Title

stata.com

How — How Mata works

Description Remarks and examples Reference Also see

Description

Below we take away some of the mystery and show you how Mata works. Everyone, we suspect, will find this entertaining, and advanced users will find the description useful for predicting what Mata will do when faced with unusual situations.

Remarks and examples

stata.com

Remarks are presented under the following headings:

If you are reading the entries in the order suggested in [M-0] **Intro**, browse [M-1] **Intro** next for sections that interest you, and then see [M-2] **Syntax**.

What happens when you define a program

Let's say you fire up Mata and type

```
: function tryit()
> {
>      real scalar i
>
>      for (i=1; i<=10; i++) i
> }
```

Mata compiles the program: it reads what you type and produces binary codes that tell Mata exactly what it is to do when the time comes to execute the program. In fact, given the above program, Mata produces the binary code

```
00b4 3608 4000 0000 0100 0000 2000 0000
0000 0000 ffff ffff 0300 0000 0000 0000
0100 7472 7969 7400 1700 0100 1f00 0700
0000 0800 0000 0200 0100 0800 2a00 0300
1e00 0300
```

which looks meaningless to you and me, but Mata knows exactly what to make of it. The compiled version of the program is called object code, and it is the object code, not the original source code, that Mata stores in memory. In fact, the original source is discarded once the object code has been stored.

It is this compilation step—the conversion of text into object code—that makes Mata able to execute programs so quickly.

Later, when the time comes to execute the program, Stata follows the instructions it has previously recorded:

```
: tryit()
  1
  2
  3
  4
  5
  6
  7
  8
  9
  10
```

What happens when you work interactively

Let's say you type

```
: x = 3
```

In the jargon of Mata, that is called an *istmt*—an interactive statement. Obviously, Mata stores 3 in `x`, but how?

Mata first compiles the single statement and stores the resulting object code under the name `<istmt>`. The result is much as if you had typed

```
: function <istmt>()
> {
>         x = 3
> }
```

except, of course, you could not define a program named `<istmt>` because the name is invalid. Mata has ways of getting around that.

At this point, Mata has discarded the source code `x = 3` and has stored the corresponding object code. Next, Mata executes `<istmt>`. The result is much as if you had typed

```
: <istmt>()
```

That done, there is only one thing left to do, which is to discard the object code. The result is much as if you typed

```
: mata drop <istmt>()
```

So there you have it: you type

```
: x = 3
```

and Mata executes

```
: function <istmt>()
> {
>         x = 3
> }
: <istmt>()
: mata drop <istmt>()
```

❑ Technical note

The above story is not exactly true because, as told, variable x would be local to function <istmt>() so, when <istmt>() concluded execution, variable x would be discarded. To prevent that from happening, Mata makes all variables defined by <istmt>() global. Thus you can type

```
: x = 3
```

followed by

```
: y = x + 2
```

and all works out just as you expect: y is set to 5.

❑

What happens when you type a mata environment command

When you are at a colon prompt and type something that begins with the word mata, such as

```
: mata describe
```

or

```
: mata clear
```

something completely different happens: Mata freezes itself and temporarily transfers control to a command interpreter like Stata's. The command interpreter accesses Mata's environment and reports on it or changes it. Once done, the interpreter returns to Mata, which comes back to life, and issues a new colon prompt:

```
: _
```

Once something is typed at the prompt, Mata will examine it to determine if it begins with the word mata (in which case the interpretive process repeats), or if it is the beginning of a function definition (in which case the program will be compiled but not executed), or anything else (in which case Mata will try to compile and execute it as an <istmt>()).

Working with object code I: .mo files

Everything hinges on the object code that Mata produces, and, if you wish, you can save the object code on disk. The advantage of doing this is that, at some future date, your program can be executed without compilation, which saves time. If you send the object code to others, they can use your program without ever seeing the source code behind it.

After you type, say,

```
: function tryit()
> {
>         real scalar i
>
>         for (i=1; i<=10; i++) i
> }
```

Mata has created the object code and discarded the source. If you now type

```
: mata mosave tryit()
```

the Mata interpreter will create file tryit.mo in the current directory; see [M-3] **mata mosave**. The new file will contain the object code.

At some future date, were you to type

```
: tryit()
```

without having first defined the program, Mata would look along the ado-path (see [P] **sysdir** and **[U] 17.5 Where does Stata look for ado-files?**) for a file named `tryit.mo`. Finding the file, Mata loads it (so Mata now has the object code and executes it in the usual way).

Working with object code II: .mlib libraries

Object code can be saved in `.mlib` libraries (files) instead of `.mo` files. `.mo` files contain the object code for one program. `.mlib` files contain the object code for a group of files.

The first step is to choose a name (we will choose `lmylib`—library names are required to start with the lowercase letter *l*) and create an empty library of that name:

```
: mata mlib create lmylib
```

Once created, new functions can be added to the library:

```
: mata mlib add lmylib tryit()
```

New functions can be added at any time, either immediately after creation or later—even much later; see [M-3] **mata mlib**.

We mentioned that when Mata needs to execute a function that it does not find in memory, Mata looks for a `.mo` file of the same name. Before Mata does that, however, Mata thumbs through its libraries to see if it can find the function there.

The Mata environment

Certain settings of Mata affect how it behaves. You can see those settings by typing `mata query` at the Mata prompt:

```
: mata query
-------------------------------------------------------------------------------

    Mata settings
      set matastrict       off
      set matalnum         off
      set mataoptimize     on
      set matafavor        space      may be space or speed
      set matacache        2000       kilobytes
      set matalibs         lmatabase;lmatapss;lmatapostest;lmataerm;lmatapath;
> lmatagsem;lmataopt;lmatasem;lmataado;lmatafc;lmatasp;lmatamcmc;lmatacollect;
> lmatamixlog
      set matamofirst      off
      set matasolvetol     .          may be . or any double-precision number

: _
```

You can change these settings by using `mata set`; see [M-3] **mata set**. We recommend the default settings, except that we admit to being partial to `mata set matastrict on`.

Reference

Gould, W. W. 2006. Mata Matters: Precision. *Stata Journal* 6: 550–560.

Also see

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