The Stata Guide's Mata Cheat Sheet

Getting started

Mata is the matrix language of Stata and extremely powerful when it comes to programming estimation commands. Its much more faster than Stata but also requires much more precision since it is a lower level language than Stata. One can also interface interactively between Stata and Mata to move commands in and out. In order to get started, see `help mata` for extensive documentation and commands. Mata variables are either: scalars (1x1), vectors (1xn or nx1), matrices (n x m), or functions and expressions.

Everything generated in Mata stays in Mata. You can move in and out of Mata but the information is preserved as long as Stata is open. See `help m1_first` for more information.

How to use Mata

Mata can be invoked in several ways. One can either use single line instance (rare) or write a Mata code block (common).

```
mata or mata:
   <mata commands>
end
```

Start the Mata code block. The use of colon matters. If a colon is specified, then the program stops if an error occurs and the Mata instance is closed. If a colon is not specified, then the Mata moves to the next line and continues till the whole instance is completed. For debugging and finalizing programs, the use of colon is recommended.

```
mata: <mata command>
mata  <mata command>
```

Mata used in a single line. The use of single line commands is rare. This option is mostly used while programming in Mata especially if some variables need to be checked.

`mata describe`	Describe all variables.
`mata clear`	Clear everything.
`mata rename <name>`	Rename a variable.
`mata drop <names>`	Drop variables.
`mata set`	Set a bunch of Mata parameters. See `help mata_set`.
`mata stata`	Execute a Stata command in Mata.
`//`	Use two forward slashes to comment-out code in Mata. Unlike Stata asterisks (*) don't work in Mata.

Defining matrices

Mata allows one to define matrices in several ways. The most common way is to export variables from Stata into Mata (see first line below). Otherwise Mata allows variables to defined in several different ways. In order to try out the code below, Mata needs to be initialized. For example the second line can written as: `mata (4,3\2,1)` or `mata A = (4,3\2,1)` followed by `mata A` to display A. These can also be executed in a Mata code block. Below are some of the common ways of defining matrices in Mata.

`X = st_data(.,("var1", "var2"))`		Import all data points from Stata variables "var1" and "var2" in Mata and label the matrix X. Also see `help mf_st_view` for `st_view` a parallel command to `st_data`.
`(4,3\2,1)`	$\begin{pmatrix}4 & 3\\2 & 1\end{pmatrix}$	Define a 2x2 matrix
`(9,8,7)\(6,5,4)\(3,2,1)`	$\begin{pmatrix}9 & 8 & 7\\6 & 5 & 4\\3 & 2 & 1\end{pmatrix}$	Define a 3x3 matrix. Brackets are not necessary when defining matrices. They are mostly for code formatting and making it more legible.
`("a","b")\("c","d")`	$\begin{pmatrix}a & b\\c & d\end{pmatrix}$	Text elements of matrices have to be enclosed in double quotes
`(6,5,4)`	$\begin{pmatrix}6 & 5 & 4\end{pmatrix}$	Row vector
`(6\5\4)`	$\begin{pmatrix}6\\5\\4\end{pmatrix}$	Column vector
`(1..4)`	$\begin{pmatrix}1 & 2 & 3 & 4\end{pmatrix}$	Sequential row vector
`(1::3)`	$\begin{pmatrix}1\\2\\3\end{pmatrix}$	Sequential column vector
`runiform(3,2)`	$\begin{pmatrix}0.07 & 0.32\\0.55 & 0.88\\0.20 & 0.79\end{pmatrix}$	A 2x3 matrix drawn from a random (0,1) uniform distribution. Other available distributions are: beta, binomial, cauchy, chi2, discrete, exponential, gamma, hypergeometric, gaussian, laplace, logistic, nbinomial, normal, poisson, t, weibull, wibullph. See `help mf_runiform` for details.
`normaldens(0,1)`	0.3989	Scalar drawn from a standard normal density function. Other are: beta, binomial, cauchy, chi-sq, Dunnett, exponential, F, gamma, hypergeometric, Gaussian, invgaussian, laplace, logistic, nbinomial, poisson, t, Tukey, Weibull, Wishart. See help density functions for a full list.
`normaldens(runiform(3,3),0,1)`	$\begin{pmatrix}0.34 & 0.37 & 0.28\\0.37 & 0.40 & 0.30\\0.31 & 0.32 & 0.6\end{pmatrix}$	Normal density function drawn from a uniform matrix. For Mata distribution functions, see `help mf_normal`.
`J(3,2,7)`	$\begin{pmatrix}7 & 7\\7 & 7\\7 & 7\end{pmatrix}$	J is a matrix of constants. Arguments are rows, cols, value of the constant.
`J(2,2,1)`	$\begin{pmatrix}1 & \\1 & 1\end{pmatrix}$	A matrix of ones. If rows = cols, then the matrix is symmetric and only the lower triangle is shown.
`I(3)`	$\begin{pmatrix}1 & & \\0 & 1 & \\0 & 0 & 1\end{pmatrix}$	Identity matrix `I(n)` of dimension `n` which is symmetric by definition.
`e(3,5)`	$\begin{pmatrix}0 & 0 & 1 & 0 & 0\end{pmatrix}$	A unit vector. All elements are zeros except the 3rd element which is 1.
`range(0,6,2)`	$\begin{pmatrix}0\\2\\4\\6\end{pmatrix}$	Range vector, start with 0 end with 6 in steps of 2.

Operators

If :<operator> is specified, it implies element-wise operations. For example `a * b` is matrix multiplication, while `a :* b` is element-wise multiplication. See `help m2_op_colon`. Below are some of the common operators.

- `a = b` — a is b
- `a == b, a :== b` — a equals b, Same logic for !=
- `a >= b, a :>= b` — a is greater or equal to b. Same logic for >, <, <=
- `a & b, a :& b` — a and b
- `a | b, a :| b` — a or b
- `a + b, a :+ b` — a plus b. Same logic for subtract (-), multiple (*), divide (/)
- `a ^ b, a :^ b` — a to the power b
- `a'` — Transpose of a
- `*a, &a` — Mata points referring to contents of a, address of a. Extremely advanced programming options which are mentioned here for the sake of completeness. In case you are curious see `help m2_pointers`.
- `!a` — Not a
- `a++, a--` — Increment, decrement in steps of 1
- `a[x::y]` — Column x to y of a
- `a[x..y]` — Row x to y of a
- `a \ b` — Column join (rows must conform)
- `a , b` — Row join (columns must conform)
- `a[<subscripts>]` — Elements of a. Can be one element (a scalar) or a range of elements or a sub-matrix. E.g. `a[2,4]` which is element in row 2 and column 4 or `a[(1,3,4),(2,3)]` which is rows 1,3,4 and columns 2,3.
- `a[|<start>\<end>|]` — Another way of recovering a sub-matrix. Starting coordinates and ending coordinates. E.g. `a[|1,2 \ 4,5|]` which means creating a submatrix of elements ranging from `a[1,2]` to `a[4,5]`.

Matrix elements

This section showcases how different elements of matrices can be accessed using the Mata syntax shown in the "Defining matrices" and "Operators" sections above.

- `A=runiform(3,3)` — $\begin{pmatrix}0.73 & 0.05 & 0.75\\0.72 & 0.86 & 0.13\\0.87 & 0.77 & 0.25\end{pmatrix}$ — Define a matrix whose elements we want to access.
- `A[1,3]` — 0.75 — Row 1 and column 3 of A.
- `A[3,2]` — 0.77 — Row 3 and column 2 of A.
- `A[3,.]` — $\begin{pmatrix}0.87 & 0.77 & 0.25\end{pmatrix}$ — 3rd row of A.
- `A[.,3]` — $\begin{pmatrix}0.75\\0.25\\0.17\end{pmatrix}$ — 3rd column of A.
- `A[|2,1\3,3|]` / `A[(2::3),(1..3)]` — $\begin{pmatrix}0.72 & 0.86 & 0.13\\0.87 & 0.77 & 0.25\end{pmatrix}$ — Submatrix from row 2, col 1 to row 3, col 3. Submatrix from rows 2 to 3, and cols 1 to 3.
- `diagonal(A)` — $\begin{pmatrix}0.73\\0.86\\0.25\end{pmatrix}$ — Diagonal elements of A. Note here that `diagonal` is a n x 1 vector, while `diag` is a n x n matrix.

Stata – Mata interaction

Stata and Mata can talk to each other. Here is a simple regression code where data is imported in Mata from Stata, and the estimates are exported to Stata matrices from Mata. For a complete overview see `help m4_stata.`

```
clear all                                                   Load the auto dataset.
sysuse auto, clear

mata                                                        Start the Mata instance.
  y = st_data(.,"price")                                    Import the dependent variable.
  X = st_data(.,("mpg", "weight"))                          Import the independent variables.
  X = X, J(rows(X),1,1)                                     Add the vector of ones as the intercept at the end of matrix X.

  beta   = invsym(cross(X,X))*cross(X,y)                    Calculate beta.
  esq    = (y - X*beta) :^ 2                                Calculate square of the error terms.
  V      = (sum(esq)/(rows(X)-cols(X)))*invsym(cross(X,X))  Calculate the variance-covariance matrix.
  stderr = sqrt(diagonal(V))                                Calculate the standard error.

  st_matrix("b", beta)                                      Export the beta matrix to Stata.
  st_matrix("se", stderr)                                   Export the standard error matrix Stata.
end

mata: beta                                                  Display the Mata beta vector.
mata: stderr                                                Display the Mata standard error vector.
mat li b                                                    Display the Stata beta vector exported from Mata.
mat li se                                                   Display the Stata standard error vector exported from Mata.
regress price mpg weight                                    Compare the results with the standard Stata regression command.
```

Scalar functions

Scalar functions modify individual elements of matrices. For example `sqrt(A)` with give the square root of each element of matrix A. Therefore scalar functions are not dependent on matrix dimensions. Below are some of the common functions. Let's define a matrix `mata A = (-2,1\0,-5)`. For a detailed list please see `help m4_scalar`.

`abs(A)`	$\begin{pmatrix}2 & 1\\0 & 5\end{pmatrix}$	Absolute values of elements. All negative values are converted into positives.
`sign(A)`	$\begin{pmatrix}-1 & 1\\0 & -1\end{pmatrix}$	Returns an indicator for the sign of a scalar. Negative values = -1, zero = 0, positive values = 1
`exp(A)`	$\begin{pmatrix}0.135 & 2.718\\1 & 0.006\end{pmatrix}$	Returns the exponential of each element. Other options are: `log, log10, log1p, log1m, expm1, ln, ln1p, ln1m`.
`sqrt(A)`	$\begin{pmatrix}. & 1\\0 & .\end{pmatrix}$	Returns the square root of each element. Negative values are returned as missing.
`sin(A)`	$\begin{pmatrix}-0.909 & 0.841\\0 & 0.958\end{pmatrix}$	Returns the sin of each element. Other options are: `sin, cos, tan, asin, acos, atan, arg, atan2, sinh, cosh, tanh, asinh, acosh, atanh, pi`.

Matrix functions

Unlike scalar functions which are agnostic of matrix dimensions, most of the matrix functions require matrices to have some properties. For example, they should either be square matrices, or full rank for some operations to take place. Matrix functions also make use of "solvers" which are numerical solutions to matrix operations and are faster than regular operations. For example `cross(X,X)` is faster than `X'X`. On a similar note `quadcross(X,X)` is more precise than `cross(X,X)`. Below only select matrix functions are shown. For a complete overview see `help m4_matrix.`

`A = (5,4\6,7)` `B = (1,6\3,2)`	$\begin{pmatrix}5 & 4\\6 & 7\end{pmatrix}$ and $\begin{pmatrix}1 & 6\\3 & 2\end{pmatrix}$	Here we define two 2x2 matrices for convenience
`rows(A)` `cols(A)`	2 2	Number of rows and columns of A. These are widely used options for extracting the dimension of a matrix for looping or for generating new variables.
`rowsum(A)` `colsum(B)`	$\begin{pmatrix}9\\13\end{pmatrix}$ $\begin{pmatrix}4 & 8\end{pmatrix}$	Row and column sums are part of mathematical operations of matrices. There are a host of other functions available including: `sum, rowmin, colmin, rowmax, colmax, minmax, runningsum` etc. These functions are frequently used and for a complete overview and for advanced functions please see `help m4_mathematical`.
`mean(B)`	$\begin{pmatrix}2 & 4\end{pmatrix}$	Returns a row vector of column means. See also variance, meanvariance. Unlike mathematical function, there is now row mean option. It can be recovered using a double transpose: `mean(B')'.`
`selectindex((1,0,2,3,0))`	$\begin{pmatrix}1 & 2 & 3\end{pmatrix}$	Select all non zero columns in a vector. Can also be extended to column vectors. Cannot be used with matrices.
`select(B, B[.,1]:>2)`	$\begin{pmatrix}3 & 2\end{pmatrix}$	Select rows of B which are greater than 2. Select is a very powerful tool for partitioning matrices based on conditions. It is highly recommended to look at various select options. See `help mf_select`.
`sort(B,2)`	$\begin{pmatrix}3 & 2\\1 & 6\end{pmatrix}$	Sort B on column 2 in ascending order. See also jumble for randomizing rows, and order for recovering a vector that provides the sort order of a matrix. See help `mf_sort`.
`det(A)`	11	Determinant of A
`invsym(A)`	$\begin{pmatrix}0.37 & \\-0.21 & 0.26\end{pmatrix}$	Real, inverse, symmetric of A. Input matrix needs to be a square matrix. Invsym is a solver and therefore faster than manual inversion of matrices.
`cross(A,B)`	$\begin{pmatrix}23 & 42\\25 & 38\end{pmatrix}$	Cross product of A and B, which is simply A'B or A transpose times B. Cross is a solver and therefore faster than manual operation of matrix multiplications especially if the matrices are large. See also `help mf_crossdev`.
`trace(A)`	12	Sum of diagonals of a square matrix.
`rank(A)`	2	Rank of A is the number of columns that are linearly independent.
`norm(A)`	11.18	Norm of a matrix. This has a fairly complex formula depending on the matrix dimensions. Parts of the formula make use of rowsum, colsum, conj, trace, svdsv matrix functions.
`cholesky((1,2\2,13))`	$\begin{pmatrix}1 & 0\\2 & 3\end{pmatrix}$	Cholesky decomposition of a real symmetric matrix X. The returned matrix Y has the property that `YY'=X`. Not all symmetric matrices can have a Cholesky decomposition.
`lusolve(A,B)`	$\begin{pmatrix}-0.45 & 3.09\\0.82 & -2.36\end{pmatrix}$	Solve the system Ax=B for x. This solver uses the LU decomposition method . See help mf_lusolve for further options. A parallel option is qrsolve() (`help mf_qrsolve`) which uses the QR decomposition method.
`eigensystem(A,X,L)`	$L = \begin{pmatrix}11 & 1\end{pmatrix}$ $X = \begin{pmatrix}-0.55 & -0.71\\-0.83 & -0.71\end{pmatrix}$	Returns the eigen system of matrix A. Here L are the eigen values and X are the eigen vectors corresponding eigen values. The matrices L and X have to be defined as empty matrices beforehand: `mata L = .` and `mata X = .` so any other set of names can be used. See `help mf_eigensystem` for further options.

While, for, if statements

Here learn how to loop over matrix indices. The while and for loops are very useful for iterating over individual elements of matrices and can also be used to "collect" and store information over some sequence of matrix operations.

```
mata
 x = 1
 X = 4
  while (x <= X) {
   printf("%g \n", x)
   x++
  }
end
```

While loops are can be used to iterate over matrices until some conditions are met. For example for the convergence of sequences. In this code, the code repeats until x=4 is satisfied. While this code will just print the values of x, the central block can be replaced with any Mata code sequence.

Here also note that x has to be incremented by 1 before the while sequence ends otherwise the loop will run indefinitely.

```
mata
 N = 4
  for (i=1; i<=N; i++) {
  printf("%g \n", i)
  }
end
```

The for loop can also by used to loop over items. Unlike the while loop which is searching for some conditions to be met, the for loop iterates over a starting value and an ending value in increments of 1. This is also a difference between Stata and Mata, where in Stata one can define the increments.

```
mata
 x = 3
  for (i=1; i<=5; i++) {
    if (x > i) {
      printf("%g\n", 0)
    }
    else {
      printf("%g\n", 1)
    }
  }
end
```

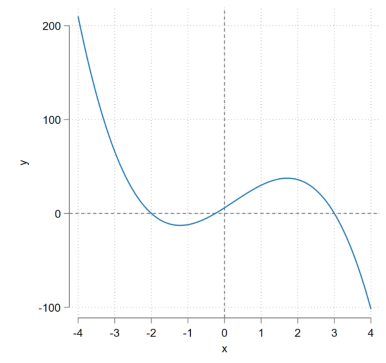
Within loops if/else conditions can also be nested. While the example here is fairly trivial, the aim is to how the code structure. Several other conditions can also be included using else if statements.

On a side note, if Mata has just two conditions, then this C++ like if/else condition can also be used: `(x > i ? 0 : 1)`

Which in the generic form equals `(a ? b : c)`, where `a ?` is a Boolean, `b` is the value if it's true and `c` if its false.

Optimization

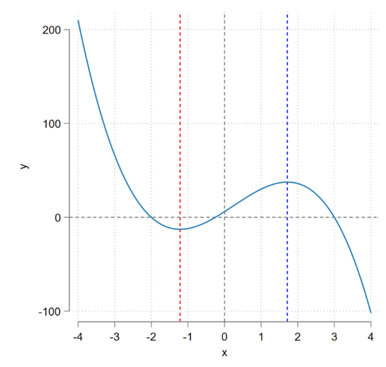
Optimize routines in Mata form the backbone of several regression functions and can be used to program and solve several complex non-linear optimization problems. For a complete overview see `help mf_optimize.` In the problem below we want to find the turning points of the function $-4x^3 + 3x^2 + 25x + 6$. This cubic function is shown below where we can see two turning points. One between (-2,-1) on the x-axis and the other between (1,2). In order to solve the problem an "evaluator function" has to be defined.




```
mata                                                Initialize Mata.
void myfunc(todo, x, y, g, H) {                     Define an evaluator function myfunc. Solve for default todo = d0, g =
  y = -4*x^3 + 3*x^2 + 25*x + 6                     gradient, H = Hessian. See help file for more details. Here y takes on the
  }                                                 value of the function we want to optimize.

  maxval = optimize_init()                          Initialize the maximum value optimize instance.
  optimize_init_which(maxval, "max")                Declare that we want to maximize (also the default option) .
  optimize_init_evaluator(maxval, &myfunc())        Initialize the evaluator on the function.
  optimize_init_params(maxval, 1)                   Since we can see the graph, we can start the search from 1.
  xmax = optimize(maxval)                           Recover the maximum value.

  minval = optimize_init()                          Same as above except we initialize to find the minimum point and start the
  optimize_init_which(minval, "min")                search from -1. For simple functions like these, starting points don't really
  optimize_init_evaluator(minval, &myfunc())        matter since convergence is achieved fast, but for more complex problems,
  optimize_init_params(minval, -1)                  these play an important role.
  xmin = optimize(minval)

    xmin, xmax                                      Print the values where (xmin, xmax) = (-1.215, 1.715).
       st_local("minx", strofreal(xmin))            Pass these values back to Stata as locals.
       st_local("maxx", strofreal(xmax))
end                                                 End Mata instance and move back to Stata.
```

Plot the function using twoway and use the minimum and maximum values stored from Mata to get the graph on the right.

```
twoway (function -4*x^3 + 3*x^2 + 25*x + 6, range(-4 4)), ///
yline(0) xline(0) xline(`minx', lc(red)) xline(`maxx', lc(blue)) ///
aspect(1) xsize(1) ysize(1)
```



Linear programming

Linear programming is arguably Mata's least known feature even though it forms the backbone of several Stata functions, like quantile regressions. Linear programming deals with optimizing a function subject to linear equality and inequality constraints and upper and lower bounds. For a complete overview see `help mf_linearprogram.`

```
mata
 func = (4, 5, 6)

 Leq = (-1, -1, 1)
 Req = 0

 Lineq = (-1, -1, 0 \ 1, -1, 0 \ -7, -12, 0)
 Rineq = (-11 \ 5 \ -35)

 Lbound = (0, 0, 0)
 Ubound = (., ., .)

  q = LinearProgram()
  q.setMaxOrMin("min")
  q.setCoefficients(func)
  q.setEquality(Leq, Req)
  q.setInequality(Lineq, Rineq)
  q.setBounds(Lbound, Ubound)

q.optimize()
q.parameters()

end
```

Here we minimize 4a + 5b + 6z
subject to z-x-y = 0, an equality constraint
and three inequality constraints:
x+y ≥11, x-y ≤5, 7x+12y ≥35

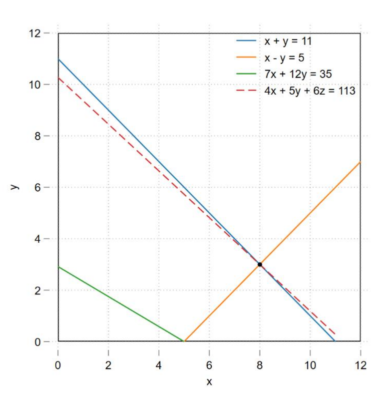
Since Mata as to take input of the form $Ax \leq B$, great or equal to constraints are switch to the negative domain by flipping the signs.

In the problem three bounds are also defined: $x \geq 0, y \geq 0, z \geq 0$. These represent the lower bounds while no upper bound exists.

Initialize the LinearProgram instance.
We explicitly state that we want to minimize, default is maximize.
Define the objective function.
Define the equality constraints which are in matrix form.
Define the inequality constraints which are in matrix form.
Define the lower and upper bounds..

Next step, the minimum value of the objective function = 113.
They have the value of $x = 8, y = 3, z = 11$.

The solution to the linear program is visualized in this figure here. See the guide for the graph code.



Notes & References

This poster represents a fraction of all the Mata options but the key ones are covered here. Broad topics of complex numbers, strings, and data and date/time formats, locals/globals, integration functions, programs, and many others are missing from this poster. They might be added in later versions subject to font size/space constraints.

In additional to the Stata help files, Stata forums, and various Stata Journal articles on Mata, the following books are highly recommended:

Baum, C. (2016). An Introduction to Stata Programming, Second Edition. *Stata Press.*
Gould, W. (2018). The Mata Book: A Book for Serious Programmers and Those Who Want to Be. *Stata Press.*

The Stata Guide (https://medium.com/the-stata-guide) by @AsjadNaqvi. This version 1.02 (26 June 2021). Fonts: Consolas for code, Futura for text.