Introduction to Mata

A Matrix programming language inside Stata

Roy Mill

October 2010

What's, Why and When Mata

- Mata is like a simple MATLAB-style environment within Stata.
- It allows easier, faster and more flexible matrix operations than Stata's regular matrix operations.
- It is also not limited in size as the regular Stata matrices, which are limited by `matsize`.
- It allows to write functions, which are impossible in non-Mata Stata.
- However, it is far less developed than MATLAB in terms of toolkits, flexibility, etc.
  - Use it mainly when you need it as an extension to a project you already do in Stata, rather than as a MATLAB replacement.

Invoking Mata

- We can start Mata either in interactive mode or in a .do or .ado file.
- To start it interactively:

```
. mata                                       <- type mata to enter Mata
---------- mata (type end to exit) ---
: 2+2                                        <- type Mata statements at the
  4                                             colon prompt

: end                                        <- type end to return to Stata
--------------------------------------
. _                                          <- you are back to Stata
```

- When we return to Stata, all variables in Mata are still there. We can invoke Mata again and keep using them.

- For a more intricate explanation on how to invoke mata see `help m3_mata`

Scalars

- Mata does not have macros. It has scalars that can be very simply defined:

  ```
  <scalar name> = <expression>
  ```

- The expression can be a real number, a complex number or a string:

  ```
  x = 4
  a = sqrt(C(-x))    // C(x) converts x to a complex number
  b = "Hello"
  ```

- Assigning an expression to a scalar suppresses output.
- Real, complex and strings are called **eltype**s – they are the elements' types.
- A scalar is an **orgtype**. Other orgtypes are vectors and matrices.
- Each variable in Mata has both an orgtype and an eltype.

Vectors and Matrices

- Matrices are defined exactly like scalars. The only difference is that <expression> is now a vector or a matrix.
- To define a matrix explicitly, use the column-join operator (,) or the row-join operator (\):

```
A = (1 , 2 \ 3, 4)
B = J(4, 2, x)  // J(r,c,x) creates an r-by-c matrix of x's
C = ("NW", "NE" \ "SW", "SE") // strings are also fine
V = (1..50)  // V will be a row vector with values from 1 to 50
V = (1::50)  // V will be a column vector with values from 1 to 50
```

- A few functions you will probably find useful:
  - `rows(X)` and `cols(X)` return the number of rows and columns
  - `missing(X)`, `rowmissing(X)`, `colmissing(X)` returns the number of missing values overall, by rows and by columns, respectively.
  - `sum(X)`, `rowsum(X)`, `colsum(X)` returns the sum of values overall, by rows and by columns, respectively.
  - More on functions like these in `help m4_utility`

Referring to elements and submatrices

To refer to elements or ranges inside a matrix we use either “list subscripting” or “range subscripting”.

List substcripts look like:

```
A[1,2]          // the 1st-row and 2nd-column element
V[5]            // for vectors you can specify only one index
B[(1\4),(1,2)] // submatrix consists of 1st and 4th rows of matrix B
                // (cols 1 and 2)
B[1::3,1..2]   // rows 1-3 and columns 1-2 of matrix B
A[2,.]          // 2nd row (all columns) of matrix A
A[2,]           // same as above
```

Alternatively, range subscripts look like:

```
A[|1,2|]      // same as A[1,2]
V[|5|]        // same as V[5]
B[|1,1\3,2|] // submatrix of B where top-left corner is (1,1) and bottom
              // bottom-right corner is (3,2). Same as B[1..3,1..2]
D[|3,.|]      // third row of D. same as D[3,.]
D[|2,1\.,4|] // From the (2,1) corner to the (last-row,4) corner.
              // equivalent to D[2::rows(D),1..4]
```

Operators

In addition to the usual arithmetic operators we have:

- multiple assignment - unlike many other languages, in Mata we can assign another variable inside <expression>:

```
a = b = 2 + 3 // Equivalent to writing b = 2 + 3 and then a = 2 + 3

stat = (numerator = beta - 1) / (denominator = sqrt(varbeta))
```

  - Be careful with confusing = with ==. The former is assignment, the latter is comparison

- colon operators - element-by-element operations (like dot operators in MATLAB)

```
A = B = (1, 2 \ 3 , 4)
A * B   // will give the matrix product of A and B
A :* B  // will give the element-by-element product of A and B
A :+ 3  // will add 3 to all elements of A
A :== 4 // will return a matrix with typical element I{a_ij == 4}
```

  - ... more on those in `help m2_op_colon`

Many more operators can be found through `help m2_intro`

Flow control 1 – `if`

Very similar to the `if` command in Stata, with two exceptions:

1. The condition must be encapsulated in parenthesis.
2. If there is only one command to run if the condition is true, you can leave the curly brackets out.

```
if (x == 6) {
   // do stuff
}
else { // else should start in a separate line
   // do something else
}

if (x == 6) printf("x equals 6")
else printf("x does not equal 6")
```

Flow control 2 – `while` and `do`

Both `while` and `do` loop until some condition is satisfied.

`while` first checks the condition and then starts the first iteration.

`do` starts the first iteration and checks the condition only when it's done.

```
while (<cond-exp>) {
   <statements>
}

// Alternatively...
do {
   <statements>
} while (<cond-exp>)
```

Flow control 3 – `for`

Forget about `foreach` and `forvalues`. In Mata we do it like the pros:

```
for (<initialization> ; <break-condition> ; <recurring-action>) {
   <statements>
}
```

For example:

```
for (i=1 ; i<=80 ; i++) {
   printf("i == %g\n", i)
}
```

incrementation operators - operators that increase or decrease by 1. They differ also by whether they do the incrementation before or after the command is executed.

```
: i++ // say i was 4 before. will first output 4 and then set i = 5
: ++i // say i was 4 before. will first set i = 5 and then output 5
: i-- // say i was 4 before. will first output 4 and then set i = 3
: --i // say i was 4 before. will first set i = 3 and then output 3
```

Flow control 4 - `continue` and `break`

It might not be the best programming practice, but sometimes you want to skip the execution of an iteration if some condition within the loop is satisfied.
`continue` will do just that.

```
for (i=1908 ; i<=2008 ; i = i + 4)  {
    if (i == 1916 | i == 1940 | i == 1944)  continue
    printf("Summer Olympic games were held in %g\n", i)
}
```

Sometimes you will want to exit the loop altogether – not just skip an iteration – if some condition is satisfied.
For this you will need `break`.

```
for (i=1 ; i<=length(V) ; i++) {
   if (V[i] == 13) {
      printf("Found 13 in cell %g", i)
      break
   }
}
```

Functions in Mata vs. Programs in Stata

- In Stata you can write programs that are like functions that don't return output directly.
- In Mata we can write real functions that return values so we can assign them to variables directly.
- The Cobb-Douglas function in Stata can only be run like this:

```
predictgdp, k(15) l(17) // predictgdp will save the output to r(Y)
scalar y = r(Y)         // saving the result to a scalar
```

- In Mata we can write it in one line:

```
y = predictgdp(15,17) // assuming predictgdp's 1st argument is capital
```

Defining a Function

To define a function we need to define the inputs and output:

```
<output-type|function> <func-name>([arg1type] <arg1name>,
                                  [arg2type] <arg2name> ...) {
    // ... do what you need to do ...
    [return (<expression>)]
}
```

where output-type or arguments-types are specified with `eltype` and then `orgtype`.

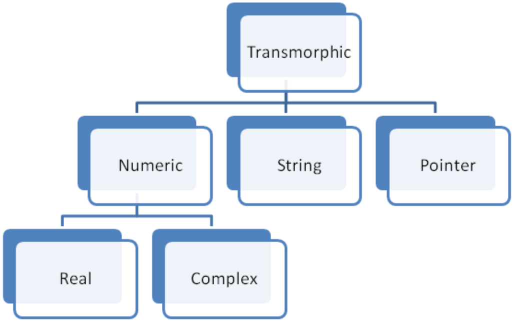
```
real scalar predictgdp(real scalar K, real scalar L) {
   A = 65 ; alpha = 1/3
   return (A * K ^ alpha * L ^ (1-alpha))
}
```

Alternatively, though less efficiently:

```
function predictgdp(K, L)
   //  ...
}
```

What happens when we dont specify a type

If you don't specify a type, Mata assumes it to be a transmorphic matrix. `transmorphic` stands for any `eltype`:



Specifying a type makes execution faster than leaving it transmorphic. It can also make debugging easier if you don't call the function right.

Optional Arguments

You can make your functions easier to call by specifying optional arguments.

When you have some arguments for which you have a reasonable default value, you can set them optional. For example, α is most likely to be $1/3$.

```
real scalar gdp(real scalar A, real scalar K, real scalar L, |real scalar alpha) {
   if (args() < 4) {
      alpha = 1/3
   }
   return (A * K ^ alpha * L ^ (1-alpha))
}

// Both will work. Second will set alpha to 0.45 instead of 1/3.
y            = gdp(65, 15, 17)
yhighcapshare = gdp(65, 15, 17, 0.45)
```

See `m2_optargs` for more details.

Variables' Scope

Inside a function, you will not be able to reach variables that were defined outside the function. Nor will you be able to reach from outside variables defined within a function.

To make the function “see" a variable set outside of it, you have three options mainly:

- Pass the value of the variable to the function as an argument (like K and L above)
- Pass the address of the variable to the function as a pointer variable (will not be discussed, see `help m2_pointers` if interested)
- Tell the function you are referring to the outside-defined (global) variable (not a good practice though):

```
A = 65 ; alpha = 1/3
function predictgdp(K, L) {
   external A, alpha
   return (A * K ^ alpha * L ^ (1-alpha))
}
```

Void Functions

Sometimes write functions that just do stuff. You don't need any answer from them.

In other programming languages they are called procedures, or subroutines. In Stata it's like writing a program.

Put the word `void` before the function name (or the word function), when you declare the function, if you don't want it to return anything.

Again, you don't have to – you can just not return a thing. But if you want to be strict and more readable you might want to.

It's also useful to know what's void in order to not-get-scared when you see it in a function somebody else wrote.

Generalizing Scalar Functions to Matrix Functions

Like in MATLAB, in many cases you want to vectorize your problem.

What will you do if you have a vector of real numbers and you want to apply a function on all of them, but the function expects a scalar?

```
real scalar cube_minus_4(real scalar x) {
   return (x ^ 3 - 4)
}

V = (1..10)
for (i = 1 ; i <= length(V) ; i++) {
   V[i] = cube_minus_4(V[i])
}
```

However, we can generalize the function to expect matrices (and scalars are a special case). One thing we need to change is the declaration. The other is the operators:

Generalizing Scalar Functions to Matrix Functions

```
real matrix cube_minus_4(real matrix x) {
   return (x :^ 3 :- 4) // Note that we changed the operators
}

V = cube_minus_4(V)
```

Or, back to our famous example:

```
real matrix gdp(real matrix A, real matrix K, real matrix L,
                                                |real scalar alpha) {
   if (args() < 4) {
      alpha = 1/3
   }
   return (A :* (K :^ (alpha)) :* (L :^ (1-alpha)))
}

A = 65
// Suppose we have vectors of K and L (same size) for different plants.
Y = gdp(A, K, L)
```

Dropping a Function from Memory

You'll most likely want to change a function while debugging it.

`mata drop` allows you to drop variables and functions from the memory.

To drop a variable (scalar/matrix/vector), use:

```
: mata drop X
```

To drop a function (you'll have to before you redefine it),

```
: mata drop gdp()
```

The () lets Stata know it's a function that we want to drop.

To clear all Mata memory, just type

```
: mata clear
```

Communicating with the Stata Environment

Up until now we worked solely within Mata environment. No data that is stored in Stata was used.

Everything I'm saying here, and much more, is well documented in `help m4_stata`. I will just guide you through the first steps, and the problem set will guide you through the rest.

To get, or set, a macro in Stata, we use either `st_local` or `st_global`:

```
string scalar st_local(string scalar localName)            // Getting value
st_local(string scalar localName, string scalar localValue) // Setting value
```

```
. local myLocal "Hello"
. mata
: x = st_local("myLocal") /* x will now contain "Hello" */
: st_local("myLocal", x + " world!")
: end
. di "‘myLocal’" /* will print "Hello world!" */
```

Communicating with the Stata Environment

Recall that even when you think you store numbers in locals and globals, you actually have strings there, so to perform numeric operations on them you should convert them to numbers

```
. local k = 15 // remember, these are saved as strings
. global l = 17 // remember, these are saved as strings

. mata
: k_mata = strtoreal(st_local("k"))
: l_mata = strtoreal(st_global("l"))
: gdp_mata = gdp(kmata, lmata)
: gdp_local_name_in_stata = "gdp_pred"
: st_local(gdp_local_name_in_stata, strofreal(gdpmata))
: end

. di "GDP is ‘gdp_pred’"
```

Getting and Setting Stata Scalars and Matrices

Same thing for scalars and matrices. Just need to handle string and numeric scalars differently (no string matrices - so no problem there)

```
. scalar x = 40
. scalar y = "hello"
. matrix Var = e(V)

. mata
: x_mata = st_numscalar("x")       // get
: y_mata = st_strscalar("y")       // get
: V = st_matrix("Var")             // get

: st_numscalar("x", x_mata ^ 2)    // set
: st_strscalar("y", "what's up?")  // set
: stmatrix("Var", V'*V)            // set
: end

. di x
. di y
. matrix list Var
```

Getting and Setting Variables in the Dataset

Three types of functions allow Mata to get and set Stata's dataset variables:

1. `st_data()` copies dataset variables into a Mata matrix.
2. `st_store()` copies a Mata matrix into dataset variables.
3. `st_view()` creates a matrix in Mata that is linked to the dataset.

`st_view()` lets you both "get" and "set" Stata's variables at the same time. If we change elements of a Mata matrix set by `st_view()`, elements in the corresponding Stata variables will be changed accordingly.

Up-side: `st_view()` takes less time to set up the matrix.
Down-side: after setting up, access time to the matrix is slower.

There are some other issues with `st_view` that you should be aware of. See `help st_view` for details.

Using `st_data()`

```
real matrix st_data(real matrix obs, rowvector vars,| scalar selectvar)
```

- 1st argument, **`obs`**, is a matrix of observations ranges
- 2nd argument, **`vars`**, has to be a row vector of variables' order or names.
- 3rd argument, **`selectvar`**, further restricts the fetched rows to those in which the variable is nonzero.
    - Usually used to kick out observations with a missing value in one of the variables of interest.

Examples will be best:

```
a = st_data(1, 2)       // returns the first observation on the 2nd var.
B = st_data(., 2)       // returns all obs of the second variable.
C = st_data((1\2\5), 2) // returns obs 1, 2, and 5 on the second var.
D = st_data((1,5), 2)   // returns obs 1 through 5 on the second var.
E = st_data((1,5\7,9), 2) // returns obs 1-5 and 7-9 on the second var.
F = st_data(2, (3,1,9)) // the 2nd ob's values of vars 3, 1 nd 9.
G = st_data(., "educ")  // returns all obs on educ variable.
H = st_data(., ("educ", "income")) // returns all obs of educ and income
```

Using st_data()

```
. gen byte mySample = female == 1 & rural == 0
. markout mySample educ income married // remove obs with missing values
                                      // in any of the variables

. mata
: Y = st_data(., "income", "mySample")
: X = st_data(., ("educ", "married"), "mySample")
: X = X , J(rows(X),1,1)          // add constant to the right.
: b = invsym(X'*X)*X'*Y
: b = invsym(cross(X,X))*cross(X,Y) // Equivalent to previous line

: st_matrix("betas", b)
: end

. matrix list betas
```

Using `st_store()`

```
void st_store(real matrix obs, rowvector vars, real matrix store_me)
void st_store(real matrix obs, rowvector vars, scalar selectvar,
                                                     real matrix store_me)
```

- The first two arguments are similar to **`st_data`**. They specify what observations and variables to save into.
- The last argument must be the matrix you want to store in the variables chosen.
    - **`selectvar`** is optional (even though it's not last).

```
. gen predicted = .

. mata     // If you did not run "clear mata" or exit,
           // X and b are still saved in Mata
: pred = X*b
: st_store(., "predicted", "mySample", pred)  // STORING the predicted value
: end

. su predicted
```

Using `st_addvar()`

- `st_store` requires that the variables you are storing the matrix into already exist in the dataset.
- If you want your program to create new variables rather than using existing ones, you can:
  - Create them before invoking mata using the `gen` command
  - Call any Stata command from mata using `: mata stata <stata-command>`
  - Use Mata's `st_addvar()` function to create a new variable.
    - You know the drill... `help mata st_addvar` for more details.

Breaking space-separated lists to row vectors – `tokens()`

- In Stata you get lists as long strings of spaced words.
  - e.g `"educ age experience union"`
- In Mata you have to use lists of strings.
  - e.g `("educ", "age", "experience", "union")`
- To translate a space-separated list to a string vector, use the function `tokens`:

```
: tokens("Hi, how are you today?")
// will print:
            1        2        3        4        5
    +----------------------------------------------+
  1 |     Hi,      how      are      you   today?  |
    +----------------------------------------------+

// Another example
. local covariates "educ married rural urban"
. mata

: indep_vars_list = st_local("covariates")
: X = st_data(., tokens(indep_vars_list), "mySample")
```