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 (2005)
5, Number 4, pp. 567–573

Mata Matters: Using views onto the data

William Gould
StataCorp

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 issue's column, we explore view matrices, matrices that are views of the underlying Stata dataset rather than copies of it.

Keywords: pr0019, Mata, views, memory

Overview

A data matrix is a matrix in which the rows are observations and the columns are variables. For instance, say we have the following data in Stata:

```
. list

     +---------------------------+
     | mpg   weight   displa~t   |
     |---------------------------|
  1. |  22    2,930        121   |
  2. |  17    3,350        258   |
  3. |  22    2,640        121   |
  4. |  20    3,250        196   |
  5. |  15    4,080        350   |
     +---------------------------+
```

In Mata, we can obtain a copy of the data by typing

```
. mata:
------------------------------------------------- mata (type end to exit) -----
: X = st_data(., .)

: X
          1      2      3
    +----------------------+
  1 |    22   2930    121  |
  2 |    17   3350    258  |
  3 |    22   2640    121  |
  4 |    20   3250    196  |
  5 |    15   4080    350  |
    +----------------------+
```

The data matrix we have just created might be used subsequently in the matrix formula $\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, for some vector $\mathbf{y}$.

What is important to understand about the above is that `X` is a copy. If we were to modify the dataset, or even drop it, that would not change `X`. If we were to modify `X`, or even drop it, that would not change the dataset.

 pr0019

Function `st_view()` provides an alternative to `st_data()` for accessing the data stored in the Stata dataset.

```
: st_view(V=., ., .)
: V
         1      2      3
    ┌────────────────────┐
  1 │    22   2930    121 │
  2 │    17   3350    258 │
  3 │    22   2640    121 │
  4 │    20   3250    196 │
  5 │    15   4080    350 │
    └────────────────────┘
```

`V` has the same contents as `X` and can be used in the same way. For instance, we might subsequently calculate $(\mathbf{V}'\mathbf{V})^{-1}\mathbf{V}'\mathbf{y}$. The difference between `X` and `V` is that `V` is a view: matrix `V` and the Stata dataset are the same. If we were to modify `V`, the dataset would change:

```
: V[1,1] = 500
: end
──────────────────────────────────────────────────────────────────────
. list

     ┌──────────────────────────┐
     │ mpg   weight   displa~t │
     ├──────────────────────────┤
  1. │ 500    2,930        121 │
  2. │  17    3,350        258 │
  3. │  22    2,640        121 │
  4. │  20    3,250        196 │
  5. │  15    4,080        350 │
     └──────────────────────────┘
```

Similarly, if we were to modify the dataset, `V` would change:

```
. replace mpg = 2 in 1
(1 real change made)
. mata:
─────────────────────────────────────────── mata (type end to exit) ────
: V
         1      2      3
    ┌────────────────────┐
  1 │     2   2930    121 │
  2 │    17   3350    258 │
  3 │    22   2640    121 │
  4 │    20   3250    196 │
  5 │    15   4080    350 │
    └────────────────────┘
```

The other difference is in the amount of memory consumed by `V` and `X`:

```
: mata describe
      # bytes   type                        name and extent
-------------------------------------------------------------------------------
           16   real matrix                 V[5,3]
          120   real matrix                 X[5,3]
-------------------------------------------------------------------------------
```

Both matrices are 5×3, but `X`, being a copy, consumes $5 \times 3 \times 8 = 120$ bytes, whereas `V`, being a view, consumes only 16 bytes. The difference, $120 - 16 = 104$ bytes, is not much, but were the matrices larger, the difference would become larger, too.

Let's assume we have a $100{,}000 \times 3$ dataset. Then here is what we would see:

```
: X = st_data(., .)
: st_view(V=., ., .)
: mata describe
      # bytes   type                        name and extent
-------------------------------------------------------------------------------
           16   real matrix                 V[100000,3]
    2,400,000   real matrix                 X[100000,3]
-------------------------------------------------------------------------------
```

`X`, our copy, would have taken $100{,}000 \times 3 \times 8 = 2{,}400{,}000$ bytes. `V`, our view, would still have taken only 16 bytes. Now there is a difference of 2,399,984 bytes.

Memory savings is the most important reason to use views. Data matrices are usually the largest matrices in matrix calculations, and it is often convenient to have more than one. With views, it does not matter how many matrices you have.

Most views take more than 16 bytes, but they never take much, especially in comparison with a copy. So far, we have included all the variables and all the observations. If we select some variables and omit others, a little more memory will be required—4 bytes per variable selected in the worst case, and sometimes fewer.

If we omit some observations and include others, we will similarly face a 4-byte-per-included-observation cost in the worst case, and just as with variables, sometimes it will be fewer.

Although memory savings is the most important feature of views, I will show that the ability to change the matrix and change the underlying data can also be put to good use.

Selecting subsets

The basic recipe for creating a view is

`st_view(`*V*`=., ., .)`

where you substitute for *V* the name of the matrix you wish to create. Do not get hung up on how odd the first argument, *V*`=.`, looks. Just change *V* to the name of your matrix or, if you prefer, you can type

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

You must code V`=.` one way or the other because the arguments to `st_view()` must already exist, just as for any function. The real question is not why you have to code V`=.` so much as why the syntax is not V = `st_view(.,.)`. That is because matrix V will be special and `st_view()` must lay its hands on V to make it special. In the process, it does not matter what V contained because `st_view()` re-creates it. When V already exists, you can dispense with the V`=.` if you wish. You can also dispense with the preassignment when writing code within Mata programs. In both cases, it does not matter which you do.

In any case, the odd-looking first argument V`=.` specifies the matrix to be created. The second argument specifies the observations to be included, and the third argument specifies the variables. Specifying the second argument as `.` means to include all observations. Specifying the third argument as `.` means to include all variables.

Usually, in interactive use, you will want all the observations, but it is rare that you will want all the variables. Let's assume that, using the auto data, we wish to calculate

$$\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

for

$$\begin{aligned} \mathbf{y} &= (\texttt{mpg}) \\ \mathbf{X} &= (\texttt{weight}, \texttt{foreign}, 1) \end{aligned}$$

The solution using views is

```
. sysuse auto, clear
(1978 Automobile Data)
. gen one = 1
. mata:
------------------------------------------------- mata (type end to exit) -----
: st_view(y=., ., "mpg")
: st_view(X=., ., ("weight", "foreign", "one"))
: b = invsym(X'X)*X'y
: b
                 1
    +----------------+
  1 |  -.0065878864  |
  2 |  -1.650029106  |
  3 |   41.67970233  |
    +----------------+

: end
-------------------------------------------------------------------------------
```

Note the two calls to `st_view()`. To create `y`, we specified the third argument as `"mpg"`. To create `X`, we specified the third argument as `("weight", "foreign", "one")`. The third argument specifies the variables to be selected, and the argument is specified as a row vector of names.

Once the view matrices are created, we use them just as we would any matrix. The bulk of the calculation is

```
b = invsym(X'X)*X'y
```

whether `X` and `y` are views or copies. And, whether `X` and `y` are views or copies, it would be better if the calculation were coded as

```
b = invsym(cross(X,X))*cross(X,y)
```

because function `cross()` is more accurate, faster, and uses less memory than multiplication in these sorts of situations; see [M-5] **cross()**. This detail has nothing to do with these discussions, but both the numerical analyst and the programmer in me demand that I mention that.

Missing values

In calculations like $\mathbf{b} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$, missing values will result in missing results. Had there been any missing values in the data, the final result would have been

```
: b
        1
    +-----+
  1 |  .  |
  2 |  .  |
  3 |  .  |
    +-----+
```

Let us assume that we wish simply to ignore observations that contain missing values. The easiest way is to drop any observations containing them before creating the views. In Stata, there are many ways of finding and eliminating observations that contain missing values. I use the following,

```
. egen missing = rowmiss(mpg weight foreign)
. drop if missing
```

and so our solution would become

```
. sysuse auto, clear
. gen one = 1
. egen missing = rowmiss(mpg weight foreign)
. drop if missing
. mata:
: st_view(y=., ., "mpg")
: st_view(X=., ., ("weight", "foreign", "one"))
: b = invsym(X'X)*X'y
: end
```

That, however, is not the solution I would choose in a programming context. `st_view()` allows an optional fourth argument in which you can specify the name of a variable that marks the observations to be included, which is often called a `touse` variable. In a programming context, I would create a `touse` variable in the standard way—`touse` variables contain nonzero for observations to be used and zero for observations to be omitted—and then I would specify that variable's name as the fourth argument. Do not get hung up on this because the point of this column is interactive use, but I do want to show programmers how this would be done:

top: myreg.ado

```
program myreg
        version 9
        syntax varlist [if] [in]
        marksample touse
        tempvar one
        qui gen byte `one'=1

        mata: myreg("`varlist' `one'", "`touse'")
end

version 9
mata:
function myreg(string scalar varnames, string scalar touse)

        string rowvector        vars, rhsvars
        string scalar           lhsvar
        real matrix             X
        real colvector          y

        vars = tokens(varnames)
        lhsvar  = vars[1]
        rhsvars = vars[|2\.|]
        st_view(X, ., rhsvars, touse)
        st_view(y, ., lhsvar, touse)
        invsym(cross(X,X))*cross(X,y)

end
```

end: myreg.ado

Using views to replace values in the dataset

When you replace a value in a view, you also change the value in the underlying Stata dataset. This feature can be useful in data-management problems.

Say you have a dataset containing the variables `stat72`, `stat73`, ..., `stat99` that record a patient's status in 1972, 1973, ..., 1999. You wish to add new variable `firstyear` recording the first year in which a status variable takes on the value 1.

The standard solution to this problem involves reshaping the data to long form, using standard commands to fill in variable `firstyear`, and reshaping the data back to wide form.

Here is another solution:

```
. gen byte firstyear = .

. mata:

: names =            "stat72 stat73 stat74 stat75 stat76 stat77 stat78 stat79
        stat80 stat81 stat82 stat83 stat84 stat85 stat86 stat87 stat88 stat89
        stat90 stat91 stat92 stat93 stat94 stat95 stat96 stat97 stat98 stat99"

: st_view(s=., ., tokens(names))

: st_view(first=., ., "firstyear")

: for (i=1; i<=rows(s); i++) {
>       for (j=1; j<=cols(s); j++) {
>               if (s[i,j]==1) {
>                       first[i] = j+71       /* <- note this line */
>                       break
>               }
>       }
>
> }

: end
```

In this program, we use two views.

`s` is a view onto variables `stat72`, `stat73`, ..., `stat99`. That makes it easy to loop over the variables.

`first` is a view onto variable `firstyear`. Notice the marked line in the midst of the `for` loops: `first[i] = j+71`. When we change `first`, we change `firstyear`.

About the Author

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