The Stata Journal

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

The Stata Journal (2006)
6, Number 1, pp. 112–123

Mata Matters: Creating new variables—sounds boring, isn't

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. In this quarter's column, we continue to explore the handling of Stata datasets in Mata and focus on creating new variables.

Keywords: pr0021, Mata, views, Stata datasets

Reprise

Think of Mata as being separate from Stata, but with functions that will allow Mata to access, manipulate, and change Stata objects. The most important Stata object is the dataset. In last quarter's column, we discussed how the Mata function `st_view()` and, to a lesser extent, `st_data()`, could be used to access the dataset. To summarize, using `st_view()`, you can obtain Mata matrices that are, in fact, views onto Stata's dataset, or even portions of it. Using the matrices accesses the dataset. Changing the matrices changes the dataset. `st_data()`, on the other hand, returns a copy of the data, and changing the values in the matrix does not change the dataset—which can be advantageous or not, depending on application.

Thus if I want to obtain variable `mpg` as a column vector, I could code

```
st_view(x=., ., "mpg")
```

or I could code

```
x = st_data(., "mpg")
```

It makes no difference which I code as long as my intention is to use and not to reset the values of `mpg`, although the view uses less memory. With either definition of `x`, I could code

```
mean = colsum(x)/rows(x)
```

to obtain the mean. With the view, I could also change the values of `mpg` to be their deviations from the mean by then coding

```
x[.] = x :- mean
```


I use the colon-minus operator because `x` is $n \times 1$ and `mean` is 1×1. I use `x[.]` on the left, rather than just `x`, for reasons that I will explain later. If I had defined `x` as `st_data(., "mpg")`, subsequently coding `x[.] = x :- mean` would have changed the Mata vector `x` but would have left the underlying Stata variable unchanged. If I wanted to change variable `mpg` in this case, I would have to code

```
st_store(., "mpg", x:-mean)
```

We did not discuss `st_store()` in the previous column because the view created by `st_view()` can be used for both reading and writing the data. Sometimes, however, using `st_store()` is easier or makes the intention clearer. We also did not discuss adding new variables to the Stata dataset. Both subjects we will discuss below.

Creating new variables

Function `st_addvar()` will add variables to the Stata dataset. The function is in one sense unnecessary because you could structure your code so that new variables were added before calling Mata, and your Mata routines could simply fill in the values of the already-created variables. That arrangement can be inconvenient, especially if the number of new variables created somehow depends on values in the dataset.

Consider an example. Here is part of a dataset on patients.

```
. list in 1/5
```

	patid	bp1	bp2	bp3	bp4	bp5
1.	1	170	168	166	163	161
2.	2	158	156	154	151	147
3.	3	161	158	155	151	147
4.	4	155	153	151	148	144
5.	5	158	160	160	161	162

The variables `bp1`, `bp2`, ..., `bp5` record blood pressure measurements at times 1, 2, ..., 5. To this dataset, assume that we want to add new variable `b`, recording each observation's trend coefficient from a regression of `bp` on measurement time. The new variable `b` would contain -2.3 in observation 1, the result of running a regression of $(170, 168, 166, 163, 161)'$ on $(1, 2, 3, 4, 5)'$, and contain -2.7 in observation 2, the result of running a regression of $(158, 156, 154, 151, 147)'$ on $(1, 2, 3, 4, 5)'$. We might use the new variable `b` as the dependent variable in a regression if we thought that, because of a treatment, blood pressure should decline linearly over the five measurements. If we thought the decline should be larger at earlier times, we might use the log of blood pressure.

In any case, pretend that I had a Mata routine, `trends(`*real matrix Y*`)`, that returned a column vector of the desired `b`'s. That is, given the matrix

$$Y = \begin{pmatrix} 170 & 168 & 166 & 163 & 161 \\ 158 & 156 & 154 & 151 & 147 \\ 161 & 158 & 155 & 151 & 147 \\ 155 & 153 & 151 & 148 & 144 \\ 158 & 160 & 160 & 161 & 162 \end{pmatrix}$$

`trends()` would return $(-2.3, -2.7, -3.5, -2.7, 0.9)'$. `trends()` will be easy to write. Given `trends()`, one solution to our problem would be

```
mata:  do_trend_var("b", "bp1 bp2 bp3 bp4 bp5")
```

after defining

```
function do_trend_var(string scalar newvarname, string scalar varnames)
{
        real matrix     Y
        real colvector  b
        real scalar     idx

        st_view(Y, ., tokens(varnames))
        b   = trends(Y)
        idx = st_addvar("float", newvarname)
        st_store(., idx, b)
}
```

All the routine `do_trend_var()` does is set up the view onto the data, call `trends()` to obtain the desired result, and then store that result as new Stata variable `b`.

`st_addvar()` creates the new Stata variable. `st_addvar()` requires two arguments: the type and the name of the new variable to be created. `st_addvar()` returns the index of the variable created, such as 7 if the new variable were the seventh in our dataset. So far, every time I have demonstrated a data-access function such as `st_view()` or `st_data()`, I have specified the variable names. These functions also allow you to specify variable numbers, and that saves computer time. Then the functions do not have to look up the names to get to the numbers, which is how Stata tracks variables internally.

`st_store()` will also accept variable names or numbers. Above, I used the number returned by `st_store()`, but I could just as well have used the name by changing the last two lines to read as follows:

```
(void) st_addvar("float", newvarname)
st_store(., newvarname, b)
```

Here I needed to insert `(void)` in front of `st_addvar()` to discard the returned result. Otherwise, the result would have been printed, just as Mata does with any unassigned expression.

Rather than using st_store() to save the values, I could have created a view onto the new variable and then stored the values in it. In this case, I could change our code to read

```
function do_trend_var(string scalar newvarname, string scalar varnames)
{
        real matrix     Y
        real colvector  b

        st_view(Y, ., tokens(varnames))
        st_view(b, ., st_addvar("float", newvarname))
        b[.] = trends(Y)
}
```

I like this solution, but understand that we are talking about issues of style, not substance. st_view(), just like st_store(), will accept variable numbers or names, and so I substituted the call to st_addvar() as st_view()'s third argument. This approach saves a line and, in the process, makes my intention clearer.

The last line is worthy of notice:

```
b[.] = trends(Y)
```

It would not do to code

```
b = trends(Y)
```

even though, in the first draft, I did just that and then wondered why my program did not work. It did not work because b = trends(Y) replaces the definition b, and what we want is to replace b's elements while maintaining b's definition as a view matrix.

I know this sounds like a fine distinction, so consider another, easier case. In some other problem, pretend that we had vector v, and to keep it simple, v is not even a view. It is just a regular Mata matrix. Let us pretend that v, right now, is 3×1. If we were to code

```
v = z
```

and if z were 18×1, we would not expect the line to generate an error, and it does not. It does not generate an error because the result on the right *replaces* the definition on the left. Old 3×1 vector v is discarded and replaced with a brand new 18×1 vector. On the other hand, had we coded

```
v[.] = z
```

we would expect an error, because that line says to replace the elements of the existing v, and existing 3×1 vector v cannot hold 18 elements.

Usually this fine distinction does not matter. With views, however, it does. If you code

```
b = trends(Y)
```

you are saying that the result on the right *replaces* the definition on the left. Eliminate existing matrix `b`, which happens to be a view, and create a shiny new matrix conformable with the result from `trends()`. In this case, however, we want to maintain the existing definition of `b` and merely replace its elements. `b` is a view, and changing its elements is what changes the underlying Stata dataset.

The distinction between `v[.] = z` and `v = z` (and in the matrix case, `v[.,.] = z` and `v = z`) arises only when `v` is a view and then only when replacing every element of it.

If you find this approach confusing, then use `st_store()`. Using `st_store()`, we can make an equally concise version of our code:

```
function do_trend_var(string scalar newvarname, string scalar varnames)
{
        real matrix     Y

        st_view(Y, ., tokens(varnames))
        st_store(., st_addvar("float", newvarname), trends(Y))
}
```

Both final versions are equally good in terms of execution speed. Both routines emphasize just how simple our solution is: get a view onto the trended variables, pass the view to `trends()`, and finally save `trends()`'s result in new variable *newvarname*.

All we need to make this work is the `trends()` routine. Here's one:

```
real colvector trends(real matrix Y)
{
        real scalar         i
        real colvector      y
        real matrix         X
        real colvector      b

        b = J(rows(Y), 1, .)
        for (i=1; i<=rows(Y); i++) {
                y    = Y[i,.]'
                X    = (1::rows(y)), J(rows(y),1,1)
                b[i] = (invsym(X'X)*X'y)[1]
        }
        return(b)
}
```

In understanding the above code, remember that the rows of `Y`, not its columns, record observations. Each row records a separate regression problem. Allow me to assist you in interpreting the code. For each row `i` of `Y`—written in Mataspeak as `Y[i,.]`—we copy the values into column vector `y` (`y = Y[i,.]'`), and we form the corresponding `X` matrix (`X = (1::rows(y)), J(rows(y),1,1)`). Essentially, we set `y` and `X` so that

$$\texttt{y} = \begin{pmatrix} \texttt{Y[i,1]} \\ \texttt{Y[i,2]} \\ \texttt{Y[i,3]} \\ \texttt{Y[i,4]} \\ \texttt{Y[i,5]} \end{pmatrix} \qquad \texttt{X} = \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 4 & 1 \\ 5 & 1 \end{pmatrix}$$

We then calculate the regression coefficients as usual: `invsym(X'X)*X'y`. That calculation results in a two-element vector, the first element of which is the trend coefficient, and that value (`(invsym(X'X)*X'y)[1]`) we store in `b[i]`. Filling in `b` element by element,

```
b[i] = (invsym(X'X)*X'y)[1]
```

requires that `b` already exist and be of the appropriate dimension. Before our loop, I defined `b` to be `J(rows(Y), 1, .)`, making it `rows(Y)` $\times$ 1 and placing missing values in it.

Program `trends()` is slick:

1. `trends()` focuses on calculation. `trends()` has nothing to say about where the data `Y` came from or what is done with the result, `b`. Given `Y`, it returns `b`, and `Y` might be a view, or not, and `b` might be stored back in the Stata dataset, or not. All that is up to the caller.

2. Nowhere in `trends()` is it coded that we intend to calculate trends based on five measurements. `trends()` gets that information from `cols(Y)` and would work just as well on three measurements or a hundred.

`trends()`, however, can be improved. For reasons of speed and accuracy, I mentioned in the previous column that calculations such as `invsym(X'X)*X'y` are better made as `invsym(cross(X,X))*cross(X,y)`. Here it hardly matters, but there is no reason to use a poorer alternative when a better one exists.

More importantly, `trends()` does not handle missing values, and we need to fix that. If one of the rows of input matrix `Y` were $(170, 168, ., 163, 161)$, rather than producing a missing value for the trend coefficient, we would prefer that the coefficient be calculated from a regression of $(170, 168, 163, 161)'$ on $(1, 2, 4, 5)'$.

An advantage of using `cross()` over `invsym(X'X)*X'y` is that `cross()` will drop observations with missing values. But we need to be cautious—so much so that the manual actually warns against using the feature. Quoting from [M-5] **cross()**, "`cross()` automatically omits rows containing missing values in making its calculation. Depending on this feature, however, is considered bad style because so many other Mata functions do not provide that feature and it is easy to make a mistake." The manual goes on to say that the right way to handle missing values is to exclude them when constructing views and subviews. All good advice, but that advice will not help us here. Function `trends()` receives `Y` and will just have to make do with it. Given how `trends()` will be

used, it would not be reasonable to say that `trends()` should not be used with missing data, which is how we might otherwise solve, so to speak, the problem.

We obtain the trend coefficient from the calculation `invsym(cross(X,X))*cross(X,y)[1]`. The documentation says `cross()` handles missing values, so isn't that good enough?

No, because in the presence of missing values, `cross(X,X)` and `cross(X,y)` could be calculated on different samples and, in our particular case, that is exactly what would happen. `X` has no missing values—it is just the numbers 1, 2, ...—so `cross(X,X)` would be computed on the full sample. Meanwhile, `cross(X,y)` would be calculated on a subsample if `y` had missing values.

I emphasize that in most statistical problems we do not have to code around missing values. In most statistical problems, the rows of the matrix are the observations and, using `st_view()` or `st_data()`, we can specify an argument that says to exclude the observations with missing values from the rows of the matrix. We do that and never concern ourselves with missing values again. In this problem, however, it is the columns of `Y` that are the observations, and the rows each record a separate regression problem, and so we do have to consider missing values.

The solution here is to code only one call to `cross()`, including all the data, so that `cross()` can observe the missing values and make a consistent calculation. That is really what the warning in the manual was trying to say: if you depend on `cross()`'s automatically dropping missing values, you can call the function only once, because otherwise you run the risk of different calls' using different samples. The choice is either one call, or eliminate the missing values so that the multiple calls are certain to use the same sample.

We can adopt the one-call solution if we use our single call to calculate

$$\begin{pmatrix} y & X \end{pmatrix}' \begin{pmatrix} y & X \end{pmatrix} = \begin{pmatrix} y'y & y'X \\ X'y & X'X \end{pmatrix}$$

and that we can do by coding `cross((y,X),(y,X))`. If we stored the result in `S`, then we could calculate the trend coefficient as `(invsym(XX)*Xy)[1]`, where `XX = S[|2,2\3,3|]` and `Xy = S[|2,1\3,1|]`. `[|...\...|]` is Mata notation for extracting a submatrix. Before the backslash appears the subscript of the top-left corner of the matrix to be extracted, and after the backslash appears the subscript of the bottom-right corner. Thus `S[|2,2\3,3|]` refers to the 2×2 submatrix of `S` starting at $(2, 2)$, which is $X'X$ in the display above.

Our improved code reads

```
real colvector trends(real matrix Y)
{
        real scalar         i
        real matrix         X, Z, XX, S
        real colvector      y, Xy, b
```

```
        b = J(rows(Y), 1, .)
        for (i=1; i<=rows(Y); i++) {
                y    = Y[i,.]'
                X    = (1::rows(y)), J(rows(y),1,1)
                Z    = (y, X)
                S    = cross(Z, Z)
                XX   = S[|2,2\3,3|]
                Xy   = S[|2,1\3,1|]
                b[i] = (invsym(XX)*Xy)[1]
        }
        return(b)
}
```

although we could collapse it to read

```
real colvector trends(real matrix Y)
{
        real scalar         i
        real matrix         Z, S
        real colvector      b

        b = J(rows(Y), 1, .)
        for (i=1; i<=rows(Y); i++) {
                Z    = Y[i,.]', (1::cols(Y)), J(cols(Y),1,1)
                S    = cross(Z, Z)
                b[i] = (invsym(S[|2,2\3,3|])*S[|2,1\3,1|])[1]
        }
        return(b)
}
```

We combine either of the above with

```
function do_trend_var(string scalar newvarname, string scalar varnames)
{
        real matrix     Y
        real colvector  b

        st_view(Y, ., tokens(varnames))
        st_view(b, ., st_addvar("float", newvarname))
        b[.]  = trends(Y)
}
```

which we can call by coding, either interactively or in a do- or ado-file,

```
mata:  do_trend_var("b", "bp1 bp2 bp3 bp4 bp5")
```

and we are done.

Creating multiple variables

`st_addvar()` can add multiple variables to the Stata dataset. Obviously, you could call `st_addvar()` repeatedly:

```
kidx1 = st_addvar("double", "b")
kidx2 = st_addvar("double", "c")
```

You may also code

```
kidxes = st_addvar( ("double", "double"), ("b","c") )
```

or you may code

```
idxes = st_addvar( "double", ("b","c") )
```

when all the variables are of the same type. In either case, `idxes` will be a 1×2 row vector containing the respective variable numbers. In this way, you may create as many variables as you wish.

It really does not matter which of the forms you use. The advantage of the combined call is that either all the variables are added or, if there is a problem such as insufficient memory, a bad variable name, etc., none are. That makes cleanup after failure easier.

If you want to create `k` variables and name them `a1`, `a2`, ..., use `sprintf()` to create the names:

```
varidxes  = J(1,k,.)
stubname  = "a"
for (i=1; i<=k; i++) {
        name = sprintf("%s%g", stubname, i)
        varidxes[i] = st_addvar("double", name)
}
```

If you wanted to create all the variables at once, you could code

```
varnames  = J(1,k,"")
stubname = "a"
for (i=1; i<=k; i++) {
        varnames[i] = sprintf("%s%g", stubname, i)
}
varidxes = st_addvar("double", varnames)
```

In both examples, I used `sprintf("%s%g", stubname, i)` to create variable names. If you preferred, you could use `stubname+strofreal(i)`. It makes no difference.

Creating temporary variables

To add temporary variables to the Stata dataset, you combine `st_addvar()` with `st_tempname()`. `st_tempname()` without arguments returns one temporary variable name—it does not create the temporary variable—and so to add one temporary variable, you code

```
tmpidx = st_addvar("double", st_tempname())
```

`st_tempname()` with a real scalar argument specifies the number of temporary variable names to be returned. To add four temporary variables, code

```
tmpidxes = st_addvar("double", st_tempname(4))
```

Either way, this is an odd thing to want to do. Temporary variables, which play such an important role in ado-file programming, play no role in Mata programming. You don't need them. If you have a temporary result that you need to hold on to, store it in a Mata vector or matrix and be done with it.

The only valid reason to create temporary variables is because you are writing an ado-file utility—adding to the ado-file language, if you will—and the entire purpose is to create temporary variables for use by the ado-file. That is why temporary variables are not automatically dropped when the Mata function concludes or even when Mata itself concludes. Temporary variables are viewed as being the property of the calling ado-file and will be dropped when the ado-file ends.

Creating time-series variables

Using `st_data()`, you may use time-series operators in the variable list. There is nothing special you have to do. For example,

```
X = st_data(., ("gnp", "l.gnp"))
```

The same ease of treatment does not apply to `st_view()`. Since most time-series datasets are small in comparison with cross-sectional or longitudinal datasets, the `st_data()` solution is usually your better alternative.

You can also use function `st_tsrevar()` with `st_view()`. This topic is advanced. `st_tsrevar(`*string rowvector s*`)` examines the elements of *s*. It skips over straight variable names such as `gnp`. Where it finds a time-series–operated variable such as `l.gnp`, it creates a temporary variable in the dataset containing the appropriate values. That is, it does that unless, sometime in the past, any other routine has already created a variable containing `l.gnp` values. In that case, it finds and reuses that variable. `st_tsrevar()` returns the variable indices, either of the original variable (for `gnp`) or of the created or found variable (for `l.gnp`).

Thus, if string row-vector `tsnames` contains a list of variable names, any of which might contain time-series operators, the appropriate way to get a view onto the data is to code

```
st_view(V=., ., st_tsrevar(tsnames))
```

Any temporary variables that are created will not be dropped when the subroutine concludes, or even when Mata itself concludes. The variables will be eliminated when the calling ado-file or do-file ends. Thus the same variables will be reused from one call to the next, which is desirable because that conserves memory.

(*Continued on next page*)

Creating datasets from whole cloth

To create an all-new dataset,

1. Start with an empty dataset. If the dataset might not be empty, clear it by coding

   ```
   st_dropvar(.)
   ```

2. Add the variables by using `st_addvar()`, either in one call or in repeated calls.

3. Increase the number of observations from zero to the desired number (say, 50) by coding

   ```
   st_addobs(50)
   ```

4. Define the contents of the variables, either by creating and filling in a view onto the dataset—or by using `st_store()`.

You may also set the number of observations before adding the variables, or you may add variables, set the observations, and then add more variables. Just do not fill in values until you have set the number of observations.

Say we wish to write Mata function `random_ds(`*real scalar n*`,` *real matrix V*`)` that would set the Stata dataset to contain n observations drawn from $N(0, V)$.

Just as we did in our trend example, we will write the numeric part of our program separately and then write another routine that will call the numeric part and save the data. This time, let's write the numeric subroutine first:

```
real matrix drawnorm(real scalar n, real matrix V)
{
        return(invnormal(uniform(n,cols(V)))*cholesky(V)')
}
```

The above subroutine does all the work and, if we needed this matrix only in Mata, we would be done. The rest of our code concerns the posting of the matrix to the Stata dataset:

```
function random_ds(real scalar n, real matrix V)
{
        real scalar     i
        real matrix     data

        st_dropvar(.)

        for (i=1; i<=rows(V); i++) {
                (void) st_addvar("float", sprintf("x%g",i))
        }

        st_addobs(n)
        st_view(data, ., .)
        data[.,.] = drawnorm(n, V)
}
```

With that, we could create a 10,000-observation dataset with three variables and a .5 correlation between each pair by typing

```
mata: random_ds(10000, (1,.5,.5 \ .5,1,.5 \ .5,.5,1))
```

Note the last line of `random_ds()`:

```
data[.,.] = drawnorm(n, V)
```

We coded that and not

```
data = drawnorm(n, V)
```

for the same reason we coded `b[.] = trends(Y)` in the regression-trend example. We want to assign to the values of `data`, not redefine `data`. We could instead have used `st_store()`, as shown below.

```
function random_ds(real scalar n, real matrix V)
{
        real scalar     i

        st_dropvar(.)

        for (i=1; i<=rows(V); i++) {
                (void) st_addvar("float", sprintf("x%g",i))
        }

        st_addobs(n)
        st_store(., ., drawnorm(n, V))
}
```

About the author

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