Kurt Schmidheiny

Coding with Mata in Stata

Version 1.0, errors likely, feedback welcome.

1 Introduction

Mata is a matrix algebra language which is available in Stata since version 9.

Stata and mata commands are set in `Courier`. Options in `[brackets]` are optional.

2 Where to get help

The two Stata manuals *Mata Matrix Programming* provide systematic information about MATA commands. It also discusses the implemented numerical methods.

The online help in Stata describes the use of all Mata commands with its options. However, it does not explain the numerical properties as in the Reference manual. We can start the online query in the Command window by

```
help mata command
```

For example, help for the cholesky decomposition is asked for by

```
help mata cholesky()
```

If you don't know the exact expression for the command, you can search for Mata commands in the Stata documentation by issuing the command

```
search mata word
```

Search commands are answered in the result window. Alternatively, you can display the result in the Viewer window by issuing the command

```
view search mata word
```

You can also use the Stata online help in the menu bar: Help/Search....

3 Using Mata in Stata

3.1 Command line use

Mata is directly accessed from the Stata command window. Type the stata command

```
mata
```

and all subsequent commands will be interpreted as Mata commands. Type

```
end
```

to return to the normal Stata command prompt.

Note: All matrix and function definitions are kept in the Mata workspace when you `end` mata and accessible when you re-enter `mata`.

3.2 Use in .do files

Mata commands can also be used in do files. A sequence of Mata commands is preceded by the Stata command `mata` and terminated by `end`. There can be several such sequences which access the same Mata workspace.

3.3 Use in .ado files

See section 7 for use of Mata to produce new Stata commands.

4 Matrix Algebra with Mata

4.1 Creating matrices, vectors and scalars

4.1.1 Building a matrix

In order to build a matrix you have two options. You can declare the whole matrix separating columns by a comma and rows by a backslash. For example a 2x2 matrix

```
A=(1, 2 \ 3, 4)
```

Or you can first declare an empty matrix with e.g. 2 rows and 3 columns

```
B = J(2, 3, .)
```

and then fill element by element

```
B[1,1] = 5
B[1,2] = 6
B[1,3] = 7
B[2,1] = 8
B[2,2] = 9
B[2,3] = 10
```

You can give a matrix whatever name you want except for reserved words (see `help m2 reswords`). Names consist of any combination of letters, numbers and underscores "_" but cannot start with a number. Mata is sensitive with respect to the upper and lower case.

If a variable has previously been assigned a value, the new value overrides value and dimensions of the predecessor.

You can display the value of variable B by simply typing its name

```
B
        1    2    3
    +----------------+
  1 |   5    6    7  |
  2 |   8    9   10  |
    +----------------+
```


4.1.2 Building a matrix out of submatrices

You can build matrices out of several submatrices. Suppose you have submatrices A to D.

```
A=(1, 2 \ 3, 4)
B=(5, 6, 7 \ 8, 9, 10)
C=(3, 4 \ 5, 6)
D=(1, 2, 3 \ 4, 5, 6)
```

The matrix $E = \left|\begin{array}{cc} A & B \\ C & D \end{array}\right|$ is built using the column join operation "," and the row-join operator "\":

```
E = (A, B \ C, D)
```

or

```
E = A, B \ C, D
```

which results in

```
        1    2    3    4    5
    +--------------------------+
  1 |   1    2    5    6    7  |
  2 |   3    4    8    9   10  |
  3 |   3    4    1    2    3  |
  4 |   5    6    4    5    6  |
    +--------------------------+
```

4.1.3 Creating a string matrix

Mata can also deal with string matrices which are matrices with string elements. To create a string matrix, we enclose its elements in quotation marks.

```
S=("Thijs", "Kurt" \ "Ghazala", "Gabrielle")
```

which results in

```
S
                1           2
    +--------------------------+
  1 |      Thijs        Kurt   |
  2 |    Ghazala   Gabrielle   |
    +--------------------------+
```

4.1.4 Creating a vector

A vector is simply a $1 \times n$ or $m \times 1$ matrix. A column vector is created as

```
f = (1, 2, 3)
```

A column vector is created either by

```
f = (1 \ 2 \ 3)
```

or by adding a apostrophe (transposition of a matrix) to the row vector

```
f = (1, 2, 3)'
```

You can declare a column vector with a series of number, e.g. from 3 to 5 as

```
g = (3::5)
g
       1
    +-----+
  1 |  3  |
  2 |  4  |
  3 |  5  |
    +-----+
```

and the corresponding row vector as

```
g = (3..5)
g
       1   2   3
    +-------------+
  1 |  3   4   5  |
    +-------------+
```


4.1.5 Creating a scalar

A scalar is given by a 1×1 matrix. For example

```
a = 2
```

Note: standard Stata matrix commands distinguish between scalars and 1×1 matrices.

4.2 Accessing data from Stata

Consider the example dataset auto.dta which you can download from the Stata (not Mata) command prompt.

```
webuse auto.dta
```

4.2.1 Creating a copy of data

st_data(*colvector*, *rowvector*) copies observations of *numeric* variables from the Stata dataset into a Mata matrix. For example,

```
X = st_data(.,("mpg", "rep78", "weight"))
```

creates a new matrix X with all observations in rows and the variables mpg, rep78, and weight in columns. A subsample of observations, e.g. observations 1 to 5 and 7 to 9, is created by

```
X = st_data((1::5\7::9),("mpg", "rep78", "weight"))
X
          1      2      3
    +----------------------+
  1 |    22      3   2930  |
  2 |    17      3   3350  |
  3 |    22      .   2640  |
  4 |    20      3   3250  |
  5 |    15      4   4080  |
  6 |    26      .   2230  |
  7 |    20      3   3280  |
  8 |    16      3   3880  |
```

Alternatively, variables can be accessed by the number of the respective column

```
X = st_data((1::5\7::9),(3, 4, 7))
```

`st_data(`*`matrix`*`, `*`rowvector`*`)` allows to choose sequences of observations by providing a r x 2 matrix as first argument. For example

```
X = st_data((1,5\7,9),("mpg", "weight", "rep78"))
```

picks observations 1 to 5 and 7 to 9.

Rows with missing values are omitted if a third argument, set to zero, is provided

```
X = st_data((1::5\7::9),("mpg", "weight", "rep78"), 0)
```

Observations can also be selected based on a Stata dummy variable, e.g. touse

```
X = st_data(.,("mpg", "weight", "rep78"), "touse")
```

4.2.2 Creating a view on the data

`st_view(X, `*`colvector`*`, `*`rowvector`*`)` creates a matrix X that is a view onto the current Stata dataset. For example

```
X = .
st_view(X, ., ("mpg", "weight", "rep78"))
```

creates a new matrix X which is a view on all observations of the variables mpg, weight, and rep78. Again the 3rd argument indicates whether missing values or particular observations are to be used:

```
st_view(X, ., ("mpg", "weight", "rep78"), 0)
st_view(X, ., ("mpg", "weight", "rep78"), "touse")
```

`st_subview(Y, X, `*`colvector`*`, `*`rowvector`*`)` creates a new view matrix Y from an existing view matrix X. For example

```
st_view(X, ., .)
st_subview(Y, X, (1::5\7::9), (3,4,7))
```


creates Y as rows 1 - 5 and 7 - 9 of columns 3,4, and 7 of matrix X.

`st_subview(Y, X, `*`matrix`*`, `*`matrix`*`)` allows to choose sequences of observations and variables by providing a r x 2 matrix as 3rd argument and a 2 x c matrix as 4th argument. For example

```
st_subview(Y, X, (1, 5\7, 9), (3,7\4,7))
```

creates Y as rows 1 - 5 and 7 - 9 of columns 3 - 4, and 7 - 7 of matrix V. See `help m5 st_subview()` for the various rules to pick the submatrices.

A memory efficient way to access data for use in e.g. linear regressions is

```
M = X = y = .
st_view(M, ., ("price", "mpg", "weight", "rep78"), 0)
st_subview(y, M, ., 1)
st_subview(X, M, ., (2\.))
```

4.2.3 Copy vs. view

Both `st_data` and `st_view` fill in matrix X with the same data. `st_data()` is generally faster than `st_view` but uses less memory. `st_view` also allows (or risks) changing the value of the underlying data. For example,

```
X[2,1] = 123
```

after `st_data()`, changes the value in the matrix X, but the Stata dataset remains unchanged; after `st_view()`, it would cause the value of mpg in the second observation to change to 123.

Note: There are two functions `st_sdata()` and `st_sview()` for use with string variables.

4.3 Managing the Mata workspace

To see all existing variables in the Mata workspace, issue the command

```
mata describe
```

The command

```
mata clear
```

deletes all variables from the Mata workspace without disturbing Stata.

Specific matrices can be deleted by

```
mata drop namelist
```

For example, matrix A and vector f are deleted with

```
mata drop A f
```

4.4 Basic matrix manipulations

4.4.1 Transposition of a Matrix

The transpose of a matrix is created with an apostrophe

```
B
        1    2    3
    +----------------+
  1 |   5    6    7  |
  2 |   8    9   10  |
    +----------------+

B'
        1    2
    +-----------+
  1 |   5    8  |
  2 |   6    9  |
  3 |   7   10  |
    +-----------+
```

4.4.2 Using partions of matrices

One of the most basic operation is to extract partitions of a matrix. Consider matrix E


```
E
        1    2    3    4    5
    +---------------------------+
  1 |   1    2    5    6    7  |
  2 |   3    4    8    9   10  |
  3 |   3    4    1    2    3  |
  4 |   5    6    4    5    6  |
    +---------------------------+
```

Scalars from individual elements are extracted by using subscripts in square brackets e.g.

```
E[2, 3]
8
```

A row vector from a row of a matrix is extracted by e.g.

```
E[2,.]
        1    2    3    4    5
    +---------------------------+
  1 |   3    4    8    9   10  |
    +---------------------------+
```

and a column vector from a column of a matrix by e.g.

```
E[.,3]
        1
    +-----+
  1 |  5  |
  2 |  8  |
  3 |  1  |
  4 |  4  |
    +-----+
```

A contiguous submatrix e.g. consisting of rows 2 to 4 and columns 2 to 5 is extracted by *range subscripts* which mark the top-left and the bottom-right element of the submatrix

```
E[|2, 2\ 3, 4|]
        1   2   3
    +-------------+
  1 |  4   8   9  |
  2 |  4   1   2  |
    +-------------+
```

or alternatively be specifying a contiguous range of rows and columns

```
E[(2::3), (2..4)]
       1   2   3
    +-------------+
  1 |  4   8   9  |
  2 |  4   1   2  |
    +-------------+
```

Note: the first alternative is performed faster than the latter.

A submatrix consisting of individual rows and columns e.g. rows 1 and 4 and columns 3, 5 and 2 can be formed as

```
 E[(1\ 4), (3, 5, 2)]
       1   2   3
    +-------------+
  1 |  5   7   2  |
  2 |  4   6   6  |
    +-------------+
```

More generally, the two arguments can be predefined vectors. For example,

```
r = (1\ 4)
c = (3, 5, 2)
E[r,c]
       1   2   3
    +-------------+
  1 |  5   7   2  |
  2 |  4   6   6  |
    +-------------+
```

Note: Stata considers it good style to specify the first subscript vector as column vector and the second subscript vector as row vector. Though this is not necessary.

More partitioning rules can be found under `help m2 subscripts`.

4.4.3 Making vectors from matrices and reverse

The Mata function `vec()` transforms a matrix into a column vector with one column stacked onto the next

```
B
       1    2    3
    +----------------+
  1 |   5    6    7  |
  2 |   8    9   10  |
    +----------------+

b = vec(B)
b
       1
    +------+
  1 |   5  |
  2 |   8  |
  3 |   6  |
  4 |   9  |
  5 |   7  |
  6 |  10  |
    +------+
```

The Mata functions `rowshape` or `colshape` reverse this vectorization

```
rowshape(b, 3)'
       1    2    3
    +----------------+
  1 |   5    6    7  |
  2 |   8    9   10  |
    +----------------+

colshape(b, 2)'
       1    2    3
    +----------------+
  1 |   5    6    7  |
  2 |   8    9   10  |
    +----------------+
```

4.4.4 Extracting the diagonal and creating diagonal matrices

`diagonal(A)` extracts the diagonal of A and returns it in a column vector:

```
A = (1,2,3\4,5,6\7,8,9)
A
        1   2   3
    +-------------+
  1 |  1   2   3  |
  2 |  4   5   6  |
  3 |  7   8   9  |
    +-------------+

b = diagonal(A)
b
        1
    +-----+
  1 |  1  |
  2 |  5  |
  3 |  9  |
    +-----+
```

`diag(b)`, creates a square matrix with the elements of vector b on its diagonal.

```
diag(b)
[symmetric]
        1   2   3
    +-------------+
  1 |  1          |
  2 |  0   5      |
  3 |  0   0   9  |
    +-------------+
```

4.4.5 Extracting the triangular matrix

`lowertriangle(A)` returns the lower triangle of A. `uppertriangleA)` returns the upper triangle of A:


```
lowertriangle(A)
        1   2   3
    +-------------+
  1 |  1   0   0  |
  2 |  4   5   0  |
  3 |  7   8   9  |
    +-------------+

uppertriangle(A)
        1   2   3
    +-------------+
  1 |  1   2   3  |
  2 |  0   5   6  |
  3 |  0   0   9  |
    +-------------+
```

4.4.6 Sorting a matrix

`sort(X, idx)` returns X with rows in ascending or descending order of the columns specified by idx. For instance, `sort(X, 1)` sorts X on its first column:

```
X = (2, 3, 1\ 2, 2, 2\ 1, 1, 3)
X
        1   2   3
    +-------------+
  1 |  2   3   1  |
  2 |  2   2   2  |
  3 |  1   1   3  |
    +-------------+

sort(X,1)
        1   2   3
    +-------------+
  1 |  1   1   3  |
  2 |  2   3   1  |
  3 |  2   2   2  |
    +-------------+
```

`sort(X, (1,2))` sorts X on its first and second columns (meaning rows with equal values in their first column are ordered on their second col-

umn):

```
sort(X,(1,2))
       1   2   3
    +-------------+
  1 |  1   1   3  |
  2 |  2   2   2  |
  3 |  2   3   1  |
    +-------------+
```

See `help mata sort()` for functions to randomly permutate matrices.

4.5 Basic matrix operators

Mata provides the following basic operators matrix operations:

- `+` Addition (of matrix or scalar)
- `-` Subtraction (of matrix or scalar)
- `*` Multiplication (with matrix or scalar)
- `/` Divison (by scalar)
- `^` Power (of scalar)

E.g., consider the vectors a and b and the scalar c:

```
a = (1..5)
b = (6::10)
c = 3
```

The vector d and the matrix F are given by:

```
d=a+c*a
 d
       1    2    3    4    5
    +--------------------------+
  1 |   4    8   12   16   20  |
    +--------------------------+
```



```
F=b*a
F
       1    2    3    4    5
    +--------------------------+
  1 |   6   12   18   24   30  |
  2 |   7   14   21   28   35  |
  3 |   8   16   24   32   40  |
  4 |   9   18   27   36   45  |
  5 |  10   20   30   40   50  |
    +--------------------------+
```

If the matrices involved in the operation are of incompatible sizes, Mata will return an error message:

```
F*d
                      *:  3200  conformability error
                <istmt>:     -  function returned error
r(3200);
```

4.6 Matrix multiplication

The mata function

```
cross(X, Z)
```

is an alternative way to calculate $X'Z$. cross() has the following advantages over the standard matrix-notation approach:

- `cross()` omits the rows in X and Z that contain missing values, which amounts to dropping observations with missing values.
- `cross()` uses less memory, especially when used with views.
- `cross()` makes calculations in special cases more efficiently. For instance, if you code cross(X, X), calculation is made for a symmetric matrix result.

In the four-argument version, `cross(X, xc, Z, zc)`, X is augmented on the right with a column of ones if xc is different from 0 and Z is similarly augmented if zc is different from 0. For example,

```
cross(X, 1, Z, 0)
```

adds a constant vector to X and nothing to Z.

`cross(X, w, Z)` returns $X'diag(w)Z$. `cross(X, xc, w, Z, zc)` is augmented as in the four-argument case. `cross(X, 0, 1, Z, 0)` is equivalent to `cross(X,Z)`.

4.6.1 Element-by-element operators

To indicate an element-by-element operation, precede a standard operator with a colon ":". Mata colon operators are:

`:*`	Multiplication (with matrix of same dimensions)
`:/`	Division (by matrix of same dimensions)
`:^`	Power (of matrix of same dimensions)

For example,

```
x = (1, 2, 3)
y = (4, 5, 6)
x:*y
        1    2    3
    +----------------+
  1 |   4   10   18  |
    +----------------+
```

Note: Element-by-element addition and subtraction is equal to the matrix operation.

4.6.2 Relational and logical operators

Mata equivalence operators

`==`	Equal to
`!=`	Not equal to

can be used with scalars, vectors and matrices and returns always a *scalar*. `==` returns false if the variables have different dimensions. For example,

```
G = (1, 2 \ 3, 4)
H = (1, 5 \ 6, 4)
G == T
  0
G[1,2] == 2
  1
G == 1
  0
G != 1
  1
```

Mata relational operators

`<, >`	Less, greater than
`<=, >=`	Less, greater than or equal to

can only be used with variables of identical dimensions and returns always a *scalar*. It evaluates the relation for all elements and returns true if it holds for all elements. For example

```
G < H
  0
G <= H
  1
```

Mata colon relational operators

`:==`	Equal to
`:!=`	Not equal to
`:<, :>`	Less, greater than
`:<=, :>=`	Less, greater than or equal to

can be used with scalars, vectors and matrices and returns the dimension of the highest dimension variable. For example,

```
G :== H
[symmetric]
       1   2
    +---------+
  1 |  1      |
  2 |  0   1  |
    +---------+
G :<= 2
       1   2
    +---------+
  1 |  1   1  |
  2 |  0   0  |
    +---------+
G :== (1\4)
[symmetric]
       1   2
    +---------+
  1 |  1      |
  2 |  0   1  |
    +---------+
```

Logical "and" and "or"

`&, &&`	logical and
`|,||`	logical or

can only be used with scalars. For example,

```
a = 5
a > 0 & a <= 10
  1
G <= H & a > 0
  1
```

Colon logical "and" and "or"

`:&`	logical and
`:|`	logical or

can be used with scalars, vectors and matrices. For example,


```
H :>= 0 :& H :<= 4
[symmetric]
       1   2
    +---------+
  1 |  1      |
  2 |  0   1  |
    +---------+
```

4.7 Building special matrices

4.7.1 Creating a matrix of constants

The mata function `J()` returns a matrix of constant. For example,

```
J(2, 3, 0)
       1   2   3
    +-------------+
  1 |  0   0   0  |
  2 |  0   0   0  |
    +-------------+

J(2, 3, 1)
       1   2   3
    +-------------+
  1 |  1   1   1  |
  2 |  1   1   1  |
    +-------------+
```

4.7.2 Creating a identity matrix

The mata function `I()` returns a matrix of constant. For example,

```
I(3)
[symmetric]
       1   2   3
    +-------------+
  1 |  1          |
  2 |  0   1      |
  3 |  0   0   1  |
    +-------------+
```

4.7.3 Creating a unit vector

The function `e(i, n)` returns a vector with all n elements equal to zero except for the ith, which is set to one.

```
e(2, 5)
       1   2   3   4   5
    +---------------------+
  1 |  0   1   0   0   0  |
    +---------------------+
```

4.7.4 Creating a random matrix

The `uniform()` function generates a matrix of random numbers whose elements are uniformly distributed in the interval (0,1). For example,

```
uniform(2, 3)
                1            2            3
    +-----------------------------------------+
  1 |  .1369840784    .643220668   .5578016951  |
  2 |  .6047949435    .684175977   .1086679425  |
    +-----------------------------------------+
```

`uniformseed(newseed)` sets the seed: a string previously obtained from uniformseed() may be specified for the argument, or an integer number may be specified. uniformseed() has the same effect as Stata's set seed command.

Mata has no built-in function to draw from the normal distribution. Independent draws from standard normal can be generated by applying the inverse cumulative normal distribution to uniform draws. For example,

```
invnormal(uniform(2,3))
                 1             2             3
    +---------------------------------------------+
  1 |   .3014337824   -1.545904789    .1389086436  |
  2 |   1.133267712   -.6583710099   -1.700496348  |
    +---------------------------------------------+
```


4.8 Inverse and linear equations system

Mata offers several functions to calculate the inverse of a matrix:

`luinv(A)`	inverse of full rank, square matrix A
`cholinv(A)`	inverse of positive definite, symmetric matrix A
`invsym(A)`	generalized inverse of positive-definite, symmetric matrix A

Mata offers several functions to solve the linear equation system $AX = B$:

`lusolve(A,B)`	A is full rank, square matrix
`cholinv(A)`	A is positive definite, symmetric matrix

Note: it is always numerically more accurate to solve $AX = B$ directly than to calculate $X = A^{-1}B$. Type `help m4 solvers` for more functions.

An example: estimate a linear regression using the dataset auto.dta:

```
webuse auto.dta
mata
y = st_data(.,"price")
X = st_data(.,("mpg", "weight"))
X = X, J(rows(X),1,1)
b = invsym(X'*X)*X'*y
```

There is a better version that saves memory, addresses missing values, elegantly adds a constant and efficiently solves for the parameter vector

```
M = X = y = .
st_view(M, ., ("price", "mpg", "weight"), 0)
st_subview(y, M, ., 1)
st_subview(X, M, ., (2\.))
XX = cross(X,1,X,1)
Xy = cross(X,1,y,0)
b = cholsolve(XX,Xy)
```

5 Controlling the flow

5.1 Loops

The while-loop

```
while (expr) {
   stmt
}
```

executes *stmt* zero or more times as long as *expr* is not equal to zero. For example

```
n = 5
i = 1
while (i<=n) {
   printf("i=%g\n", i)
   i++
}
printf("done\n")
```

The for-loop

```
for (expr1; expr2; expr3) {
   stmts
}
```

is equivalent to

```
expr1
while (expr2) {
   stmt
   expr3
}
```

For example,

```
n = 5
for (i=1; i<=n; i++) {
   printf("i=%g\n", i)
}
printf("done\n")
```


The do-loop

```
do {
   stmt
} while (exp)
```

executes *stmt* one or more times, until *expr* is zero. For example

```
n = 5
i = 1
do {
   printf("i=%g\n", i)
    i++
} while (i<=n)
printf("done\n")
```

`break` exits the innermost for, while, or do loop.

`continue` restarts the innermost for, while, or do loop.

5.2 Conditional statements

The if condition

```
if (expr) {
   stmt
}
```

evaluates *expr*, and if it is true (evaluates to a nonzero number), *stmt* is executed. Several conditions can be nested by *else if*; *else* introduces a statement that is executed when all conditions are false:

```
if (expr1) {
   stmt1
}
else if (expr2) {
   stmt2
}
else {
   stmt3
}
```

For example,

```
n = 5
i = 1
while (1) {
   printf("i=%g\n", i)
   i++
   if (i>n) {
      break
   }
}
printf("i=%g\n", i)
printf("done\n")
```

6 Coding Mata functions

6.1 Defining functions

Mata allows you to create new function. For example,

```
function zeros(c)
{
   a = J(c, 1, 0)
   return(a)
}
```

returns a cx1 column vector of zeros. You can call the function as any Mata function:

```
b = zeros(3)
b
       1
    +-----+
  1 |  0  |
  2 |  0  |
  3 |  0  |
    +-----+
```


6.2 Declarations

Variables (scalar, vectors and matrices) need not be declared in Mata functions. However, careful programmers use declarations in Mata. See `help m2 declarations` for a discussion.

In the above example we would add declarations at three places: in front of function definitions, inside the parentheses defining the function's arguments, and at the top of the body of the function, defining private variables the function will use:

```
real colvector zeros(real scalar c)
{
   real colvector a
   a = J(c,1,0)
   return(a)
}
```

In this case we tell Mata what input to expect (a scalar) and, if someone attempts to use our function incorrectly, Mata will stop execution.

Declarations consist of an element type (`transmorphic, numeric, real, complex, string, pointer`) and an organizational type (`matrix, vector, rowvector, colvector, scalar`).

By default, declarations are optional in Mata. However, you can ask in the Stata command prompt (not Mata) to be strict on declarations

```
set matastrict on
```

6.3 Passing arguments

Functions may require several arguments separated by a ",". Suppose we extend the above function `zeros()` to create matrices of zeros:

```
real matrix zeros(real scalar c, real scalar r)
{
   real matrix A
```

```
   A = J(c, r, 0)
   return(A)
}
```

Functions may be coded to allow receiving a variable number of arguments. This is done by placing a vertical or bar (|) in front of the first argument that is optional. For instance, we could allow the function `zeros()` to return a matrix if both arguments are specified and a column vector if only the first one is specified:

```
real matrix zeros(real scalar c,| real scalar r)
{
   real matrix A

   if (args()==1) r = 1
   A = J(c, r, 0)
   return(A)
}
```

The function `args()` is used to determine the number of arguments.

Important to note: Mata passes arguments to functions by address not by value. When you code

```
f(n)
```

it is the address of n that is passed to `f()`, not a copy of the values in n. `f(n)` can therefore modify n.

See `help m2 syntax` for more details.

6.4 Returning values

`return(expr)` causes the function to stop execution and return to the caller, returning the evaluation of `expr`. `return(expr)` may appear multiple times in the program. For example,

```
real matrix zeros(real scalar c,| real scalar r)
{
   if (args()==1) {
      return(J(c, 1, 0))
   }
   else {
      return(J(c, r, 0))
   }
}
```


A function is said to be void if it returns nothing. For example,

```
void zeros(real matrix A, real scalar c,| real scalar r)
{
   if (args()==1) {
      A = J(c, 1, 0)
   }
   else {
      A = J(c, r, 0)
   }
}
```

returns nothing but changes the value of the matrix A which is passed as first argument:

```
A = .
zeros(A, 2, 3)
A
       1   2   3
    +-------------+
  1 |  0   0   0  |
  2 |  0   0   0  |
    +-------------+
```

6.5 .mo and .mlib files

Once defined, new Mata functions can be stored as object code (i.e. in compiled form) in .mo files using the `mata mosave` command. The function can then be accessed in subsequent Mata sessions without being defined again.

For example, running the following .do file:

```
version 10
mata:

real matrix zeros(real scalar c,| real scalar r)
{
   real matrix A

   if (args()==1) r = 1
   A = J(c, r, 0)
   return(A)
}

mata mosave zeros(), replace

end
```

will produce the file zeros.mo which is stored in the working directory. If you close Stata and run it again, the function `zeros()` will be readily available.

Several Mata functions can be simultaneously stored in a .mlib file. See `help mata_mlib` for more details.

7 Coding Stata commands using Mata

7.1 An example

The following Stata and Mata commands are stored in the file ols.ado:

```
program define ols, eclass
   version 10.0
   syntax varlist(numeric) [if] [in]
   gettoken depvar indepvar: varlist
   marksample touse

   mata: m_ols("`varlist'", "`touse'")

   tempname b V
   matrix `b' = r(b)'
   matrix `V' = r(V)
   local N = r(N)
   matname `b' `indepvar' _cons, c(.)
   matname `V' `indepvar' _cons
   ereturn post `b' `V', depname(`depvar') obs(`N') esample(`touse')
   ereturn local cmd = "ols"
   ereturn display

end

capture mata mata drop m_ols()
version 10
mata:
void m_ols(string scalar varlist, string scalar touse)
{
   real matrix       M, X, V
   real colvector    y, b
   real scalar       n, k, s2

   M = X = y = .
   st_view(M, ., tokens(varlist), touse)
   st_subview(y, M, ., 1)
   st_subview(X, M, ., (2\.))
   n = rows(X)
   k = cols(X)
   XX = cross(X,1,X,1)
   if (rank(XX) < k+1) {
      errprintf("near singular matrix\n")
      exit(499)
   }
   Xy = cross(X,1,y,0)
   b = cholsolve(XX,Xy)
   e = y - (X, J(n,1,1))*b
   s2 = (e'e)/(n-k)
   V = s2*cholinv(XX)

   st_eclear()
   st_matrix("r(b)", b)
   st_matrix("r(V)", V)
   st_numscalar("r(N)", n)
}
end
```