Title

stata.com

Ado — Using Mata with ado-files

Description Remarks and examples Also see

Description

This section provides advice to ado-file programmers on how to use Mata effectively and efficiently.

Remarks and examples

stata.com

For those interested in extending Stata by adding new commands, Mata is not a replacement for ado-files. Rather, the appropriate way to use Mata is to create subroutines used by your ado-files. Below we discuss how to do that under the following headings:

A first example

We will pretend that Stata cannot produce sums and that we want to write a new command for Stata that will report the sum of one variable. Here is our first take on how we might do this:

begin `varsum.ado`

```
program varsum
     version 17.0
     syntax varname [if] [in]
     marksample touse
     mata: calcsum("`varlist'", "`touse'")
     display as txt "  sum = " as res r(sum)
end

version 17.0
mata:
void calcsum(string scalar varname, string scalar touse)
{
     real colvector  x

     st_view(x, ., varname, touse)
     st_numscalar("r(sum)", colsum(x))
}
end
```

end `varsum.ado`

Running this program from Stata results in the following output:

```
. varsum mpg
  sum = 1576
```

Note the following:

1. The ado-file has both ado-code and Mata code in it.
2. The ado-code handled all issues of parsing and identifying the subsample of the data to be used.
3. The ado-code called a Mata function to perform the calculation.
4. The Mata function received as arguments the names of two variables in the Stata dataset: the variable on which the calculation was to be made and the variable that identified the subsample of the data to be used.
5. The Mata function returned the result in `r()`, from where the ado-code could access it.

The outline that we showed above is a good one, although we will show you alternatives that, for some problems, are better.

Where to store the Mata functions

Our ado-file included a Mata function. You have three choices of where to put the Mata function:

1. You can put the code for the Mata function in the ado-file, as we did. To work, your `.ado` file must be placed where Stata can find it.
2. You can omit code for the Mata function from the ado-file, compile the Mata function separately, and store the compiled code (the object code) in a separate file called a `.mo` file. You place both your `.ado` and `.mo` files where Stata can find them.
3. You can omit the code for the Mata function from the ado-file, compile the Mata function separately, and store the compiled code in a `.mlib` library. Here both your `.ado` file and your `.mlib` library will need to be placed where Stata can find them.

We will show you how to do each of these alternatives, but before we do, let's consider the advantages and disadvantages of each:

1. Putting your Mata source code right in the ado-file is easiest, and it sure is convenient. The disadvantage is that Mata must compile the source code into object code, and that will slow execution. The cost is small because it occurs infrequently; Mata compiles the code when the ado-file is loaded and, as long as the ado-file is not dropped from memory, Stata and Mata will use the same compiled code over and over again.
2. Saving your Mata code as `.mo` files saves Mata from having to compile them at all. The source code is compiled only once—at the time you create the `.mo` file—and thereafter, it is the already-compiled copy that Stata and Mata will use.

   There is a second advantage: rather than the Mata functions (`calcsum()` here) being private to the ado-file, you will be able to use `calcsum()` in your other ado-files. `calcsum()` will become a utility always available to you. Perhaps `calcsum()` is not deserving of this honor.
3. Saving your Mata code in a `.mlib` library has the same advantages and disadvantages as (2); the only difference is that we save the precompiled code in a different way. The `.mo`/`.mlib` choice is of more concern to those who intend to distribute their ado-file to others. `.mlib` libraries allow you to place many Mata functions (subroutines for your ado-files) into one file, and so it is easier to distribute.

Let's restructure our ado-file to save `calcsum()` in a `.mo` file. First, we simply remove `calcsum()` from our ado-file, so it now reads

begin `varsum.ado`

```
program varsum
     version 17.0
     syntax varname [if] [in]
     marksample touse
     mata: calcsum("`varlist'", "`touse'")
     display as txt "  sum = " as res r(sum)
end
```

end `varsum.ado`

Next, we enter Mata, enter our `calcsum()` program, and save the object code:

```
: void calcsum(string scalar varname, string scalar touse)
> {
>        real colvector  x
>
>        st_view(x, ., varname, touse)
>        st_numscalar("r(sum)", colsum(x))
> }
: mata mosave calcsum(), dir(PERSONAL)
```

This step we do only once. The `mata mosave` command created file `calcsum.mo` stored in our `PERSONAL` sysdir directory; see [M-3] **mata mosave** and [P] **sysdir** for more details. We put the `calcsum.mo` file in our `PERSONAL` directory so that Stata and Mata would be able to find it, just as you probably did with the `varsum.ado` ado-file.

Typing in the `calcsum()` program interactively is too prone to error, so instead what we can do is create a do-file to perform the step and then run the do-file once:

begin `varsum.do`

```
version 17.0
mata:
void calcsum(string scalar varname, string scalar touse)
{
     real colvector  x
     st_view(x, ., varname, touse)
     st_numscalar("r(sum)", colsum(x))
}
mata mosave calcsum(), dir(PERSONAL) replace
end
```

end `varsum.do`

We save the do-file someplace safe in case we should need to modify our code later, either to fix bugs or to add features. For the same reason, we added a `replace` option on the command that creates the `.mo` file, so we can run the do-file over and over.

To follow the third organization—putting our `calcsum()` subroutine in a library—is just a matter of modifying `varsum.do` to do `mata mlib add` rather than `mata mosave`. See [M-3] **mata mlib**; there are important details having to do with how you will manage all the different functions you will put in the library, but that has nothing to do with our problem here.

Where and how you store your Mata subroutines has nothing to do with what your Mata subroutines do or how you use them.

Passing arguments to Mata functions

In designing a subroutine, you have two paramount interests: how you get data into your subroutine and how you get results back. You get data in by the values you pass as the arguments. For `calcsum()`, we called the subroutine by coding

```
mata: calcsum("‘varlist’", "‘touse’")
```

After macro expansion, that line turned into something like

```
mata: calcsum("mpg", "__0001dc")
```

The `__0001dc` variable is a temporary variable created by the `marksample` command earlier in our ado-file. `mpg` was the variable specified by the user. After expansion, the arguments were nothing more than strings, and those strings were passed to `calcsum()`.

Macro substitution is the most common way values are passed to Mata subroutines. If we had a Mata function `add(`*a*`, `*b*`)`, which could add numbers, we might code in our ado-file

```
mata: add(‘x’, ‘y’)
```

and, if macro `‘x’` contained 2 and macro `‘y’` contained 3, Mata would see

```
mata: add(2, 3)
```

and values 2 and 3 would be passed to the subroutine.

When you think about writing your Mata subroutine, the arguments your ado-file will find convenient to pass and Mata will make convenient to use are

1. numbers, which Mata calls real scalars, such as 2 and 3 (`‘x’` and `‘y’`), and
2. names of variables, macros, scalars, matrices, etc., which Mata calls string scalars, such as `"mpg"` and `"__0001dc"` (`"‘varlist’"` and `"‘touse’"`).

To receive arguments of type (1), you code `real scalar` as the type of the argument in the function declaration and then use the real scalar variable in your Mata code.

To receive arguments of type (2), you code `string scalar` as the type of the argument in the function declaration, and then you use one of the Stata interface functions (in [M-4] **Stata**) to go from the name to the contents. If you receive a variable name, you will especially want to read about the functions `st_data()` and `st_view()` (see [M-5] **st_data()** and [M-5] **st_view()**), although there are many other utilities for dealing with variable names. If you are dealing with local or global macros, scalars, or matrices, you will want to see [M-5] **st_local()**, [M-5] **st_global()**, [M-5] **st_numscalar()**, and [M-5] **st_matrix()**.

Returning results to ado-code

You have many more choices on how to return results from your Mata function to the calling ado-code.

1. You can return results in `r()`—as we did in our example—or in `e()` or in `s()`.
2. You can return results in macros, scalars, matrices, etc., whose names are passed to your Mata subroutine as arguments.
3. You can highhandedly reach back into the calling ado-file and return results in macros, scalars, matrices, etc., whose names are of your devising.

In all cases, see [M-5] **st_global( )**. `st_global()` is probably not the function you will use, but there is a wonderfully useful table in the *Remarks and examples* section that will tell you exactly which function to use.

Also see all other Stata interface functions in [M-4] **Stata**.

If you want to modify the Stata dataset in memory, see [M-5] **st_store( )** and [M-5] **st_view( )**.

Advice: Use of matastrict

The setting `matastrict` determines whether declarations can be omitted (see [M-2] **Declarations**); by default, you may. That is, `matastrict` is set `off`, but you can turn it on by typing `mata set matastrict on`; see [M-3] **mata set**. Some users do, and some users do not.

So now, consider what happens when you include Mata source code directly in the ado-file. When the ado-file is loaded, is `matastrict` set `on`, or is it set `off`? The answer is that it is `off`, because when you include the Mata source code inside an ado-file, `matastrict` is temporarily switched off when the ado-file is loaded even if the user running the ado-file has previously set it on.

For example, `varsum.ado` could read

begin `varsum.ado`

```
program varsum
     version 17.0
     syntax varname [if] [in]
     marksample touse
     mata: calcsum("`varlist'", "`touse'")
     display as txt "  sum = " as res r(sum)
end
version 17.0
mata:
void calcsum(varname, touse)
{                                          // (note absence of declarations)
     st_view(x, ., varname, touse)
     st_numscalar("r(sum)", colsum(x))
}
end
```

end `varsum.ado`

and it will work even when run by users who have `set matastrict on`.

Similarly, in an ado-file, you can `set matastrict on` and that will not affect the setting after the ado-file is loaded, so `varsum.ado` could read

begin varsum.ado

```
program varsum
     version 17.0
     syntax varname [if] [in]
     marksample touse
     mata: calcsum("`varlist'", "`touse'")
     display as txt "  sum = " as res r(sum)
end
version 17.0
mata:
mata set matastrict on
void calcsum(string scalar varname, string scalar touse)
{
     real colvector  x
     st_view(x, ., varname, touse)
     st_numscalar("r(sum)", colsum(x))
}
end
```

end varsum.ado

and not only will it work, but running `varsum` will not change the user's `matastrict` setting.

This preserving and restoring of `matastrict` is something that is done only for ado-files when they are loaded.

Advice: Some useful Mata functions

In the `calcsum()` subroutine, we used the `colsum()` function—see [M-5] **sum()**—to obtain the sum:

```
void calcsum(string scalar varname, string scalar touse)
{
     real colvector  x
     st_view(x, ., varname, touse)
     st_numscalar("r(sum)", colsum(x))
}
```

We could instead have coded

```
void calcsum(string scalar varname, string scalar touse)
{
     real colvector  x
     real scalar     i, sum
     st_view(x, ., varname, touse)
     sum = 0
     for (i=1; i<=rows(x); i++) sum = sum + x[i]
     st_numscalar("r(sum)", sum)
}
```

The first way is preferred. Rather than construct explicit loops, it is better to call functions that calculate the desired result when such functions exist. Unlike ado-code, however, when such functions do not exist, you can code explicit loops and still obtain good performance.

Another set of functions we recommend are documented in [M-5] **cross()**, [M-5] **crossdev()**, and [M-5] **quadcross()**.

`cross()` makes calculations of the form

$X'X$
$X'Z$
$X'\texttt{diag}(w)X$
$X'\texttt{diag}(w)Z$

`crossdev()` makes calculations of the form

$(X\texttt{:}-x)'(X\texttt{:}-x)$
$(X\texttt{:}-x)'(Z\texttt{:}-z)$
$(X\texttt{:}-x)'\texttt{diag}(w)(X\texttt{:}-x)$
$(X\texttt{:}-x)'\texttt{diag}(w)(Z\texttt{:}-z)$

Both these functions could easily escape your attention because the matrix expressions themselves are so easily written in Mata. The functions, however, are quicker, use less memory, and sometimes are more accurate. Also, quad-precision versions of the functions exist; [M-5] **quadcross( )**.

Also see

[M-2] **version** — Version control

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