predict() at(south=(0 1) married=(0 1)) atmeans post
test 4._at-3._at-2._at-1._at == 0
```

The first line loads the database, on the second line I estimate a probit model with an interaction term between two dummy variables (see 3.10 for details on factor variables). The third line computes the four possible conditional probabilities, where the option `atmeans` set all other control variables to their respective mean and the option `post` saves the coefficient for posterior use. Finally, the fourth line performs a test whether the marginal effect of the interaction term is significant or not[11].

Final remarks on `margins`

I presented here only the most basic features of `margins`, however, there are many more. The command is very complete and usable for a wide range of estimations. Even though the syntax is somewhat heavy at first sight, I encourage the reader to invest some time in this command, because you can really get very nice results out of it.

[11] See Ai and Norton (2003) to see why the test on the coefficient is not useful in non-linear models

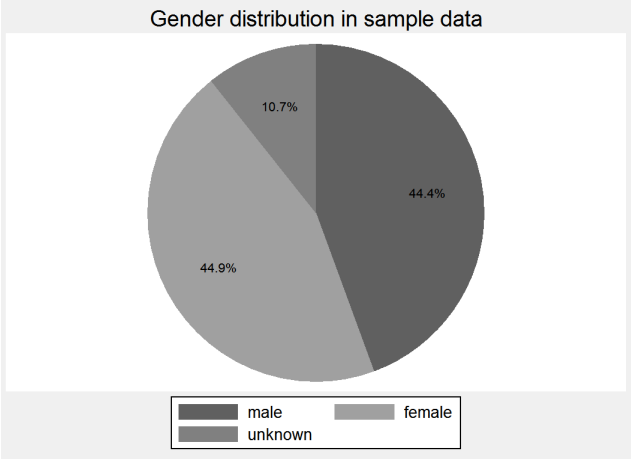
8 Stata meets LaTeX

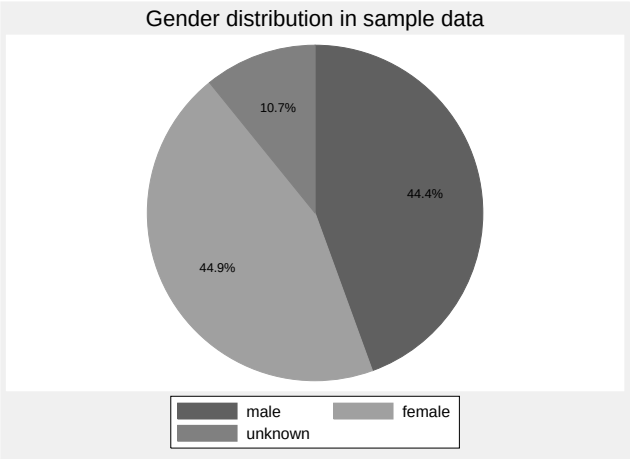
Besides the high quality features in econometric analysis and data handling, Stata® is also a very powerful tool thanks to its ability to communicate with other software package. More precisely, it is relatively easy to export econometric results and graphics to formats that can be read by other packages, especially LaTeX . In this section I present some of the possible ways to export your results to LaTeX and also how to import them into your LaTeX project. The ultimate goal of this section is to enable you to automate the process completely. Changing your model slightly in Stata® will automatically change the tables and graphs in your LaTeX document, thus running the Stata® code and compiling the LaTeX document will suffice![12]

8.1 Exporting high quality graphics to LaTeX

8.1.1 `graph export` - Graph export to eps

Once you have your graph in Stata® you can export it into the .eps-format. This format is generally not very easily readable under Windows, but the import into LaTeX is easy. The normal command `includegraphics` can be used in combination with the package epstopdf. The package converts the eps graphic into a pdf graphic which LaTeX can use. Hence the complete LaTeX code to use is:
In the preamble: `\usepackage{epstopdf}`
and in the body: `\includegraphics{mygraphic.eps}`
The following two graphs were built with the same code, once exported as PNG (left) and once as EPS (right). I suggest you to zoom in the two graphs to a maximum and then I let you choose your preferred graph:





8.2 Exporting estimation results to LaTeX

8.2.1 `estout` - Creating estimation tables

Different genuine Stata® - and user written-commands are available to export your estimation results to LaTeX . Here I focus on the command `estout` written by Ben Jann. In my opinion it is the most flexible command and offers great possibilities. The basic syntax is similar to `est table`, but it offers much more features. I illustrate its use with a small example based on the two regressions used in the demonstration of `est table` in section 7.8.4:

[12] The part of interpreting the results and writing the paper cannot be automated yet ;-)

```
1:    estout reg1 reg2 using regtable.tex, replace style(tex) margin cells(b(fmt(%9.3f) star) se(par) ) ///
2:    starlevels(* .1 ** 0.05 *** .01) ///
3:    stats(N  r2_a , labels(N  "Adj. $ R^2$ ") fmt(%9.0f %6.3g)) mlabels("Model 1" "Model 2") ///
4:    collabels(none) varlabels(_cons "Constant"weight "Weight" length "Length" trunk "Trunk space") ///
5:    order(length weight trunk _cons) ///
6:    prehead(\begin{center}\begin{tabular}{l*{@M}{l}} "\hline") ///
7:    posthead("\hline") ///
8:    prefoot("\hline") ///
9:    postfoot("\hline\end{tabular}\end{center}")
```

produces the following table:

	Model 1	Model 2
Length	-97.960**	-102.665**
	(39.175)	(42.587)
Weight	4.699***	4.722***
	(1.122)	(1.132)
Trunk space		28.376
		(97.058)
Constant	10386.541**	10812.329**
	(4308.159)	(4574.211)
N	74	74
Adj. R^2	.329	.32

The command might look a bit scary at first, but you will get used to it very quickly. Moreover, once you defined your style correctly, you can copy-paste large parts of the code from one table to the next. Let's go through the code with detail. The first line corresponds mainly to what we saw for the command `est table`. First we include with `reg1 reg2` the names of the regressions we want to display, followed by the file to create. After the comma we use the option `replace` in order to be able to re-run the script (otherwise there is an error) and the `style(tex)` to declare that we want a LaTeX output. The option `margin` does not make much sense in this example, since in the OLS estimation all estimated effects are marginal effects. However, if you estimate for instance a `dprobit` model, this option is needed to display marginal effects rather than the coefficient estimates. The option `cells` is used to choose all the coefficient-related information to be displayed. In this example the betas (`b`) is formatted with the *9.3f* `format` with stars included and for the standard errors (`se`) I use the option `par` to get the standard errors in parentheses.
The second line simply defines the significance levels of the stars and in contrast to the option saw in `est table`, you can also choose other symbols than the stars.
The third line first declares the statistics of the estimation to be displayed, followed by the label of the statistic and the format. The option `mlabels` allows you to give each regression a name; if you do not specify this option, *reg1* and *reg2* will be displayed.
Line 4 formats the first column where the variables are indicated. Without any of these option the variable name, as it appears in Stata® , will be displayed. The first option, which is not active here, would allow you to use the labels of the variable rather than the names and the option `varlabels` allows you to change each variable name separately for the table output without changing anything in your data. In this example I just changed all the names to capital letters and the constant term to a nicer looking word.
The option `order` on line 5 allows you to change the order of the variables in the table and the remaining 4 lines are all related to the LaTeX code. They allow you to put some free text (or LaTeX commands)

at very specific places in the table. The `prehead` is before the name of the regressions. I use it here to include directly the *tabular* environment of LaTeX using the variable @M, which is the number of models, directly computed by the command. Hence, you do not have to adapt the number of columns if you add a regression, this will be done automatically. If you want to include your table in a *table*-environment of LaTeX , you could start it here as well. All the information will simply be transferred to the generated LaTeXcode. The `posthead` is between the regression names and the first coefficients and the `prefoot` will be placed between the coefficients and the statistics, while the `postfoot` is after the statistics.
This is only one of many possible examples of the command `estout` and I encourage you to consult the manual of the command.
To include the command to your latex file, simply use

```
\input{regtable}
```

The big advantage is that you can make small changes directly in your do-file and the table will be adapted in your LaTeX -paper. This command should definitely help you to avoid copying estimation results by hand to you LaTeX file, a process with some risk of making errors.

8.2.2 `tabout` - Creating descriptive statistics tables

Similar to the command `estout` you can export frequency tables with the command `tabout`, written by Ian Watson. This command has extremely many options, which makes it a little bit difficult at the beginning. A very simple example of the code is:

```
tabout female indigenous using crosstable.tex  , replace ///
    style(tex) cl1(2-4)  c(cell) layout(rb) f(1p) h3(nil) font(bold)
```

which makes a cross frequency table of the variables *gender* and *indigenous*as follows:

	indigenous		
female	**No**	**Yes**	**Total**
No	44.4%	5.8%	50.2%
Yes	45.7%	4.1%	49.8%
Total	90.1%	9.9%	100.0%

The `using crosstable.tex` defines the file to be written, the `replace` options allows you to overwrite an earlier version of the file. `style(tex)` indicates that you want to have a LaTeX file, the `cl1(2-4)` add a horizontal line between column 2 and 4 right after the variable *indigenous*. `c(cell)` indicates that you want to have the percentage by cell, alternatives are `freq` for frequencies and `col` and `row` for percentages according to the column and row respectively. The `f(1p)` option indicates the format of the table, here the *p* stands for percentage. Simply `f(1)` would create a number with one decimal. The `h3(nil)` option avoids that at the top of each column a %-sign is shown (N in case of frequencies) and finally `font(bold)` makes some parts of the table bold. You could also include some pretable and posttable LaTeX -code, but this must be done with external files. Therefore I prefer not to do it in my latex-code, thus the import becomes then:

```
\begin{tabular}{llll}
 \input{crosstable}\endrule
\end{tabular}
```

9 Importing data in other formats

9.1 `usespss` - Read SPSS data

Oftentimes micro data is made public in the SPSS (or PASW) format (file extension .sav), however, it is very easy to import such data into Stata® . Using the command `usespss` you can open SPSS-data just like the normal Stata® format. Labeled data will be imported with labels. The command works like `use`:

```
usespss $source/myspssdata.sav
```

will import the data and then you can use it directly or save (convert) it into the .dta-format by using the command `save`.

9.2 `fdause` - Read SAS XPORT data

Exactly like the `usespss`-command, you can import data from SAS using the command `fdause`.

10 Stata is not enough? Add user-written modules to enhance Stata

Stata® comes with a quite complete set of commands and modules to perform most of modern econometrics analyses. However, some very specific analyses might still be missing. If you are not eager to program the modules yourself, you can search for a user-written command in the internet. For econometric analyses the best platform is SSC (Statistical Software Components) on IDEAS[13]. To find the correct module, use the `findit` command. For instance, if you would like to run a treatment effect analysis, type

```
findit treatment effect
```

Several propositions are made. An alternative way to find user written modules is to search on the internet and most of the time you will end up at the SSC on IDEAS.
If you already know the name of the command and it is available on SSC, you can install it very easily

```
ssc install NAMEOFMODULE
```

Other SSC related commands that might be useful are:

Command	Description
`ssc new`	Displays the newest user-written command available on SSC.
`ssc hot`	Displays the most downloaded modules.
`ssc uninstall mypackage`	Uninstalls the package named `mypackage`
`ssc install mypackage, replace`	Updates the package `mypackage`
`ssc describe mypackage`	Displays the description of the package

Hint 17. To check which version of a command you have installed on your machine, you can simply type

```
which packagename
```

and Stata® will show you the version of the module `packagename`

Very useful user-written commands include: `ivreg2`, `estout`, `tabout`. Many of the user-written commands do not necessarily help you estimating very complicated model, but they might be very useful for some basic tasks like converting data to other formats, exporting nice-looking tables, performing some basic statistical tests, etc...

[13] http://ideas.repec.org/s/boc/bocode.html

11 Programming your own command

When you start using Stata® a lot and perform many times the same type on analysis it might be interesting to program your own routine. This is fairly easy and not much more knowledge is needed than for a standard do-file. In this section I will just write how to start programming, but I will not enter a long discussion about specific commands for programmers.

```
1:  capture program drop iop
2:  program define iop , rclass
3:  version 8.2
4:  syntax varlist [if] [in] [, BOOTstrap(integer 0) DECOMPosition
    GRoups(varlist max=1)  PRopt(str) BOOTOPT(str) PROBIT*]
5:  marksample touse
6:  quietly{

7:  tokenize ‘varlist’
8:  local depvar ‘1’
9:  macro shift
10: local xvars ‘*’

11: // BOOTSTRAP
12: if(‘bootstrap’ & ‘bootstrap’>0){

13: forvalues i=1/‘bootstrap’{

[SOME LINES OF THE CODE ARE NOT REPORTED HERE]

14: end
```

The first line drops the program (if already existent) from the memory. This is needed to enable us to define it again. The word *capture* is included to avoid an execution stop in case of an error, typically when the program does not exist yet. The second line then defines the program with the name *iop*, which must be equal to the name of the ado-file (see later on). The option after the coma refers to the class of the routine, r-class is generally a good option. Line 3 declares Stata® that the ado-file should work from version 8.2 on, hence trying to run it on an earlier version causes a problem. The fourth line is the most important for the moment, since we declare the syntax - this is basically the same as in the help files. After the word *syntax* you write `varlist` if you will enable the user to provide a varlist. You can include the possibility to offer the user the `if` and `in` statements. If you do so, you have to consider this after in the code, since Stata® does not limit the routine to the limitations given by the user automatically! After the coma you might include all the options. In this case all options are non-mandatory, since the opening bracket is before the comma. The capital letters indicate the minimum amount of letters the user has to write, for instance the option `bootstrap` will be understood by Stata® whether the user writes `boot` or `bootst` or `bootstrap`, but not if he writes `boo`. Two types of options are available: with and without arguments. Those without arguments generate simply a local containing the complete name of the option when the option is chosen. The option with arguments save the arguments in a local variable with the name of the option. You have to

indicate always the nature of a argument, being for instance *str* for a string or *varlist* for a `varlist` . Line 5 then converts the `if` and `in` conditions into a temporary variable I call here touse. In any routine you use afterwards you will have to indicate

```
commandname varlist if ‘touse’
```

in order to limit your routine to the sample. An alternative is to keep only the sample you need:

```
marksample touse
preserve //saves the current state of the DB for later
keep if ‘touse’
[ALL YOUR PROGRAM]
restore //restores the database as it was before the preserve command
```

The remaining lines of the code are then more or less standard. You can use all the routines available in Stata® . The do-file end with the command `end` (line 14).

11.1 Where and how to save your routine?

The routine can be written like a normal do-file but must be saved under the extension .ado under the form myroutinename.ado. It is important that the file has the same name as your command, otherwise Stata® cannot execute it. Under Windows it is most recommended to have a folder `C://ADO` where Stata® saves all updates anyway and there to save your ado-files in `C://ADO/personal` in order to have them at a save place. Once you saved and executed your ado-file once, it is available for the use in Stata® as long as you don't delete the file!

11.2 Useful commands for programming

Some commands are especially useful when programming ado-files, allowing you to achieve the highest possible flexibility. Here are some examples:

11.2.1 `tokenize` - Tokenize a string

Many commands include a `varlist` where the first variable is the dependent variable and the remaining independent. Technically the `varlist` is just a string stored in a local variable, thus Stata® does not know a priori what to do with it. An easy way is to tokenize the string. Assume to have a `varlist` in you syntax and that the string will be available for you in the form of a local variable called *varlist*. By using

```
tokenize ‘varlist’
```

you can split up your string into its elements. To illustrate this point consider the following example:

```
local varlist="income educ exper gender"
tokenize ‘varlist’
```

which splits up the text *income educ exper gender* into 4 elements stored in local variables called 1, 2, 3 and 4. For instance, you can now change the order:

```
display="‘3’ ‘2’ ‘4’"
```

would produce *exper educ gender*. Additionally the local named simply "*" contains the whole initial string:

```
di "`*'"
```

produces *income educ exper gender*. The interesting feature of * is the use with `macro shift`.

11.2.2 `macro shift` - Shifting the content of a local

The command `macro shift` makes a very simple but effective operation on the local * (see `tokenize`) by eliminating the first element. Assume that the local * contains the string from section 11.2.1: *income educ exper gender*, then

```
macro shift
```

convert the string into *educ exper gender*.
Consider now the combination of `macro shift` with `tokenize`. Assume that the `varlist` contains as fist element the dependent variable and the remaining elements represent the explanatory variables, then the following code help you to divide them:

```
tokenize `varlist'   // Splits up the local varlist
local depvar=`1'     // Stores the first element in the local depvar
macro shift          // Drop the first element of the local *
local indep=`*'      // Stores the remaining elements as local indep
```

11.2.3 `marksample` - Selecting the sample considering if and in

As already mentioned in the beginning of this section, the command `marksample` allows you to retake the information of the `if` and `in` option of your syntax.

```
marksample touse
```

simply generates a temporary variable *touse* taking the value of 1 if the observation should be included and zero otherwise.
You can then use in every command of your ado-file the condition

```
my command if `touse'
```

or simply limit the sample from the beginning. If choosing the second option, do not forget to `preserve` the data at the beginning and `restore` it in the end, otherwise the user will have bad surprises after using your command!

11.2.4 `preserve` - Making an image of your current data

The command `preserve` is not particularly a command for ado-files, but I present it here, since in the programming of ado-files its most often used. `preserve` simply makes an image of the current state of your data, allowing you to manipulate it thereafter and being able to recover it again, as it was, with the command `restore`

11.2.5 `restore` - Restoring an image of your data

`restore` can only be used when `preserve` was used before and allows you to recover the image of the data created earlier.

11.2.6 `local:extended` - Extract labels from variables and value labels

When programming a module it is sometimes useful to get the main labels and the value labels of the variables. For instance, instead of displaying the variable names in the output, it might be nicer to display the variable label. Using the extended functions for local variables (it works also with

global variables), you can easily extract this information from the data and use it afterwards. For a discussion on the labels and how to create them, see section 5.1.5.

Extracting the variable label

Let us start by extracting the variable `label` from a variable `female`. Simply type

```
local mylocal:variable label female
```

to save the variable label of the variable `female` as local called `mylocal`[14]. To use the extracted text, you can simply include the local, for instance by typing:

```
display "The variable label is:`mylocal'"
```

Extracting the value label

Besides the label for the variable, the value labels might be of interest for the programmer. To extract the value labels we have to proceed in two steps. First we have to extract the name of the value label for a given variable and we can then extract the label for every possible value. Let us consider again the variable `female` and assume that it has a value label called `gender` where 0 is labeled `Male` and 1 `Female`. To get this information we simply extract the name of the value label by writing

```
local foo:value label female
```

We have now stored the name of the value label in the local `foo` and can proceed to the extraction of the individual labels, for instance for the value of 1:

```
local label2:label `foo' 1
```

We have now stored the value label for 1 in the local `label2` and could use if for instance by typing

```
display "The value label for female=1 is:`label2'"
```

Of course, if we already know the name of the value label, we could directly request the value by typing:

```
local label2:  label gender 1
```

However, this is rarely the case when we program a routine which should be applicable for different datasets.

[14] Of course, you can use whatever name for your local.

12 Simulation

The discussion of simulation in this document is limited to the discussion of generating random variables.

12.1 Uniform distribution

The easiest way to create a random variable is definitely the uniform variable, which can be combined with the `generate` command:

```
gen x=uniform()
```

which gives a variable $x \sim U_{[0,1]}$ To generate rather $U_{[10,20]}$ you only need some basic algebra:

```
gen x=10 + uniform()*(20-10)
```

12.2 Normal distribution

To generate normal variable a special command is available besides the one related to a normal `generate`. I suggest the use of `drawnorm`:

```
drawnorm x1 x2
```

will create two i.i.d variables with standard normal distribution.

```
drawnorm x1 x2, means(1 2) sds(4 3)
```

will generate two unrelated random variables:

$$x_1 \sim \mathcal{N}(1, 4)$$
$$x_2 \sim \mathcal{N}(2, 3)$$

One can go further by simulating random variables form a higher dimension normal distribution. To do this, you will need to indicate either the covariance or the correlation matrix. The following example uses the correlation matrix:

```
matrix define R=(1,0.9\0.9,1)
drawnorm x1 x2, means(0 0) sds(2 3) corr(R)
```

The first line defines the correlation matrix (see section 3.9.1) and the second line perform the random draw. Note that by default as many observations as the dataset has will be drawn. This can be changed with the option `n(#)`

12.3 Other distributions

Like the uniform distribution in section 12.1, random variables can be drawn from many different distributions. Here is a short overview:

`runiform()`	= uniform()
`rbeta(a,b)`	Beta distribution
`rbinomial(n,p)`	Binomial distribution
`rchi2(df)`	χ^2 distribution
`rgamma(a,b)`	Gamma distribution (Γ)
`rhypergeometric(N,K,n)`	Hypergeometric distribution
`rnbinomial(n, p)`	Negative binomial distribution
`rnormal()`	Standard normal distribution (`drawnorm` preferable)
`rnormal(m,s)`	Normal distribution (`drawnorm` preferable)
`rpoisson(m)`	Poisson distribution
`rt(df)`	Student distribution

To generate such a variable, you can simply use the `generate` command:

```
gen myvar=rpoisson(4)
```

generates a variable called `myvar` drawn from a Poisson distribution with parameter $\lambda = 4$.

12.4 Setting the random seed

By default no fixed random seed is set, thus every time you run you simulation another result will appear. To control the random seed, you can set it with the following statement:

```
set seed n
```

where n can take any integer value, which allows you also to repeat the same simulation with different seeds, using for example loops (section 3.11).

References

Ai, Chunrong and Edward C. Norton, "Interaction terms in logit and probit models," *Economics Letters*, 2003, *80* (1), pp. 123–129.

Dahl, G. B., "Mobility and the Returns to Education: Testing a Roy Model with Multiple Markets," *Econometrica*, 2003, *70*, 2367–2420.

Dubin, J.A. and D.L. McFadden, "An Econometric Analysis of Residential Electric Appliance Holdings and Consumption," *Econometrica*, 1984, *52*, 345–362.

Greene, William H., *Econometric Analysis*, 6 ed., Pertinence Hall, Upper Sadle River, New Jersey, 2008.

Heckman, J., "The common structure of statistical models of truncation, sample selection, and limited dependent variables and a simple estimator for such models.," *Annals of Economic and Social Measurement*, 1976, *5*, 475–492.

Lee, L.F., "Generalized Econometric Models with Selectivity," *Econometrica*, 1983, *51*, 507–512.

White, Halbert, "A Heteroskedasticity-Consistent Covariance Matrix Estimator and a Direct Test for Heteroskedasticity," *Econometrica*, 1980, *48* (4), 817–838.

Index