

Title

[M-1] first — Introduction and first session

Description

Mata is a component of Stata. It is a matrix programming language which can be used interactively or as an extension for do-files and ado-files. Thus

1. Mata can be used by users who want to think in matrix terms and perform (not necessarily simple) matrix calculations interactively, and
2. Mata can be used by advanced Stata programmers who want to add features to Stata.

Mata has something for everybody.

Primary features of Mata are that it is fast and that it is C-like.

Remarks

This introduction is presented under the following headings:

Invoking Mata
Using Mata
Making mistakes: Interpreting error messages
Working with real numbers, complex numbers, and strings
Working with scalars, vectors, and matrices
Working with functions
Distinguishing real and complex values
Working with matrix and scalar functions
Performing element-by-element calculations: Colon operators
Writing programs
More functions
Mata environment commands
Exiting Mata

If you are reading the entries in the order suggested in [M-0] **intro**, see [M-1] **interactive** next.

Invoking Mata

To enter Mata, type `mata` at Stata's dot prompt and press enter; to exit Mata, type `end` at Mata's colon prompt:

```
. mata                               ← type mata to enter Mata
----- mata (type end to exit) -----
: 2 + 2                               ← type Mata statements at the colon prompt
  4
: end                                  ← type end to return to Stata
-----
. _                                    ← you are back to Stata
```

Using Mata

When you type a statement into Mata, Mata compiles what you typed and, if it compiled without error, executes it:

```
: 2 + 2
      4
: -
```

We typed $2 + 2$, a particular example from the general class of expressions. Mata responded with 4, the evaluation of the expression.

Often what you type are expressions, although you will probably choose more complicated examples. When an expression is not assigned to a variable, the result of the expression is displayed. Assignment is performed by the = operator:

```
: x = 2 + 2
: x
      4
: -
```

When we type “ $x = 2 + 2$ ”, the expression is evaluated and stored in the variable we just named x . The result is not displayed. We can look at the contents of x , however, simply by typing “ x ”. From Mata’s perspective, x is not only a variable but also an expression, albeit a rather simple one. Just as $2 + 2$ says to load 2, load another 2, and add them, the expression x says to load x and stop there.

As an aside, Mata distinguishes uppercase and lowercase. X is not the same as x :

```
: X = 2 + 3
: x
      4
: X
      5
: -
```

Making mistakes: Interpreting error messages

If you make a mistake, Mata complains, and then you continue on your way. For instance,

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

$2,,3$ makes no sense to Mata, so Mata complained. This is an example of what is called a compile-time error; Mata could not make sense out of what we typed.

The other kind of error is called a run-time error. For example, we have no variable called y . Let us ask Mata to show us the contents of y :

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

Here what we typed made perfect sense—show me y —but y has never been defined. This ugly message is called a run-time error message—see [M-2] **errors** for a complete description—but all that's important is to understand the difference between

```
invalid expression
```

and

```
<istmt>: 3499 y not found
```

The run-time message is prefixed by an identity (<istmt> here) and a number (3499 here). Mata is telling us, “I was executing your *istmt* [that's what everything you type is called] and I got error 3499, the details of which are that I was unable to find y .”

The compile-time error message is of a simpler form: `invalid expression`. When you get such unprefixed error messages, that means Mata could not understand what you typed. When you get the more complicated error message, that means Mata understood what you typed, but there was a problem performing your request.

Another way to tell the difference between compile-time errors and run-time errors is to look at the return code. Compile-time errors have a return code of 3000:

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

Run-time errors have a return code that might be in the 3000s, but is never 3000 exactly:

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

Whether the error is compile-time or run-time, once the error message is issued, Mata is ready to continue just as if the error never happened.

Working with real numbers, complex numbers, and strings

As we have seen, Mata works with real numbers:

```
: 2 + 3
5
```

Mata also understands complex numbers; you write the imaginary part by suffixing a lowercase i :

```
: 1+2i + 4-1i
5 + 1i
```

For imaginary numbers, you can omit the real part:

```
: 1+2i - 2i
1
```

Whether a number is real or complex, you can use the same computer notation for the imaginary part as you would for the real part:

```
: 2.5e+3i
2500i
: 1.25e+2+2.5e+3i          /* i.e., 1.25+e02 + 2.5e+03i */
125 + 2500i
```

We purposely wrote the last example in nearly unreadable form just to emphasize that Mata could interpret it.

Mata also understands strings, which you write enclosed in double quotes:

```
: "Alpha" + "Beta"
   AlphaBeta
```

Just like Stata, Mata understands simple and compound double quotes:

```
: "'Alpha'" + "'Beta'"
   AlphaBeta
```

You can add complex and reals,

```
: 1+2i + 3
   4+2i
```

but you may not add reals or complex to strings:

```
: 2 + "alpha"
                                     <istmt>: 3250 type mismatch
r(3250);
```

We got a run-time error. Mata understood `2 + "alpha"` all right; it just could not perform our request.

Working with scalars, vectors, and matrices

In addition to understanding scalars—be they real, complex, or string—Mata understands vectors and matrices of real, complex, and string elements:

```
: x = (1, 2)
: x
      1  2
1 | 1  2
```

`x` now contains the row vector (1, 2). We can add vectors:

```
: x + (3, 4)
      1  2
1 | 4  6
```

The “,” is the column-join operator; things like (1, 2) are expressions, just as (1 + 2) is an expression:

```
: y = (3, 4)
: z = (x, y)
: z
      1  2  3  4
1 | 1  2  3  4
```

In the above, we could have dispensed with the parentheses and typed “`y = 3, 4`” followed by “`z = x, y`”, just as we could using the `+` operator, although most people find vectors more readable when enclosed in parentheses.

\ is the row-join operator:

```
: a = (1 \ 2)
```

```
: a
```

```
      1
1  [ 1 ]
2  [ 2 ]
```

```
:
```

```
: b = (3 \ 4)
```

```
: c = (a \ b)
```

```
: c
```

```
      1
1  [ 1 ]
2  [ 2 ]
3  [ 3 ]
4  [ 4 ]
```

Using the column-join and row-join operators, we can enter matrices:

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

```
: A
```

```
      1  2
1  [ 1  2 ]
2  [ 3  4 ]
```

The use of these operators is not limited to scalars. Remember, x is the row vector $(1, 2)$, y is the row vector $(3, 4)$, a is the column vector $(1 \ 2)$, and b is the column vector $(3 \ 4)$. Therefore,

```
: x \ y
```

```
      1  2
```

```
1  [ 1  2 ]
2  [ 3  4 ]
```

```
: a, b
```

```
      1  2
```

```
1  [ 1  3 ]
2  [ 2  4 ]
```

But if we try something nonsensical, we get an error:

```
: a, x
```

```
<istmt>: 3200 conformability error
```

We create complex vectors and matrices just as we create real ones, the only difference being that their elements are complex:

```
: Z = (1 + 1i, 2 + 3i \ 3 - 2i, -1 - 1i)
```

```
: Z
```

```
      1      2
1  [ 1 + 1i  2 + 3i ]
2  [ 3 - 2i  -1 - 1i ]
```

Similarly, we can create string vectors and matrices, which are vectors and matrices with string elements:

```
: S = ("1st element", "2nd element" \ "another row", "last element")
: S
      1          2
1  1st element  2nd element
2  another row  last element
```

For strings, the individual elements can be up to 2,147,483,647 characters long.

Working with functions

Mata's expressions also include functions:

```
: sqrt(4)
2
: sqrt(-4)
.
```

When we ask for the square root of -4 , Mata replies `."` Further, `.` can be stored just like any other number:

```
: findout = sqrt(-4)
: findout
.
```

`."` means missing, that there is no answer to our calculation. Taking the square root of a negative number is not an error; it merely produces missing. To Mata, missing is a number like any other number, and the rules for all the operators have been generalized to understand missing. For instance, the addition rule is generalized such that missing plus anything is missing:

```
: 2 + .
.
```

Still, it should surprise you that Mata produced missing for the `sqrt(-4)`. We said that Mata understands complex numbers, so should not the answer be $2i$? The answer is that it should be if you are working on the complex plane, but otherwise, missing is probably a better answer. Mata attempts to intuit the kind of answer you want by context, and in particular, uses inheritance rules. If you ask for the square root of a real number, you get a real number back. If you ask for the square root of a complex number, you get a complex number back:

```
: sqrt(-4 + 0i)
2i
```

Here complex means multipart: $-4 + 0i$ is a complex number; it merely happens to have 0 imaginary part. Thus:

```
: areal = -4
: acomplex = -4 + 0i
: sqrt(areal)
.
: sqrt(acomplex)
2i
```

If you ever have a real scalar, vector, or matrix, and want to make it complex, use the `C()` function, which means “convert to complex”:

```
: sqrt(C(areal))
2i
```

`C()` is documented in [M-5] `C()`. `C()` allows one or two arguments. With one argument, it casts to complex. With two arguments, it makes a complex out of the two real arguments. Thus you could type

```
: sqrt(-4 + 2i)
.485868272 + 2.05817103i
```

or you could type

```
: sqrt(C(-4, 2))
.485868272 + 2.05817103i
```

By the way, used with one argument, `C()` also allows complex, and then it does nothing:

```
: sqrt(C(acomplex))
2i
```

Distinguishing real and complex values

It is virtually impossible to tell the difference between a real value and a complex value with zero imaginary part:

```
: areal = -4
: acomplex = -4 + 0i
: areal
-4
: acomplex
-4
```

Yet, as we have seen, the difference is important: `sqrt(areal)` is missing, `sqrt(acomplex)` is `2i`. One solution is the `eltype()` function:

```
: eltype(areal)
real
: eltype(acomplex)
complex
```

`eltype()` can also be used with strings,

```
: astring = "hello"
: eltype(astring)
string
```

but this is useful mostly in programming contexts.

Working with matrix and scalar functions

Some functions are matrix functions: they require a matrix and return a matrix. Mata's `invsym(X)` is an example of such a function. It returns the matrix that is the inverse of symmetric, real matrix X :

```
: X = (76, 53, 48 \ 53, 88, 46 \ 48, 46, 63)
: Xi = invsym(X)
: Xi
[symmetric]
      1          2          3
1  .0298458083
2  -.0098470272 .0216268926
3  -.0155497706 -.0082885675 .0337724301

: Xi * X
      1          2          3
1      1 -8.67362e-17 -8.50015e-17
2  -1.38778e-16      1 -1.02349e-16
3      0  1.11022e-16      1
```

The last matrix, $X_i * X$, differs just a little from the identity matrix because of unavoidable computational roundoff error.

Other functions are, mathematically speaking, scalar functions. `sqrt()` is an example in that it makes no sense to speak of `sqrt(X)`. (That is, it makes no sense to speak of `sqrt(X)` unless we were speaking of the Cholesky square-root decomposition. Mata has such a matrix function; see [M-5] `cholesky()`.)

When a function is, mathematically speaking, a scalar function, the corresponding Mata function will usually allow vector and matrix arguments and, then, the Mata function makes the calculation on each element individually:

```
: M = (1, 2 \ 3, 4 \ 5, 6)
: M
      1  2
1  1  2
2  3  4
3  5  6

:
: S = sqrt(M)
: S
      1          2
1      1  1.414213562
2  1.732050808      2
3  2.236067977  2.449489743

:
: S[1,2]*S[1,2]
2
: S[2,1]*S[2,1]
3
```

When a function returns a result calculated in this way, it is said to return an element-by-element result.

Performing element-by-element calculations: Colon operators

Mata's operators, such as + (addition) and * (multiplication), work as you would expect. In particular, * performs matrix multiplication:

```
: A = (1, 2 \ 3, 4)
: B = (5, 6 \ 7, 8)
: A*B
      1  2
1  19  22
2  43  50
```

The first element of the result was calculated as $1 * 5 + 2 * 7 = 19$.

Sometimes, you really want the element-by-element result. When you do, place a colon in front of the operator: Mata's :* operator performs element-by-element multiplication:

```
: A:*B
      1  2
1  5  12
2  21  32
```

See [M-2] **op_colon** for more information.

Writing programs

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

```
: function add(a,b) return(a+b)
```

That single statement creates a new function, although perhaps you would prefer if we typed it as

```
: function add(a,b)
> {
>     return(a+b)
> }
```

because that makes it obvious that a program can contain many lines. In either case, once defined, we can use the function:

```
: add(1,2)
3
: add(1+2i,4-1i)
5 + 1i
: add( (1,2), (3,4) )
      1  2
1  4  6
```

```

: add(x,y)
      1  2
1  

|   |   |
|---|---|
| 4 | 6 |
|---|---|



: add(A,A)
      1  2
1  

|   |   |
|---|---|
| 2 | 4 |
| 6 | 8 |



:
: Z1 = (1+1i, 1+1i \ 2, 2i)
: Z2 = (1+2i, -3+3i \ 6i, -2+2i)
: add(Z1, Z2)
      1      2
1  

|        |         |
|--------|---------|
| 2 + 3i | -2 + 4i |
| 2 + 6i | -2 + 4i |



:
: add("Alpha","Beta")
AlphaBeta

:
: S1 = ("one", "two" \ "three", "four")
: S2 = ("abc", "def" \ "ghi", "jkl")
: add(S1, S2)
      1      2
1  

|          |         |
|----------|---------|
| oneabc   | twodef  |
| threeghi | fourjkl |


```

Of course, our little function `add()` does not do anything that the `+` operator does not already do, but we could write a program that did do something different. The following program will allow us to make $n \times n$ identity matrices:

```

: real matrix id(real scalar n)
> {
>   real scalar i
>   real matrix res
>
>   res = J(n, n, 0)
>   for (i=1; i<=n; i++) {
>     res[i,i] = 1
>   }
>   return(res)
> }

:
: I3 = id(3)
: I3
[symmetric]
      1  2  3
1  

|   |   |   |
|---|---|---|
| 1 |   |   |
| 0 | 1 |   |
| 0 | 0 | 1 |


```

The function `J()` in the program line `res = J(n, n, 0)` is a Mata built-in function that returns an $n \times n$ matrix containing 0s (`J(r, c, val)` returns an $r \times c$ matrix, the elements of which are all equal to `val`); see [M-5] **J()**.

`for (i=1; i<=n; i++)` says that starting with `i=1` and so long as `i<=n` do what is inside the braces (set `res[i, i]` equal to 1) and then (we are back to the `for` part again), increment `i`.

The final line—`return(res)`—says to return the matrix we have just created.

Actually, just as with `add()`, we do not need `id()` because Mata has a built-in function `I(n)` that makes identity matrices, but it is interesting to see how the problem could be programmed.

More functions

Mata has many functions already and much of this manual concerns documenting what those functions do; see [M-4] **intro**. But right now, what is important is that many of the functions are themselves written in Mata!

One of those functions is `pi()`; it takes no arguments and returns the value of π . The code for it reads

```
real scalar pi() return(3.141592653589793238462643)
```

There is no reason to type the above function because it is already included as part of Mata:

```
: pi()
   3.141592654
```

When Mata lists a result, it does not show as many digits, but we could ask to see more:

```
: printf("%17.0g", pi())
   3.14159265358979
```

Other Mata functions include the hyperbolic functions `sinh(u)`, `cosh(u)`, etc. The code for `sinh(u)`, `cosh(u)`, and `tanh(u)` reads

```
numeric matrix sinh(numeric matrix u) return((exp(u)-exp(-u)):2)
numeric matrix cosh(numeric matrix u) return((exp(u)+exp(-u)):2)
numeric matrix tanh(numeric matrix u)
{
    numeric matrix eu, emu
    eu = exp(u)
    emu = exp(-u)
    return( (eu-emu):(eu+emu) )
}
```

See for yourself: at the Stata dot prompt (not the Mata colon prompt), type

```
. viewsource sinh.mata
. viewsource cosh.mata
. viewsource tanh.mata
```

When the code for a function was written in Mata (as opposed to having been written in C), `viewsource` can show you the code; see [M-1] **source**.

Returning to the functions themselves,

```
numeric matrix sinh(numeric matrix u) return((exp(u)-exp(-u))/2)
numeric matrix cosh(numeric matrix u) return((exp(u)+exp(-u))/2)
numeric matrix tanh(numeric matrix u)
{
    numeric matrix eu, emu
    eu = exp(u)
    emu = exp(-u)
    return( (eu-emu)/(eu+emu) )
}
```

this is the first time we have seen the word `numeric`: it means real or complex. Built-in (previously written) function `exp()` works like `sqrt()` in that it allows a real or complex argument and correspondingly returns a real or complex result. Said in Mata jargon: `exp()` allows a numeric argument and correspondingly returns a numeric result. `sinh()`, `cosh()`, and `tanh()` will also work like `sqrt()` and `exp()`.

Another characteristic `sinh()`, `cosh()`, and `tanh()` share with `sqrt()` and `exp()` is element-by-element operation. `sinh()`, `cosh()`, and `tanh()` are element-by-element because `exp()` is element-by-element and because we were careful to use the `:/` (element-by-element) divide operator.

In any case, there is no need to type the above functions because they are already part of Mata. You could learn more about them by seeing their manual entry, [M-5] `sin()`.

At the other extreme, Mata functions can become long. Here is Mata's function to solve $AX = B$ for X when A is lower triangular, placing the result X back into A :

```
real scalar _solvelower(
    numeric matrix A, numeric matrix b,
    |real scalar usertol, numeric scalar userd)
{
    real scalar          tol, rank, a_t, b_t, d_t
    real scalar          n, m, i, im1, complex_case
    numeric rowvector    sum
    numeric scalar       zero, d

    d = userd

    if ((n=rows(A))!=cols(A)) _error(3205)
    if (n != rows(b))         _error(3200)
    if (isview(b))            _error(3104)
    m = cols(b)
    rank = n

    a_t = iscomplex(A)
    b_t = iscomplex(b)
    d_t = d <. ? iscomplex(d) : 0

    complex_case = a_t | b_t | d_t
```

```

if (complex_case) {
  if (!a_t) A = C(A)
  if (!b_t) b = C(b)
  if (d<. & !d_t) d = C(d)
  zero = 0i
}
else zero = 0

if (n==0 | m==0) return(0)

tol = solve_tol(A, usertol)

if (abs(d) >=. ) {
  if (abs(d=A[1,1])<=tol) {
    b[1,.] = J(1, m, zero)
    --rank
  }
  else {
    b[1,.] = b[1,.] ./ d
    if (missing(d)) rank = .
  }

  for (i=2; i<=n; i++) {
    im1 = i - 1
    sum = A[|i,1|,im1|] * b[|1,1\im1,m|]
    if (abs(d=A[i,i])<=tol) {
      b[i,.] = J(1, m, zero)
      --rank
    }
    else {
      b[i,.] = (b[i,.]-sum) ./ d
      if (missing(d)) rank = .
    }
  }
}
else {
  if (abs(d)<=tol) {
    rank = 0
    b = J(rows(b), cols(b), zero)
  }
  else {
    b[1,.] = b[1,.] ./ d

    for (i=2; i<=n; i++) {
      im1 = i - 1
      sum = A[|i,1|,im1|] * b[|1,1\im1,m|]
      b[i,.] = (b[i,.]-sum) ./ d
    }
  }
}
return(rank)
}

```

If the function were not already part of Mata and you wanted to use it, you could type it into a do-file or onto the end of an ado-file (especially good if you just want to use `_solvelower()` as a subroutine). In those cases, do not forget to enter and exit Mata:

```

----- top of ado-file -----
program mycommand
  ...
  ado-file code appears here
  ...
end
mata:
  _solvelower() code appears here
end
----- end of ado-file -----

```

Sharp-eyed readers will notice that we put a colon on the end of the Mata command. That's a detail, and why we did that is explained in [M-3] **mata**.

In addition to loading functions by putting their code in do- and ado-files, you can also save the compiled versions of functions in `.mo` files (see [M-3] **mata mosave**) or into `.mlib` Mata libraries (see [M-3] **mata mlib**).

For `_solvelower()`, it has already been saved into a library, namely, Mata's official library, so you need not do any of this.

Mata environment commands

When you are using Mata, there is a set of commands that will tell you about and manipulate Mata's environment.

The most useful such command is `mata describe`:

```

: mata describe

```

# bytes	type	name and extent
76	transmorphic matrix	add()
200	real matrix	id()
32	real matrix	A[2,2]
32	real matrix	B[2,2]
72	real matrix	I3[3,3]
48	real matrix	M[3,2]
48	real matrix	S[3,2]
47	string matrix	S1[2,2]
44	string matrix	S2[2,2]
72	real matrix	X[3,3]
72	real matrix	Xi[3,3]
64	complex matrix	Z[2,2]
64	complex matrix	Z1[2,2]
64	complex matrix	Z2[2,2]
16	real colvector	a[2]
16	complex scalar	acomplex
8	real scalar	areal
16	real colvector	b[2]
32	real colvector	c[4]
8	real scalar	findout
16	real rowvector	x[2]
16	real rowvector	y[2]
32	real rowvector	z[4]

```

: _

```

Another useful command is `mata clear`, which will clear Mata without disturbing Stata:

```
: mata clear
: mata describe
      # bytes   type                               name and extent
```

There are other useful `mata` commands; see [M-3] **intro**. Do not confuse this command `mata`, which you type at Mata's colon prompt, with Stata's command `mata`, which you type at Stata's dot prompt and which invokes Mata.

Exiting Mata

When you are done using Mata, type `end` to Mata's colon prompt:

```
: end
```

```
. -
```

Exiting Mata does not clear it:

```
. mata
----- mata (type end to exit) -----
: x = 2
: y = (3 + 2i)
: function add(a,b) return(a+b)
: end
-----
. ...
. mata
----- mata (type end to exit) -----
: mata describe
      # bytes   type                               name and extent
-----
          38   transmorphic matrix             add()
           8   real scalar                       x
          16   complex scalar                   y
-----
: end
```

Exiting Stata clears Mata, as does Stata's `clear mata` command; see [D] **clear**.

Also See

[M-1] **intro** — Introduction and advice