Title

stata.com

putmata — Put Stata variables into Mata and vice versa

Description

`putmata` exports the contents of Stata variables to Mata vectors and matrices.

`getmata` imports the contents of Mata vectors and matrices to Stata variables.

`putmata` and `getmata` are useful for creating solutions to problems more easily solved in Mata. The commands are also useful in teaching.

Quick start

Create a Mata vector for each Stata variable in memory

```
putmata *
```

Same as above, but create a vector only for nonmissing values of `idvar`, `v1`, and `v2`

```
putmata idvar v1 v2, omitmissing
```

Place variables `v1` and `v2` into column vectors `x1` and `x2`

```
putmata idvar x1=v1 x2=v2
```

Create Mata matrix `X` from `v1` and `v2`

```
putmata X=(v1 v2)
```

Create Stata variables `newv1` and `newv2` from Mata matrix `X`

```
getmata (newv1 newv2)=X
```

Replace `v1` and `v2` with columns from Mata matrix `X`

```
getmata (v1 v2)=X, replace
```

Same as above, and match observations using `idvar` Mata vector

```
getmata (v1 v2)=X, replace id(idvar)
```

Syntax

```
putmata putlist [if] [in] [, putmata_options]

getmata getlist [, getmata_options]
```

putmata_options	Description
omitmissing	omit observations with missing values
view	create vectors and matrices as views, not as copies
replace	replace existing Mata vectors and matrices

A *putlist* can be as simple as a list of Stata variable names. See below for details.

getmata_options	Description
double	create Stata variables as doubles
update	update existing Stata variables
replace	replace existing Stata variables
id(*name*)	match observations with rows based on equal values of variable *name* and matrix *name*; id(*varname*=*vecname*) is also allowed
force	allow nonconformable matrices; usually, id() is preferable

A *getlist* can be as simple as a list of Mata vector names. See below for details.

collect is allowed with putmata and getmata; see **[U] 11.1.10 Prefix commands**.

Definition of *putlist* for use with putmata:

A *putlist* is one or more of any of the following:

```
*
varname
varlist
vecname=varname
matname=(varlist)
matname=([varlist] # [varlist] [...])
```

Example: putmata *
Creates a vector in Mata for each of the Stata variables in memory. Vectors contain the same data as Stata variables. Vectors have the same names as the corresponding variables.

Example: putmata mpg weight displ
Creates a vector in Mata for each variable specified. Vectors have the same names as the corresponding variables. In this example, displ is an abbreviation for the variable displacement; thus the vector will also be named displacement.

Example: putmata mileage=mpg pounds=weight
Creates a vector for each variable specified. Vector names differ from the corresponding variable names. In this example, vectors will be named mileage and pounds.

Example: putmata y=mpg X=(weight displ)
Creates $N \times 1$ Mata vector $\mathbf{y}$ equal to Stata variable mpg, and creates $N \times 2$ Mata matrix $\mathbf{X}$ containing the values of Stata variables weight and displacement.

Example: `putmata y=mpg X=(weight displ 1)`
Creates $N \times 1$ Mata vector `y` containing `mpg`, and creates $N \times 3$ Mata matrix `X` containing `weight`, `displacement`, and a column of 1s. After typing this example, you could enter Mata and type `invsym(X'X)*X'y` to obtain the regression coefficients.

Syntactical elements may be combined. It is valid to type

```
. putmata mpg foreign X=(weight displ) Z=(foreign 1)
```

No matter how you specify the *putlist*, you will need to specify the `replace` option if some or all vectors already exist in Mata:

```
. putmata mpg foreign X=(weight displ) Z=(foreign 1), replace
```

Definition of *getlist* for use with `getmata`:

A *getlist* is one or more of any of the following:

vecname
varname`=`*vecname*
`(`*varname varname ... varname*`)=`*matname*
`(`*varname*`*)=`*matname*

Example: `getmata x1 x2`
Creates a Stata variable for each Mata vector specified. Variables will have the same names as the corresponding vectors. Names may not be abbreviated.

Example: `getmata myvar1=x1 myvar2=x2`
Creates a Stata variable for each Mata vector specified. Variable names will differ from the corresponding vector names.

Example: `getmata (firstvar secondvar)=X`
Creates one Stata variable corresponding to each column of the Mata matrix specified. In this case, the matrix has two columns, and corresponding variables will be named `firstvar` and `secondvar`. If the matrix had three columns, then three variable names would need to be specified.

Example: `getmata (myvar*)=X`
Creates one Stata variable corresponding to each column of the Mata matrix specified. Variables will be named `myvar1`, `myvar2`, etc. The matrix may have any number of columns, even zero!

Syntactical elements may be combined. It is valid to type

```
. getmata r1 r2 final=r3 (rplus*)=X
```

No matter how you specify the *getlist*, you will need to specify the `replace` or `update` option if some or all variables already exist in Stata:

```
. getmata r1 r2 final=r3 (rplus*)=X, replace
```

Options for putmata

`omitmissing` specifies that observations containing a missing value in any of the numeric variables specified be omitted from the vectors and matrices created in Mata. In

```
. putmata y=mpg X=(weight displ 1), omitmissing
```

rows would be omitted from `y` and `X` in which the corresponding observation contained missing in any of `mpg`, `weight`, or `displ`. In this case, specifying `omitmissing` would be equivalent to typing

```
. putmata y=mpg X=(weight displ 1) if !missing(mpg) & !missing(weight) ///
      & !missing(displ)
```

All vectors and matrices created by a single `putmata` command will have the same number of rows (observations). That is true whether you specify `if`, `in`, or the `omitmissing` option.

`view` specifies that `putmata` create views rather than copies of the Stata data in the Mata vectors and matrices. Views require less memory than copies and offer the advantage (and disadvantage) that changes in the Stata data are immediately reflected in the Mata vectors and matrices, and vice versa.

If you specify numeric constants using the *matname*`=(...)` syntax, *matname* is created as a copy even if the `view` option is specified. Other vectors and matrices created by the command, however, would be views.

Use of the `view` option with `putmata` often obviates the need to use `getmata` to import results back into Stata.

Warning 1: Mata records views as "this vector is a view onto variable 3, observations 2 through 5 and 7". If you change the order of the variables, the order of the observations, or drop variables once the views are created, then the contents of the views will change.

Warning 2: When assigning values in Mata to view vectors, code

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

not `v = ...`.

To have changes reflected in the underlying Stata data, you must update the elements of the view `v`, not redefine it. To update all the elements of `v`, you literally code `v[.]`. In the matrix case, you code `X[.,.]`.

`replace` specifies that existing Mata vectors or matrices be replaced should that be necessary.

Options for getmata

`double` specifies that Stata numeric variables be created as `double`s. The default is that they be created as `float`s. Actually, variables start out as `float`s or `double`s, but then they are compressed (see [D] **compress**).

`update` and `replace` are alternatives. They have the same meaning unless the `id()` or `force` option is specified.

When `id()` or `force` is not specified, both `replace` and `update` specify that it is okay to replace the values in existing Stata variables. By default, vectors can be posted to new Stata variables only.

When `id()` or `force` is specified, `replace` and `update` allow posting of values of existing variables, just as usual. The options differ in how the posting is performed when the `id()` or `force` option causes only a subset of the observations of the variables to be updated. `update` specifies that the remaining values be left as they are. `replace` specifies that the remaining values be set to missing, just as if the existing variable(s) were being created for the first time.

`id(`*name*`)` and `id(`*varname*`=`*vecname*`)` specify how the rows in the Mata vectors and matrices match the observations in the Stata data. Observation i matches row j if variable *name*$[\,i\,]$ equals vector *name*$[\,j\,]$, or in the second syntax, if *varname*$[\,i\,] =$ *vecname*$[\,j\,]$. The ID variable (vector) must contain values that uniquely identify the observations (rows). Only in observations that contain matching values will the variable be modified. Values in observations that have no match will not

be modified or will be set to missing, as appropriate; values in the ID vector that have no match will be ignored.

Example: You wish to run a regression of `y` on `x1` and `x2` on the males in the data and use that result to obtain the fitted values for the males. Stata already has commands that will do this, namely, `regress y x1 x2 if male` followed by `predict yhat if male`. For instructional purposes, let's say you wish to do this in Mata. You type

```
. putmata  myid y  X=(x1 x2 1) if male
. mata
: b = invsym(X'X)*X'y
: yhat = X*b
: end
. getmata yhat, id(myid)
```

The new Stata variable `yhat` will contain the predicted values for males and missing values for the females. If the `yhat` variable already existed, you would type

```
. getmata yhat, id(myid) replace
```

or

```
. getmata yhat, id(myid) update
```

The `replace` option would set the female observations to missing. The `update` option would leave the female observations unchanged.

If you do not have an identification variable, create one first by typing `generate myid = _n`.

`force` specifies that it is okay to post vectors and matrices with fewer or with more rows than the number of observations in the data. The `force` option is an alternative to `id()`, and usually, `id()` is the appropriate choice.

If you specify `force` and if there are fewer rows in the vectors and matrices than observations in the data, new variables will be padded with missing values. If there are more rows than observations, observations will be added to the data and previously existing variables will be padded with missing values.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Use of putmata

In this example, we will use Mata to make a calculation and report the result, but we will not post results back to Stata. We will use `putmata` but not `getmata`.

Consider solving for $\mathbf{b}$ the set of linear equations

$$\mathbf{y} = \mathbf{X}\mathbf{b} \tag{1}$$

where $\mathbf{y}$: $N \times 1$, $\mathbf{X}$: $N \times k$, and $\mathbf{b}$: $k \times 1$. If $N = k$, then $\mathbf{y} = \mathbf{X}\mathbf{b}$ amounts to solving k equations for k unknowns, and the solution is

$$\mathbf{b} = \mathbf{X}^{-1}\mathbf{y} \tag{2}$$

That solution is obtained by premultiplying both sides of (1) by $\mathbf{X}^{-1}$.

When $N > k$, (2) can be used to obtain least-square results if matrix inversion is appropriately defined. Assume that you wish to demonstrate this when matrix inversion is defined as the Moore–Penrose generalized inverse for nonsquare matrices. The demonstration can be obtained by typing

```
. sysuse auto, clear
. regress mpg weight displacement
. putmata y=mpg X=(weight displacement 1)
. mata
: pinv(X)*y
: end
. _
```

The Mata expression `pinv(X)*y` will display a 3×1 column vector. The elements of the vector will equal the coefficients reported by `regress mpg weight displacement`.

For your information, the Moore–Penrose inverse of rectangular matrix $\mathbf{X}$: $N \times k$ is a $k \times N$ rectangular matrix. Among other properties, `pinv(X)*X` $= \mathbf{I}$, where $\mathbf{I}$ is the $k \times k$ identity matrix. You can demonstrate that using Mata, too:

```
. mata: pinv(X)*X
```

Use of putmata and getmata

In this example, we will use Mata to calculate a result that we wish to post back to Stata. We will use both `putmata` and `getmata`.

Some problems are more easily solved in Mata than in Stata. For instance, say that you need to create new Stata variable `D` from existing variable `C`, defined as

$$\texttt{D}[i] = \text{sum}(\texttt{C}[j] - \texttt{C}[i]) \text{ for all } \texttt{C}[j] > \texttt{C}[i]$$

where i and j index observations.

This problem can be solved in Stata, but the solution is elusive to most people. The solution is more natural in Mata because the Mata solution corresponds almost letter for letter with the mathematical statement of the problem. If `C` and `D` were Mata vectors rather than Stata variables, the solution would be

```
D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
        for (j=1; j<=rows(C); j++) {
                if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
        }
}
```

The most difficult part of this solution to understand is the first line, `D = J(rows(C), 1, 0)`, and that is because you may not be familiar with Mata's `J()` function. `D = J(rows(C), 1, 0)` creates a `rows(C)` $\times$ 1 column vector of 0s. The arguments of `J()` are in just that order.

`C` and `D` are not vectors in Mata, or at least they are not yet. Using `getmata`, we can create vector `C` from variable `C` and run our Mata solution. Then using `putmata`, we can post Mata vector `D` back to new Stata variable `D`. The solution includes these three steps, also shown in the do-file below:

(1) In Stata, use `putmata` to create vector `C` in Mata equal to variable `C` in Stata: `putmata C`.

(2) Use Mata to solve the problem, creating new Mata vector `D`.

(3) In Stata again, use `getmata` to create new variable `D` equal to Mata vector `D`.

Because of the typing involved in the solution, we would package the code in a do-file:

begin myfile.do

```
use mydata, clear
putmata C                                                    (1)
mata:                                                        (2)
D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
        for (j=1; j<=rows(C); j++) {
                if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
        }
}
end
getmata D                                                    (3)
save mydata, replace
```

end myfile.do

With `myfile.do` now in place, in Stata we would type

```
. do myfile
```

Notes:

(1) Our program might be better if we changed `putmata C` to read `putmata C, replace` and if we changed `getmata D` to read `getmata D, replace`. As things are right now, typing `do myfile` works, but if we were then to run it a second time, it would not work. Stata would encounter the `putmata` command and issue an error that matrix `C` already exists. Even if Stata got through that, it would encounter the `getmata` command and issue an error that variable `D` already exists. Perhaps that is an advantage. You cannot run `myfile.do` again without dropping matrix `C` and variable `D`. If you consider that a disadvantage, however, include the `replace` option.

(2) In our solution, we entered Mata by typing `mata:`, which is to say, `mata` with a colon. Interactively, we usually enter Mata by just typing `mata`. The colon affects how Mata treats errors. When working interactively, we want Mata to note errors but then to continue running so we can correct ourselves. In do-files, we want Mata to note the error and stop. That is the difference between `mata` without the colon and `mata` with the colon. Remember to use `mata:` when writing do-files.

(3) Rather than specify the `replace` option, you could modify the do-file to drop any preexisting Mata vector `C` and any preexisting variable `D`. To drop vector `C`, in Mata you can type `mata drop C`, or in Stata, you can type `mata: mata drop C`. To drop variable `D`, in Stata you can type `drop D`. You must worry that the variables do not exist, so in your do-file, you would code

```
capture mata: mata drop C
capture drop D
```

Rather than dropping vector `C`, you might prefer just to clear Mata:

```
clear mata
```

Using putmata and getmata on subsets of observations

putmata can be used to create Mata vectors that contain a subset of the observations in the Stata data, and getmata can be used to fetch such vectors back into Stata. Thus you can work with only the males or only outcomes in which failures are observed, and so on. Below we work with only the observations in which C does not contain missing values.

In the create-variable-D-from-C example above, we assumed that there were no missing values in C, or at least we did not consider the issue. It turns out that our code produces several missing values in the presence of just one missing value in C. Perhaps, if there are missing values, we want to exclude them from our calculation. We could complicate our Mata code to handle that. We could modify our Mata code to read

```
use mydata, clear
putmata C

D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
        if (C[i]>=.) D[i] = .                    //  new
        else for (j=1; j<=rows(C); j++) {
                if (C[j]<.) {                    //  new
                        if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
                }
        }
}
end

getmata D
save mydata, replace
```

Easier, however, is simply to restrict Mata vector C to the nonmissing elements of Stata variable C, which we could do by replacing putmata C with

```
putmata C if !missing(C)
```

or, equivalently,

```
putmata C, omitmissing
```

Whichever way we coded it, if the data contained 100 observations and variable C contained 82 nonmissing values, new Mata vector C would contain 82 rows rather than 100. The observations corresponding to missing(C) would be omitted from the vector, and that means we could run our original Mata solution without modification.

There is, however, an issue. At the end of our code when we post the Mata solution vector D to Stata variable D—getmata D—we will need to specify which of the 100 observations are to receive the 82 results stored in the vector. getmata has an option to handle this situation—id(*varname*), where *varname* is the name of an identification variable.

An identification variable is a variable that takes on different values for each observation in the data. The values could be 1, 2, ..., 100; or they could be 1.25, −2, ..., 16.5; or they could be Nick, Bill, ..., Mary. The values can be numeric or string, and they need not be in order. All that is important is that the variable contain a unique (different) value in each observation. Possibly, the data already contain such a variable. If not, you can create one by typing

```
generate fid = _n
```

When we use putmata to create vector C, we will need simultaneously to create vector fid containing the selected values of variable fid, which we can do by adding fid to the *putlist*:

```
putmata fid C if !missing(C)
```

The above command creates two vectors in Mata: `fid` and `C`. When we post the resulting vector `D` back to Stata, we will specify the `id(fid)` option to indicate into which observations `getmata` is to post the results:

```
getmata D, id(fid)
```

The `id(fid)` option is taken to mean that there exists a variable named `fid` and a vector named `fid`. It is by comparing the values in each that `getmata` determines how the rows of the vectors correspond to the observations of the data.

The entire solution is

begin myfile.do

```
use mydata, clear
putmata fid C if !missing(C)         // new: we put fid & add if !missing(C)
mata:
D = J(rows(C), 1, 0)
for (i=1; i<=rows(C); i++) {
        for (j=1; j<=rows(C); j++) {
                if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
        }
}
end
getmata D, id(fid)           // new: we add option id(fid)
save mydata, replace
```

end myfile.do

The above code will run on data with or without missing values. New variable `D` will be missing in observations where `C` is missing, but `D` will otherwise contain nonmissing values.

Using views

When you type or code `putmata C`, vector `C` is created as a copy of the Stata data. The variable and the vector are separate things. An alternative is to make the Mata vector a view onto the Stata variable. By that, we mean that both the variable and the vector share the same recording of the values. Views save memory but are slightly less efficient in terms of execution time. Views have other advantages and disadvantages, too.

For instance, if you type `putmata mpg` and then, in Mata, type `mpg[1]=20`, you will change not only the Mata vector but also the Stata data! Or if, after typing `putmata mpg`, you typed `replace mpg = 20 in 1`, that would modify both the data and the Mata vector! This is an advantage if you are fixing real errors and a disadvantage if you intend to do something else.

If in the middle of your Mata session where you are working with views you take a break and return to Stata, it is important that you do not modify the Stata data in certain ways. Rather than recording copies of the data, views record notes about the mapping. A view might record that this Mata vector corresponds to variable 3, observations 2 through 20 and 39. If you change the sort order of the data, the view will still be working with observations 2 through 20 and 39 even though those physical observations now contain different data. If you drop the first or second variable, the view will still be working with the third variable even though that will now be a different variable!

The memory savings offered by views are considerable, at least when working with large datasets. Say that you have a dataset containing 200 variables and 1,000,000 observations. Your data might be 1 GB in size. Even so, typing `putmata *, view`, and thus creating 200 vectors each with 1,000,000 rows, would consume only a few dozen kilobytes of memory.

All the examples shown above work equally well with copies or views. We have been working with copies, but in the previous example, where we coded

```
putmata fid C if !missing(C)
```

we could switch to working with views by coding

```
putmata fid C if !missing(C), view
```

With that one change, our code would still work and it would use less memory.

With that one change, we would still not be working with views everywhere we could, however. Vector D—the vector we create in Mata and then post back to Stata—would still be a regular vector. We can save additional memory by making D a view, too. Before we do that, let us warn you that we do not recommend doing this unless the memory savings is vitally important. The result, when complete, will be elegant and memory efficient, but the extra memory savings is seldom worth the debugging effort.

No extra changes are required to your code when the vectors you make into views contain values that are not modified in the code. Vector C is such a vector. We use the values stored in C, but we do not change them. Vector D, on the other hand, is a vector in which we change values. It is usually easier if you do not convert such vectors into views.

With that proviso, we are going to make D into a view, too, and in the process, we will drop the use of fid altogether:

begin myfile.do

```
use mydata, clear
generate D = .                          // new
putmata C D if !missing(C), view        // changed
mata:
D[.] = J(rows(C), 1, 0)                 // changed
for (i=1; i<=rows(C); i++) {
             for (j=1; j<=rows(C); j++) {
                     if (C[j]>C[i]) D[i] = D[i] + (C[j] - C[i])
             }
}
end
                                        // we drop the getmata
save mydata, replace
```

end myfile.do

In this solution, we create new Stata variable D at the outset, and then we modify the putmata command to create view vectors for both C and D. Our code, which stores results in vector D, now simultaneously posts to variable D when we store results in vector D, so we can omit the getmata D at the end because results are already posted! Moreover, we no longer have to concern ourselves with matching observations to rows via fid. Rows of D now automatically align themselves with the selected observations in variable D by the mere fact of D being a view.

The beginning of our Mata code has an important change, however. We change

```
D = J(rows(C), 1, 0)
```

to

```
D[.] = J(rows(C), 1, 0)
```

That change is very important. What we coded previously created vector D. What we now code changes the values stored in existing vector D. If we left what we coded previously, Mata would discard the view currently stored in D and create a new D—a regular Mata vector unconnected to Stata—containing 0s.

Constructing do-files

putmata and getmata can be used interactively, but if you have much Mata code between the put and the get, you will be better off using a do-file because do-files can be easily edited when they have a mistake in them. We recommend the following outline for such do-files:

begin outline.do

```
version 18.0                          (1)
mata clear                            (2)
// Stata code for setup goes here     (3)
putmata ...                           (4)
mata:
// Mata code goes here                (5)
end
getmata                               (6)
mata clear                            (7)
```

end outline.do

Notes on do-file steps:

(1) A do-file should always start with a version statement; it ensures that the do-file continues to work in the years to come as new versions of Stata are released. See [P] **version**.

(2) The do-file should not depend on Mata having certain vectors, matrices, or programs already loaded and set up because if you attempt to run the do-file again later, what you assumed may not be true. A do-file should be self-contained. To ensure that is true the first time we write and run the do-file and to ensure on subsequent runs that nothing lying around in Mata gets in our way, we clear Mata.

(3) You may need to sort your data, create extra variables that your do-file will use, or drop variables that you are assuming do not already exist. In the last iteration of myfile.do, we needed to generate D = ., and it would not have been a bad idea to capture drop D before we did that. Our example did not depend on the sort order of the data, but if it had, we would have included the sort even if we were certain that the data would already be in the right order.

(4) Put the putmata command here. If putmata includes the omitmissing option, then put everything you need to put in a single putmata command. Otherwise, you can use multiple putmata commands if you find that more convenient. If you use multiple putmata commands, be sure to include the same if *expression* and in *range* qualifiers on each one.

(5) The Mata code goes here. Note that we type mata: (mata with a colon) to enter Mata. mata: ensures that errors stop Mata and thus our do-file.

(6) The getmata command goes here if you need it. Be sure to include getmata's id(*name*) or id(*vecname*=*varname*) option if, on the putmata command in step 4, you included the if *expression* qualifier or the in *range* qualifier or the omitmissing option. If you include id(), be sure you included the ID variable in the putmata command in step 4.

(7) We conclude by clearing Mata again to avoid leaving memory allocated needlessly and to avoid causing problems for poorly written do-files that we might subsequently run.

`putmata` and `getmata` are designed to work interactively and in do-files. The commands are not designed to work with ado-files. An ado-file is something like a do-file, but it defines a program that implements a new command of Stata, and well-written ado-files do not use globals such as the global vectors and matrices that `putmata` creates. Ado-files use local variables. Ado-file programmers should use the Mata functions `st_data()` and `st_view()` (see [M-5] **st_data( )** and [M-5] **st_view( )**) to create vectors and matrices, and if necessary, use `st_store()` (see [M-5] **st_store( )**) to post the contents of those vectors and matrices back to Stata.

Stored results

`putmata` stores the following in `r()`:

Scalars

`r(N)`	number of rows in created vectors and matrices
`r(K_views)`	number of vectors and matrices created as views
`r(K_copies)`	number of vectors and matrices created as copies

The total number of vectors and matrices created is `r(K_views) + r(K_copies)`.

`r(N)=.` if `r(K_views) + r(K_copies) = 0`. `r(N)=0` means that zero-observation vectors and matrices were created, which is to say, vectors and matrices dimensioned 0×1 and $0 \times k$.

`getmata` stores the following in `r()`:

Scalars

`r(K_new)`	number of new variables created
`r(K_existing)`	number of existing variables modified

The total number of variables modified is `r(K_new) + r(K_existing)`.

Reference

Gould, W. W. 2010. Mata Matters: Stata in Mata. *Stata Journal* 10: 125–142.

Also see

[M-4] **Stata** — Stata interface functions

[M-5] **st_data( )** — Load copy of current Stata dataset

[M-5] **st_store( )** — Modify values stored in current Stata dataset

[M-5] **st_view( )** — Make matrix that is a view onto current Stata dataset