Introduction to Programming Stata

Ulrich Kohler

June 3, 2011

Contents

1 Programming Stata
  1.1 Some basic tools .................................................. 1
  1.2 Loops .......................................................................... 4
  1.3 Programs ..................................................................... 7
  1.4 Step by step: a practical example ................................ 9
  1.5 Programming style .................................................... 10

2 Programming Mata
  2.1 First principles ......................................................... 11
  2.2 Stata in Mata ............................................................ 14
  2.3 Mata in Stata ............................................................. 17
  2.4 Common problems proper solutions ............................. 19
  2.5 Programming Style ................................................... 20

1 Programming Stata

1.1 Some basic tools

The display command

. display "Hello"
. display 42
. display 6*7
. display "6*7 = " 6*7
. display as text "6*7 = " as result 6*7
. display "{txt} 6*7 = {res}" 6*7
. display cond("SPSS">"Stata","{txt} Yes","{err} No")
. display cond("SPSS"<"Stata","{txt} Yes","{err} No")
Local Macros

```plaintext
.display "hello"
.local i hello
.display "i*
.display 21+21
.local i 21
di `i='+i'
.local i = 21+21
di `i'
di `i`*
.local j 21+21
di `j'
di `j`*
di `i` = 2
.dia
```

Extended Macro functions

```plaintext
.use data1, clear
.local x: variable label ymove
display "x"*
.local x: label (state) 1
display "x"*
.local q The answer to life the universe and everything
display "x"*
.local a 42 said Deep Thought, with infinite majesty and calm
display "q" is `word 1 of `a``
display "q" is `word `= `x`/`x` of `a``
.local list1 Don't talk to me about life!
.local list2 life about
display "": list list1 - list2``
```

Also see help extended_fcn.

Scalars

```plaintext
.scalar answer = 42
.scalar question = "What is the answer?"
display as text question " " as result answer
```

Maximum lengths if string scalars is 244 characters, while locals can hold 1,081,511 characters. Also note:

```plaintext
.scalar area = "Denmark"
display area
```

- Do not use variable names for scalar names
- . display scalar(area)
• 
  . tempname area
  . scalar `area' = "Denmark"
  . display `area'

Saved results

. sum np9501
. return list
. gen econworry = (r(max)+1)-np9501
. reg rent sqfeet
. return list
. sc rent sqfeet ///
  > || lfit rent sqfeet ///
  > || , note("n= `e(N)'" "R-Quadrat = `e(r2)"")

Some practical examples

• A derived statistic

  . sum hhinc if gender=="women":sex, meanonly
  . local mean1 = r(mean)
  . sum hhinc if gender=="men":sex, meanonly
  . local mean2 = r(mean)
  . display "{txt}Mean-Difference {res}`mean1 - `mean2"

• Parse labels into output

  . sum sqfeet
  . sc rent sqfeet, xline(`r(mean)')
  . sc rent sqfeet, ///
    > title(":variable label rent' on ':variable label sqfeet"")

Advanced example I

```do
3: sum sqfeet if ost & !mi(rent)
4: local xost = r(max)
5: reg rent sqfeet if ost
8: local yost = _b[_cons] + _b[sqfeet] * `xost'
10: local bost = round(_b[sqfeet],.001)
14: twoway ///
15: || lfit rent sqfeet if ost ///
16: || lfit rent sqfeet if !ost ///
17: || , text("yost' `xost' b= `bost' = ///
18: "ywest' `xwest' b= `=round(_b[sqfeet],.001)'") ///
19: , place(ne) ) ///
20: legend(order(1 "Ost" 2 "West") )
```

Advanced example II
1.2 Loops

foreach

A foreach-loop has 7 elements

- The command foreach
- The definition of the name for a placeholder
- The definition of a list-type
- The definition of a list of elements
- A curly opening bracket ( { )
- The commands inside the loop
- A curly closing bracket ( )

Example

. foreach X of varlist np9501-np9507 {
    .    tabulate `X' gender
    .}

List types

. foreach var of newlist r1-r10 {
    .    gen `var' = runiform()
    .}

. foreach num of numlist 1/10 {
    .    replace r`num' = rnormal()
    .}

. levelsof state, local(K)
. foreach k of local K {
    .    di "*[res]k"[txt] has label [res] `:label (state) `k'"
    .}

. foreach piece in You live and learn. At any rate you live. {
    .    display "*piece"
More than one lines

```stata
. foreach var of varlist ybirth income {
    .   summarize `var', meanonly
    .   generate `var'_c = `var' - r(mean)
    .   label variable `var'_c "`var' (centered)"
    .
}
.
. foreach num of numlist 1/10 {
    .   replace r`num' = rnormal(`num'/2,1+`num'/4)
    .   local plots `plots' || kdensity r`num'
    .
}.
. tw `plots', legend(off)
```

Shifting parallel list

```stata
. local i 1
. foreach var of varlist kitchen shower wc {
    .   generate equip`i++' = `var' - 1
    .
}
```

**forvalues**

A forvalues-loop has 6 elements

- The command **forvalues**
- The definition of the name for a placeholder
- A range
- A curly opening bracket ({})
- The commands inside the loop
- A curly closing bracket ()

▷ **Example**

```stata
. forvalues i = 1/10 {
    .   display `i'
    .
}.
. forvalues i = 2(2)10 {
    .   display `i'
    .
}
```

**forvalues** is preferable to **foreach** with a numlist in many situations.

**Example 1**

Display $\eta^2$ between satisfaction with living conditions and various categorical variables for each state.

---

```stata
1: input emarital eedu egender ehcond
2: end
3: levels state, local(K)
4: foreach k of local K {
5:   foreach var of varlist marital edu gender hcond {
```

5
Example 2

Regression coefficients of general life satisfaction on income for each state.

Example 3

Example 2, but keep information about names of states.

Example 4

We add confidence intervals and dress up the graph.
8: } 
9: postclose coefs 
10: 
11: use 'example', clear 
12: gen ub = b + 1.96*se 
13: gen lb = b - 1.96*se 
14: egen axis = axis(b), label(state) reverse 
15: levels axis, local(K) 
16: graph twoway /// 
17: || rspike ub lb axis, horizontal /// 
18: || scatter axis b /// 

1.3 Programs

What is program?

Programs are Stata commands between the Stata commands program and end

. program hello 
  . display "{txt}Hello, world" 
  . end 
  . 
  . hello

Programs are stored into the computer’s memory (RAM).

The problem of redefinition

Stata does not allow you to override a program already saved in memory. You must delete the old version from the RAM before you can create the new version.

. program drop hello 
. program hello 
  . display "{res}Hi, back" 
  . end 
  . 
  . hello

The problem of naming

Stata searches for programs only when it has not found a built in Stata command.

. program q 
  . display "Hello, world" 
  . end 
  . q

Saving programs in do-files

Programs defined with program get lost when you terminate Stata. Saving the program definition in a do-file is a clever workaround.
Automatically loaded Do-Files

Programs in Do-Files saved with the extension .ado are loaded and carried out automatically.

Parsing argument

Positional Arguments

You can parse user input into the execution of a program using positional arguments.

The syntax command

Putting the command syntax at the top of a program is the standard way of parsing user input into programs.
Making programs behave Stataish

`syntax` is the way to make programs behave in a Stataish way.

Example Ado-file

```stata
*! A very simple program for centering a varlist
program mycenter
version 9.2
syntax varlist(numeric) [if] [in] [, Listwise]
tempvar touse
mark `touse'
markout `touse' 'if' 'in'
if "listwise" != "**" markout `touse' `varlist'
foreach var of local varlist {
    summarize `var' if `touse', meanonly
    qui generate `var'_c = `var' - r(mean) if `touse'
    label variable `var'_c "variable label `var' (centered)"
}
end
```

1.4 Step by step: a practical example

Problem setup

Problem: We need a wrapper for creating dummy-variables with labels suitable for `estimates table` and/or `esttab`.

```
tab state, gen(state)
.reg income yedu state2-state16
.estimates table
.estimates table, label
```

Let’s fire up an editor and start creating a program for this.
1.5 Programming style

Acknowledgement

- The following documents style-rules suggested by Cox (2005).
- They are also reprinted in Baum (2009, 244–248).
- Style rules are suggestions. However, I strongly recommend to always follow suggestions by Nick Cox.

Presentation

- `*! version 2.3 Mai 30, 2011 @ 11:07:19 UK`
- Choose good names – new (`findit`), informative, short, no English words
- Group `tempname`, `tempvar`, `tempfile` declarations
- Program error messages
- Use subprograms.

Helpful Stata features

- Use the most recent Stata version
- `syntax`
- `marksample` and `if `touse``
- SMCL to format Output
- `return` or `ereturn`

Respect for datasets

- Do not change the dataset in memory unless that’s what you program is designed for
- Do not use permanent names for files, variables, scalars and matrices. Use `tempfile`, `tempvar`, `tempname`

Speed

- `forvalues` is faster than `foreach`, `foreach` is faster than `while`
- Avoid `egen`
- Do not loop over observations. Never.
- Avoid `preserve`
- Specify variable type, i.e. `gen byte `myvar``
- Consider dropping `tempvars` when they not longer needed
What about ...

- String variables?
- By-ability
- Weights?
- Help-file? [u] 18.11.6 Writing online help and help examplehelpfile
- Verification scripts

2 Programming Mata

2.1 First principles

What is Mata?

- A full-fledged programming language that operates in the Stata environment:
  - Mata programs can be called by Stata
  - Mata programs can call Stata programs

- The language of Mata is designed to make programming functions for matrices real easy.
- Mata is fast because Mata code is compiled into bytecode.

Starting and and stopping Mata

```plaintext
: mata
: 6*7
 42
: end
```

Mata statements

When you type something at the Mata prompt, Mata compiles what you typed and, if it compiled without error, executes it.

```plaintext
: 6*7
 : 42/6
 : sqrt(1764)
```

The above statements are expressions. You can assign the expression to a variable using the = operator:

```plaintext
: answer = 6*7
 : answer
```
**Definition of matrices**

Mata is designed for working with vectors and matrices. The comma is Mata’s column-join operator:

```
: r1 = (3,2,1)
: r1
```

The backslash is Mata’s row-join operator:

```
: c1 = (4\9\12)
: c1
```

We can combine the column-join and row-join operators

```
: m1 = (3,2,1) \ (4, 9,12)
: m1
```

Column-join and row-join operators work on vectors and matrices, too

```
: m2 = c1, r1`
: m2
```

**Matrix Operations**

The standard algebraic operators work on vectors and matrices

```
: r1 + c1`
: r1 * c1
: m1 * c1
```

Algebraic operators preceded with a colon forces element-by-element computations.

```
: m1 :* m2`
: m1 :/ m2`
: m1 :^ m2`
```

**Matrix and scalar functions**

Mata has matrix and scalar functions. The function `invsym()` is a matrix function returning the inverse of a symmetric matrix:

```
: invsym(m1*m1')
```

The function `sqrt()` is a scalar function returning the square root of a scalar. Using scalar functions on matrices forces elementwise calculation:
Loops
Loops belong to the not so frequently used “frequently used” concepts in Mata.

```stata
: X = J(10,10,"empty")
: for (i=1; i<=10; i++) {
>     for (j=1; j<=10; j++) {
>         X[i,j] = "[" + strofreal(i) + "," + strofreal(j) + "]"
>     }
> }
: X
```

Submatrix selection
Selection of sub-matrices is done with subscripts.

```stata
: X[1,1]
: X[1,2),(3,4]
: X[1..10,(3..6)]
: X[(10..1),(6..3)]
: X[(10..1),(6..3)]
: X[| 1 , 3 | 7 , 7]|
```

Writing programs
Mata is a programming language; it will allow you to create your own functions:

```stata
: function deepthought(a)
> {
>     return(a :/: a :* 42)
> }
```

Once we defined the function we can use it:

```stata
: deepthought(2)
: deepthought(answer)
: deepthought(r1)
: deepthought(m1)
```

How Mata works
If you interactively define a function in Mata,

- Mata reads that function
- Mata compiles the program into binary code (object code)
- Mata stores the compiled code in memory
If you call a function,

- Mata reads your “interactive statement”
- Mata compiles what you have typed into the object code
- Mata stores that object code as `<istmt>()` in memory
- Mata executes `<istmt>()`
- Mata drops `<istmt>()`

Making mistakes

If you make a mistake, Mata complains. Later on it will become helpful to understand the difference between compile-time errors and run-time errors.

Here is a compile time error:

```
: 2,,3
invalid expression
r(3000);
```

And this is a run-time error:

```
: y
    <istmt>: 3499 y not found
    r(3499);
```

Getting help

- `. help mata`
- `: help mata`
- Chapter 13 and 14 of Baum (2009)
- The "‘Mata matters’" column in the Stata Journal (Gould, 2005a,b, 2006a,b,c, 2007b,a, 2008, 2009, 2010; Linhart, 2008)
- Studying the source code of others with `viewsource`

2.2 Stata in Mata

Stata interface functions

Mata has several functions that interface with Stata. Here is a list of the more important ones. See help `m4_stata` for a complete list.

- `stata()` executes a Stata command
- The functions `st_view()` and `st_data()` make Mata matrices from a Stata dataset in memory.
- `st_store()` does the opposite: it modifies values stored in current Stata dataset
- `st_local()` Obtain strings from and put strings into Stata macros
- `st_numscalar()` and `st_matrix()` obtain values from and put values into Stata scalars and matrices.
stata()
The function *stata()* lets you perform arbitrary Stata commands from inside Mata.

: stata("use data1, clear")
: stata("drop if mi(income,yedu,ybirth)")
: stata("gen cons = 1")
: stata("regress income yedu ybirth")

If you work interactively, you will perhaps better finish with Mata and type these commands into Stata directly.

Matrices from Stata data
The functions *st_view()* and *st_data()* both return the Stata dataset as a Mata Matrix. *st_data()* creates a *copy* of the data:

: st_data(,,)
: st_data(1,2)
: st_data((1,10),2)
: y = st_data(,,"income")
: X = st_data(,,{"yedu","ybirth","cons"})

*st_view()* , on the other hand, creates a *view* on the data. Otherwise it works very similar:

: y1 = .
: X1 = .
: st_view(y1,,"income")
: st_view(X1,,{"yedu","ybirth","cons"})

Views vs. Copies

- Changing a value in a matrix that is a view also changes the value in the dataset (Which is good and bad).
- Views take only 128 bytes storage. Copies take what it takes to store the data. (Aside: Don’t make copies of views).
- Looping over rows of a copy and using the individual values as scalar is faster than doing the same with views.

Working with views and copies

▷ Example Consider you found the following formulas for a fancy statistical method:

\[
\begin{align*}
\mathbf{b} &= (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y} \\
\mathbf{V} &= s^2(\mathbf{X}'\mathbf{X})^{-1}
\end{align*}
\]
with

\[
    s^2 = \frac{e'e}{n - k}
\]

\[
    e = y - Xb
\]

\[
    n = \text{rows}(X)
\]

\[
    k = \text{cols}(X)
\]

and \(X\) and \(y\) defined analogous to our copies/views above.

**Implementing an estimator**

The following uses the copies to implement the formulas above. However, we could have also used the views.

\[
    : b = \text{invsym}(X'X) \ast X'y
\]

\[
    : e = y - Xb
\]

\[
    : n = \text{rows}(X)
\]

\[
    : k = \text{cols}(X)
\]

\[
    : s2 = (e'e)/(n-k)
\]

\[
    : V = s2 \ast \text{invsym}(X'X)
\]

**Use Stata to display Mata matrices**

Let’s use Stata’s `matlist` to show the results nicely. Start by creating a matrix with results of interest.

\[
    : se = \text{sqrt}([\text{diagonal}(V)])
\]

\[
    : \text{results} = b, se, b/\text{se}, 2\ast \text{ttail}(n-k, \text{abs}(b/\text{se})), b - (1.96\ast \text{se}), b + (1.96\ast \text{se})
\]

Push the Mata matrix into a Stata matrix ...

\[
    : \text{st_matrix}("results", \text{results})
\]

... use Stata commands to set row and column names ...

\[
    : \text{end}
\]

\[
    . \text{matrix rownames results} = \text{yedu ybirth } _\text{cons}
\]

\[
    . \text{matrix colnames results} = \text{Coef Std_Err t sig lower upper}
\]

and display the results in comparison to Stata’s `regress`.

\[
    . \text{matlist results, rowtitle(income) border(rows)}
\]

\[
    . \text{reg, noheader}
\]

**Knowing where one’s towel is**

In practice, you might find if convenient to implement regression analysis as an intermediate step yourself instead of processing thru the results of official Stata’s `regress`. 

16
Don’t!

- Missing values and Perfect collinearity
- Profit from Stata developments (Factor variables)
- Accuracy problems with large datasets: Variables should have similar variance. Order of values matter when taking sums.
- Problems increase with parallel computing.

2.3 Mata in Stata

The function of Mata for Stata

- Mata is not a replacement for ado-files. Rather, Mata is used to create subroutines used by ado-files.
- Ado-file parse user input and create a call to a Mata function from it.
- The Mata function does the statistical stuff and returns back the results to the Ado-file.
- The Ado-file takes the results from Mata and produces the output.

Let me explain this is step-by-step.

Problem 1: Generalize Mata code

Consider what we typed before.

```plaintext
begin myreg.do.1
1: mata:
2: y = .
3: X = .
4: st_view(y,.,"income")
5: st_view(X,.,("yedu","ybirth","cons"))
6: b = invsym(X'X)*X'y
14: st_matrix("results", results)
15: end
end myreg.do.1
```

We might want to have a function that does this for arbitrary variables.

Solution: Define a function

```plaintext
begin myreg.do.2
1: version 11  // <- new
2: mata:
3: mata clear  // <- new
4: function myreg(yvar,xvars)  // <- new
5: |
6: y = .
7: X = .
8: st_view(y,.,yvar)  // <- new
9: st_view(X,.,tokens(xvars))  // <- new
10: b = invsym(X'X)*X'y
19: }  // <- new
20: mata mosave myreg(), replace  // <- new
21: end
end myreg.do.2
```
Problem 2, Call Mata program

Realize what is necessary to call the Mata routine:

```
. run myreg.do.2
. mata: myreg("income", ("ybirth yedu cons") )
. matrix dir
. matrix list results
```

Solution: Ado-code

```
begin myreg ado.1
1: use data1, clear // <- Data required
2: drop if mi(income,yedu,ybirth) // <- Listwise deletion
3: gen cons = 1 // <- We want a constant
4: mata: myreg("income",("yedu ybirth cons") ) // Call it
5: end myreg ado.1
```

Problem 3, Produce output

Remember what was necessary to produce some output

```
. myreg income ybirth yedu
. matrix list results
```

Solution: Ado code, again

```
begin myreg ado.3
15: // Output
16: matrix rownames results = yedu ybirth _cons
17: matrix colnames results = Coef Std_Err t sig lower upper
18: matlist results, rowtitle(income) border(rows)
end myreg ado.3
```
Where to store Mata functions

Ado-file  Put the Mata code below the program in the ado-file. The Mata code is private for the ado file, then. Mata code gets compiled when the ado-file is called the first time during a Stata session.

Do-file  We did this above. Running the do-file produces a .mo-file with the compiled code. If .mo-file is stored along the search path, all programs can work with it. Use the extension .mata instead of .do if you do this.

As a library  Same as above, but more than one function. The compiled libraries get a .mlib extension, and the names all start with l.

Note that you can distribute your Mata functions with and without the source-code.

2.4 Common problems proper solutions

Debugging

Debugging Mata functions can be tedious:

- Error messages are not quite informative
- Much space between user input and Mata function
- No feedback from Mata on position of error.

Don’t panic:

- Put noisily in front of call to Mata
- Put commands that produce output in the Mata code here and there.
- Always set matastrict on (see below)

Line breaks

In Mata, lines breaks do not (necessarily) end a command. Line breaks only end a command when it makes sense that command ends there.

```
. mata:
: x = 6 *
> 7
: x
: end
```

Note also that you can use the semicolons to force an end of a command. It may be used to place two statements on the same physical line:
Macros in Mata functions

Stata programs use macros. Mata programs do not. Forget that macros exist!

- After having forgotten that macros exist, note that Stata macros are accessed at compile-time.

```mata
. local x = 42
. mata:
: function deepthought(input)
> { 
> return(input :/ input : `x')
> }
: end
. mata: deepthought(2)
. local x 36
. mata: deepthought(2)
```

- Now, try again with `st_local()'.

An example of good use of Stata macros

```
begin myreg.ado.5
5: // Necessary Prepartions
6: preserve
7: quietly keep if `touse'
8: gettoken depvar indepvars: varlist
9:
10: // Note: generation of constants dropped
11:
12: // Call mata
13: mata: myreg("depvar",("indepvars")) // <- Note
14: end myreg.ado.5
```
```
begin myreg.do.3
4: function myreg(yvar,xvars)
5: st_view(X,.,tokens(st_local(xvars))) // <- New
6: X = X,J(rows(X),1,1) // <- New
end myreg.do.3
```

2.5 Programming Style

Use functions instead of loops

▷ Example Instead of

```mata
. mata
: function calcsum(varname)
> {
```

. mata: x = 6*7; x

```mata
: x = . ; sum = 0 ; st_view(x,.,varname)
: for (i=1;i<=rows(x);i++) {
:     sum = sum + x[i]
: }
: sum

: mata: calcsum("ybirth")
```

**Take advantage of cross products**

Typing $X'X$ is easy in Mata, but should be avoided. Use `cross()`, `crossdev()` and `quadcross()` instead.

**Example**

```mata
: y = X = .
: st_view(y, ., "income")  
: st_view(X, ., "ybirth yedu")
: XX = cross(X,1 , X,1)
: Xy = cross(X,1 , y,0)
: invsym(XX)*Xy
```

Read carefully the remarks in `help mf_cross`

**Use Declarations**

Use declarations as much as you can. Declarations can occur in three places:

- In front of a function definition (“function declaration”)
- Inside the parentheses defining the function’s arguments (“argument declarations”)
- In the body of the program (“variable declarations”)

```mata
function_declaration function_name ( argument_declarations )
{
    variable_declarations
}
```

Also see `help m2_declarations` and subsequent slides.

**Function declarations**

- Function declarations state what kind of variable the function will return.
• Function declaration announce to other programs what to expect. If the other program expects a real scalar, but our function returns a string matrix, Stata complains before running our function and trying to execute the other program.

• Including function declarations makes debugging easier.

Syntax of function declarations is

```
function | type | function] | void [function]
```

where function is just a word, type will be explained below, and void means that the function returns nothing.

**Argument declarations**

• Argument declarations state what kind input the function expects.

• If specified, Mata will whether the caller attempts to use your function incorrectly before executing the function.

• Including function declarations makes debugging easier.

Syntax of argument declarations is

```
[type]argname [, [type]argname [, ...]]
```

where argname is the name the argument receives and type will be explained below.

**Variable declarations**

• Variable declarations state which variables are used inside the program.

• Variable declarations are optional unless you set matastrict on

• Better run-time error messages (Debugging)

• Whether declared or not, variables are private to the function

Syntax of variable declarations is

```
[type]varname [, [type]varname [, ...]]
```

where varname is the name of the variable to be declared and type will be explained below.

**Variable types**

The syntax of declarations involve the concept of a variable type. The type of a variable has two parts:

- **eltype** the type of the elements the variable contains
- **orgtype** how those elements are organized

<table>
<thead>
<tr>
<th>eltype</th>
<th>orgtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>scalar</td>
</tr>
<tr>
<td>string</td>
<td>rowvector</td>
</tr>
<tr>
<td>complex</td>
<td>colvector</td>
</tr>
<tr>
<td>pointer</td>
<td>vector</td>
</tr>
<tr>
<td>numeric</td>
<td>matrix</td>
</tr>
<tr>
<td>transmorphic</td>
<td>transmorphic</td>
</tr>
</tbody>
</table>
Tables are sorted from the specific to the general. Specific types should be prefered.

**Exercise on declarations**

What declarations should be used for the functions we have implemented so far:

- `deepthought()`
- `calcsum()`
- `myreg()`

**Be strict**

If you put `mata set matastrict on` at the top of you programs variable declarations are pre-scribed. Programing Mata then becomes more cumbersome, but Mata produces more efficient code (making functions run faster):

```stata
begin myreg.do.4
4: mata set matastrict on // <- New
end myreg.do.4
```

```stata
run myreg.do.4
variable y undeclared
```

```stata
begin myreg.do.5
7: real colvector y // <- New
16: real matrix results // <-New
end myreg.do.5
```

So long . . .

... and thanks for all the fish.

**Bibliograpy**

**References**

Baum, C. F. 2009. *An Introduction to Stata Programming*. College Station: Stata Press.


