The Stata Journal

The *Stata Journal* publishes reviewed papers together with shorter notes or comments, regular columns, book reviews, and other material of interest to Stata users. Examples of the types of papers include 1) expository papers that link the use of Stata commands or programs to associated principles, such as those that will serve as tutorials for users first encountering a new field of statistics or a major new technique; 2) papers that go "beyond the Stata manual" in explaining key features or uses of Stata that are of interest to intermediate or advanced users of Stata; 3) papers that discuss new commands or Stata programs of interest either to a wide spectrum of users (e.g., in data management or graphics) or to some large segment of Stata users (e.g., in survey statistics, survival analysis, panel analysis, or limited dependent variable modeling); 4) papers analyzing the statistical properties of new or existing estimators and tests in Stata; 5) papers that could be of interest or usefulness to researchers, especially in fields that are of practical importance but are not often included in texts or other journals, such as the use of Stata in managing datasets, especially large datasets, with advice from hard-won experience; and 6) papers of interest to those who teach, including Stata with topics such as extended examples of techniques and interpretation of results, simulations of statistical concepts, and overviews of subject areas.

For more information on the *Stata Journal*, including information for authors, see the web page

http://www.stata-journal.com

The *Stata Journal* is indexed and abstracted in the following:

- Science Citation Index Expanded (also known as SciSearch®)
- CompuMath Citation Index®

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

The Stata Journal (2008)
8, Number 3, pp. 401–412

Mata Matters: Macros

William Gould
StataCorp
College Station, TX
wgould@stata.com

Abstract. Mata is Stata's matrix language. In the Mata Matters column, we show how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. Macros are the subject of this column.

Keywords: pr0040, Mata, macros

1 Introduction

Macros, constructs such as `‘varlist’` and `‘i’`, play a major role in Stata's ado-programming language. Stata ado-file programmers use macros to generalize concepts from specific examples, to perform looping, to hold numeric or string values, and to use as the basis for decisions that then perform one action or another.

Stata (ado-file) programmers use macros to generalize concepts from specific examples such as subtracting 100 from the variable `amount` to subtracting 100 from a user-specified variable:

```
program myfixvars
        syntax varlist(max=1)
        ...
        replace `varlist´ = `varlist´ - 100
        ...
end
```

Ado-file programmers use macros to perform looping:

```
program myfixvars
        syntax varlist
        ...
        foreach var of local varlist {
                replace `var´ = `var´ - 100
        }
end
```


Ado-file programmers use macros to hold numeric or string values (although they sometimes use scalars):

```
program myfixvars
        syntax varlist

        local N   = 0
        local sum = 0
        foreach var of local varlist {
                quietly summarize `var´, meanonly
                local sum = `sum´ + r(sum)
                local N   = `N´   + r(N)
        }

        local mean = `sum´ / `N´
        foreach var of local varlist {
                replace `var´ = `var´ - `mean´
        }
end
```

And finally, ado-file programmers use macros as the basis for decisions to perform one action or another:

```
program myfixvars
        syntax varlist [, mean]

        if ("`mean´"!="") {
                local N   = 0
                local sum = 0
                foreach var of local varlist {
                        quietly summarize `var´, meanonly
                        local sum = `sum´ + r(sum)
                        local N   = `N´   + r(N)
                }
                local mean = `sum´ / `N´
        }
        else    local mean = 100

        foreach var of local varlist {
                replace `var´ = `var´- `mean´
        }
end
```

Without macros, Stata's ado-file language simply would not work as a programming language; it would degenerate to being little more than a scripting language. You could record specific actions, but you could not generalize them.

Users new to Mata assume that macros play a similarly important role in Mata. They do not. In fact, you do not even need macros to program in Mata. How you program without macros constitutes the first part of this article. What you might do with macros in Mata constitutes the second part.

2 Macroless programming in Mata

The first rule for programming in Mata is to forget that macros even exist. Macros are not necessary and, in fact, all your ado-file instincts will mislead you. Everything macros do for you in ado-files is performed in Mata by Mata variables.

Consider the name of a variable in your Stata dataset. In ado-files, that name would be recorded in a macro. In Mata, the name will be recorded in a string variable:

```
function fixvars(string scalar varname)
{
       real vector    x

       st_view(x, ., varname)
       x[.] = x :- 100
}
```

We could call this Mata program directly from Stata by typing

```
. mata: fixvars("myvar")
```

or we could write an ado-file interface:

```
program myfixvars
        syntax varlist(max=1)
        mata: fixvars("`varlist´")
end
```

With the ado-file interface, you can use this program just as you did when the program was written entirely in Stata.

The example is silly, but the underlying logic applies equally to substantive programming efforts. This style of combining ado-files and Mata code—parsing performed in Stata and the substantive work performed in Mata—is often used in official ado-files. Here the Stata command `syntax` set up the macro `varlist` to contain a variable name, and then the variable name was passed to our Mata program when we coded `mata: fixvars("`varlist'")`. If `varlist` contained the variable name `blresp`, the line would read `mata: fixvars("blresp")` after expansion. `fixvars("blresp")` is perfectly good Mata syntax; `"blresp"` is a string literal in Mataspeak. On the Mata side of things, execution of `fixvars()` begins with

```
function fixvars(string scalar varname)
{
       real vector    x

       st_view(x, ., varname)
       x[.] = x :- 100
}
```

The variable name `"blresp"` is stored in the Mata variable `varname`, and after that, the rest is pure Mata. In this code, we set up a view onto the variable and then subtracted 100 from each of its elements. How that works was discussed in a previous column (Gould 2005). What's important to note this time are the details. In Mata, you do

not type `st_view(x, ., blresp)` to set up a view onto `blresp`; instead, you type `st_view(x, ., "blresp")`. The variable name appears in quotes because `st_view()` expects a string scalar argument and `"blresp"` is one way of specifying a string scalar. In our program, however, we did not even code `st_view(x, ., "blresp")` because we did not want a view onto the fixed variable `blresp`. We wanted a view onto a user-specified variable. We coded `st_view(x, ., varname)`, where `varname` was the name of the Mata variable that contained the Stata variable name.

Now consider a list of Stata variable names. In ado-files, those names would be recorded in a macro, one name after the other, with spaces in between. In Mata, the list might be formatted the same way, but even so it would be recorded in a string scalar. Alternatively, the list of names might be recorded in a string vector with each name occupying an element of the vector. Lists of variables are often received from Stata in the first format, but they are easier to use in Mata when they are in the second. Mata's `tokenize()` function converts the first format to the second. The following program uses this approach:

```
function fixvars(string scalar varnames)
{
        string vector    varvec
        real vector      x

        varvec = tokenize(varnames)

        for (j=1; i<=length(varvec); j++) {
              st_view(x, ., varvec[j])
              x[.] = x :- 100
        }
 }
```

Another way we could have written the above program is

```
function fixvars(string scalar varnames)
{
        string vector    varvec
        real matrix      X

        varvec = tokenize(varnames)
        st_view(X, ., varvec)

        for (j=1; j<=length(varvec); j++) {
              x[.,j] = x[.,j] :- 100
        }
}
```

In the code above, the setting up of the view is moved outside the loop. In terms of execution time, it does not matter much either way. In fact, the loop itself can be removed, and we could code

```
function fixvars(string scalar varnames)
{
        real matrix      X

        st_view(X, ., tokenize(varnames))
        X[.,.] = X :- 100
}
```

With any of the above Mata programs, our ado-file to call it could read

```
program myfixvars
        syntax varlist
        mata: fixvars("`varlist´")
end
```

In all three versions, Stata's `‘varlist’` macro, in quotes, expands to the space-separated names. Placed in double quotes, `"‘varlist’"`, the quoted and expanded macro, looks like a Mata string literal. That string literal is received by the Mata program `fixvars()` and stored in the string scalar `varnames`. Inside `fixvars()`, `tokenize(varnames)` returns a string vector. If, for example, `"‘varlist’"` were `"mpg weight displ"`, then `tokenize(varnames)` returns the vector (`"mpg"`, `"weight"`, `"displ"`). Macros were used on the Stata side but not on the Mata side.

The final version of the Stata ado-file `myfixvars` in the introduction subtracted 100 from the variables specified or subtracted the overall group mean calculated across variables if the option `mean` was specified. Here is the code in combined Stata/Mata to do that:

```
program myfixvars
        syntax varlist [, means]
        mata: fixvars("`varlist´", "`means´")
end

mata:
function fixvars(string scalar varnames, string scalar means)
{
        real matrix      X
        real scalar      tosub

        st_view(X, ., tokenize(varnames))
        tosub = (means=="" ? 100 : sum(X)/nonmissing(X))
        X[.,.] = X :- tosub
}
end
```

This code takes fewer lines than our original, is more readable, and will run more quickly. On the Mata side of things, we never needed macros.

3 Dealing with macros in Mata

Mata does not need macros, but that does not mean you cannot work with macros in Mata. Macros are such an important part of Stata that you will sometimes want to access a macro's contents or change those contents. The Mata function `st_local(`*name*`)`

will return the contents of a local macro. For instance, `st_local("varlist")` returns the contents of the local macro `varlist`; the syntax is `st_local("varlist")`, not `st_local("‘varlist’")` or `st_local(‘varlist’)`. `st_local()` can reset the contents of macros, too. `st_local(`*name*`,` *contents*`)` sets the contents of the Stata local macro *name* to be *contents*. *contents* is a string scalar, as is *name*. You can access and reset global macros by using the function `st_global()`, which has the same syntax as `st_local()`.

To demonstrate the use of these functions, we are going to rewrite our final version of `myfixvars`/`fixvars()` to use them. There is even good reason that we might want to do that. Using our final version of `myfixvars`, written in combined Stata and Mata, pretend we typed

```
. myfixvars q1-q5, means
```

Then the line that reads `mata: fixvars("‘varlist’", "‘means’")` would expand to

```
mata: fixvars("q1 q2 q3 q4 q5", "means")
```

Now imagine that we typed

```
. myfixvars surveyquestion1-surveyquestion2000, means
```

I will not show you the result of expanding `mata: fixvars("‘varlist’", "‘means’")` because it would take 10 pages to print. `"‘varlist’"`, when expanded, is a 36,892-character string, not counting the quotes. Despite the length of the input string, however, our `myfixvars`/`fixvars()` solution will work, although it is unfortunate that the string is so long because there will be a lot of copying of the expanded `"‘varlist’"` string before it finally makes its way to Mata's `varnames` variable. First, the string will be expanded and stored in Stata's `varlist` macro. Then the `varlist` macro will be expanded in the ado-file when it executes the line `mata: fixvars("‘varlist’", "‘means’")`. Next that line will be passed to Mata for execution. All told, the 36,892-character string will be copied three times before `fixvars()` begins execution, but things will proceed efficiently after that.

You should not be overly concerned about efficiency, but if you have concerns, there is a solution. We could change our combined Stata/Mata program to read

```
program myfixvars
        syntax varlist [, means]
        mata: fixvars("varlist", "`means´")
end

mata:
function fixvars(string scalar varnames, string scalar means)
{
        real matrix      X
        real scalar      tosub

        st_view(X, ., tokenize(st_local(varnames)))
        tosub = (means=="" ? 100 : sum(X)/nonmissing(X:<=.))
        X[.,.] = X :- tosub
}
end
```

The differences are subtle. First, I changed

```
        mata: fixvars("`varlist´", "`means´")
```

to read

```
        mata: fixvars("varlist", "`means´")
```

That is, I removed the single quotes around `varlist` so that it would not be expanded. In the updated version, I pass the name of the macro to `fixvars()` rather than the names of the variables. Then I changed

```
        st_view(X, ., tokenize(varnames))
```

to

```
        st_view(X, ., tokenize(st_local(varnames)))
```

Previously, `varnames` contained the variable names, but now it contains `"varlist"`, the name of the macro that contains the variable names, and `st_local("varlist")` returns the contents of the macro. Thus, in the improved version, rather than depending on Stata and Mata to pass the contents of the macro to me, I inserted code to copy the contents for myself. By doing that, I reduced execution time because Stata no longer needs to expand the macro and the Mata compiler no longer needs to deal with a potentially long expanded macro (now called a string literal in Mataspeak).

I could go back and do the same thing with the `"‘means’"` macro, passing its name rather than its contents, but I would never bother. I know `"‘means’"` expands either to `""` or to `"means"`, and that is not long enough to be worth the trouble. It is not clear whether the change was worth the bother even in the case of `varlist`.

It is important to understand, however, that you can access the current contents of Stata macros in your Mata code. You do this not by coding `x = ‘value’`, but instead by coding

```
x = st_local("value")
```

In a similar way, you can access (and change) Stata's scalars and matrices by using the Mata functions `st_numscalar()` and `st_matrix()`.

4 How Mata deals with macros

Let's understand what would happen if you were to code `x = ‘value’` in Mata. Let's assume `‘value’` is 3. In Stata, when you type

```
. generate x = 3
```

Stata just does it. Stata goes directly from what you type to filling in `x` in one step. There is work involved in that. Stata studies (parses) the line to figure out what the line is telling Stata to do, but after figuring that out, Stata just does it.

That is very different from what Mata does when you type

```
: x = 3
```

Mata breaks the problem into two parts, called compilation and execution. In the compilation step, Mata translates what you type into a byte code that means

```
push_literal 3
store        x
```

and in fact reads

```
001f00000000122a785a000500000000173a227c
```

In the above, I assume the literal 3 is stored at the computer address 0x122a785a and matrix `x` is stored at 0x173a227c. The compiled version looks like a mess, but it is something your computer can execute very quickly. The above compiled code can be stored and executed in the future, and that ability to reexecute compiled code is what makes Mata so fast. The difference between Stata (an interpreter) and Mata (a compiler) is that every time Stata sees a line, it must figure out what it means, and that happens even when Stata sees the same line repeatedly. When Stata executes

```
forvalues i=1(1)100 {
        local x`i´ = 0
}
```

Stata literally looks at the line `local x‘i’ = 0` one hundred times, and each time, Stata behaves as if it is seeing the line for the first time! Mata, executing

```
for (i=1; i<=100; i++) {
        x[i] = 0
}
```

figures out what it means just once.

Going back to our discussion, we are considering `x = 3`, and we just learned that to achieve that, Mata executes the code `001f00000000122a785a000500000000173a227c`.

Now think about the Stata line

```
. generate x = `value´
```

Let's pretend the macro `‘value’` contains 3. The result is that 3 is stored in `x`. If at some future time you reexecute the line `generate x = ‘value’` and the contents of `‘value’` are different, the results will be different. They will be different because, every time Stata sees a line such as `generate x = ‘value’`, Stata reinterprets its meaning. On the other hand, if in Mata you code

```
: x = `value´
```

Mata goes through its two-step logic. After substitution, the line reads `x = 3`, and after compilation, we have

```
001f00000000122a785a000500000000173a227c
```

That is exactly the same compiled code we had before. When the compiled code is executed, the result is that `x` contains 3. The difference between Stata and Mata is that, if later we reexecute this code, the result will be unchanged even if `‘value’` changes. That will happen within a loop even if the value of the macro is changing. We code `x = ‘value’` and what is substituted for `‘value’` is the contents of `value` at the time the code was compiled, not when it is executed. That will also happen across loops and programs. If you have the line `x = ‘value’` embedded in a longer program,

```
function myfunction(...)
{
        ...
        x = `value´
        ...
}
```

then every time you execute `myfunction()`, the value of `x` will be set to the contents of `‘value’` as it was at the time of compile. That could be the value two seconds ago, two minutes ago, or if you save compiled programs in libraries, two days or even two years ago.

5 Putting compile-time macros to use in Mata

There is, however, a good use for lines such as

```
x = `value´
```

once you understand that the value substituted will be the value at compile time.

Let's pretend we are writing a program that has four alternative ways to calculate its estimate of variance. One calculation is used if the user specifies no options, another if the user specifies `dofadjusted`, another if the user specifies `unequal`, and yet another if the user specifies both options together. Here is very readable code to accomplish this:

```
local RS            real scalar
local SS            string scalar

local Varlist       `SS´

local VarType       `RS´
local VT_dflt       1
local VT_dofadj     2
local VT_uneq       3
local VT_dofuneq    4

mata:

function calculation(`Varlist´ vars, `SS´ op_dofadj, `SS´ op_uneq)
{
        `VarType´   vt

        vt = parse_options(op_dofadj, op_uneq)
        make_base_calculation(vars)
        make_var_calculation(vt)
}

`VarType´ parse_options(`SS´ op_dofadj, `SS´ op_uneq)
{
        if (op_dofadj!="") {
                return(op_uneq=="" ? `VT_dofadj´ : `VT_dofuneq´)
        }
        else if (op_uneq!="") {
                return(`VT_uneq´)
        }
        return(`VT_dflt´)
}

void make_base_calculation(`Varlist´ vars)
{
        ...
}

void make_var_calculation(`VarType´ vt)
{
        if (varcalc == `VT_dflt´) {
                ...
        }
        else if (varcalc == `VT_dofadj´) {
                ...
        }
        else if (varcalc == `VT_uneq´) {
                ...
        }
        else if (varcalc == `VT_dofuneq´) {
                ...
        }
        else    _error(3999)
}

end
```

The macros used above are all defined at compile time and intended to be substituted at compile time; they are defined right at the top, in the same file as the code itself.

The first macros defined are `RS` and `SS`, and they are shorthands for `real scalar` and `string scalar`. Type declarations are optional in Mata, but we at StataCorp use them in all official code. It gets tiresome typing out `real scalar`, `string scalar`, and the like, so our style is to define local macros such as `RS`, `SS`, etc., meaning the same thing. Throughout the rest of the code, I use `'RS'` rather than `real scalar` and `'SS'` rather than `string scalar`. This saves typing and makes code more readable because it makes lines shorter.

The next macro defined is `Varlist`. StataCorp style is to define new types designating how variables are used rather than how they happen to be stored. `'Varlist'` literally means `string scalar`, as does `'SS'`. The `'Varlist'` type is not just any string scalar, however; it is a string scalar containing space-separated Stata variable names. Using `'Varlist'` in place of `'SS'` makes code even more readable.

Finally, there is a block that defines `VarType` along with `VT_dflt`, `VT_dofadj`, `VT_uneq`, and `VT_dofuneq`. The first of these is a new type in the same spirit as `Varlist`. `'VarType'` is in fact `real scalar`, but the name was chosen to indicate that it contains a particular numeric coding. The remaining macros define the numeric code. In the subsequent Mata program, if a variable is `'VarType'`, that means it contains a numeric code for the variance calculation to be made. And in the program, we never refer to the numeric codes themselves; we refer to their more readable equivalents: `'VT_dflt'`, `'VT_dofadj'`, `'VT_uneq'`, and `'VT_dofuneq'`. Thus the variable `vt` might be a `'VarType'`, and it might contain the value `'VT_dflt'`.

In describing the above style, I have written phrases like, "StataCorp style is ...". I am not laying down the law. These are local macros defined in this file for the purposes of this code, the intent being to make this code more readable and hence less likely to contain errors, and to be more easily modified later. Local macros come into existence when the file is read and disappear thereafter. The local macros defined in this file have no implications for other files, and so varying styles can be combined freely. We at StataCorp do not have a committee meeting every time a developer wants to use a new macro-defined type or code. StataCorp style is the above, taken generically. StataCorp law is that code will be readable. Comments, written between `/*` and `*/`, are sometimes used to improve readability, but we at StataCorp want to avoid code-plus-comments of the form

```
vt = 3    /* remember, 3 means VT_uneq.  Or is it 4? */
```

It is easy to confuse the meaning of 1, 2, 3, and 4. It is more difficult to confuse `'VT_dflt'`, `'VT_dofadj'`, `'VT_uneq'`, and `'VT_dofuneq'`.

You can look at the code and decide for yourself whether this use of macros improves readability. There is only one thing I want to draw your attention to, and that is the end of `make_var_calculation()`. There are four potential variance calculations, and many programmers would have been tempted to code the selection as

```
if (varcalc == `VT_dflt´)         {  calculation 1 }
else if (varcalc == `VT_dofadj´)  {  calculation 2 }
else if (varcalc == `VT_uneq´)    {  calculation 3 }
else                              {  calculation 4 }
```

I coded it as

```
if (varcalc == `VT_dflt´)          {  calculation 1 }
else if (varcalc == `VT_dofadj´)   {  calculation 2 }
else if (varcalc == `VT_uneq´)     {  calculation 3 }
else if (varcalc == `VT_dofuneq´)  {  calculation 4 }
else                               _error(3999)
```

The second style is preferred. It is too easy to omit a code and, if you omit one using the first style, you might not notice because you will obtain the ‘`VT_dofuneq`’ result. In the second style, if you omit a code, you will get an error. In addition, we at StataCorp often go back and add additional calculations. It is even easier to omit a code in that case.

6 Conclusion

In summary, here is my advice:

1. Do not use ‘*name*’ or $*name* to refer to the run-time contents of Stata macros. Moreover, Stata macros play no formal role in Mata’s programming language.

2. Do use `st_local(`*name*`)` and `st_global(`*name*`)` to access the contents of local and global macros should the need arise. Use `st_local(`*name*`,` *contents*`)` and `st_global(`*name*`,` *contents*`)` to set local and global macros.

3. Do use ‘*name*’ to refer to the compile-time contents of Stata macros. Do this to improve the readability of your code.

7 Reference

Gould, W. 2005. Mata Matters: Using views onto the data. *Stata Journal* 5: 567–573.

About the author

William Gould is president of StataCorp, head of development, and principal architect of Mata.