Mata in Stata

Christopher F Baum

Faculty Micro Resource Center
Boston College

January 2007

Mata: Stata's matrix programming language

As of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with all of the capabilities of MATLAB, Ox or GAUSS. Mata can be used interactively, or Mata functions can be developed to be called from Stata. A large library of mathematical and matrix functions is provided in Mata, including equation solvers, decompositions, eigensystem routines and probability density functions. Mata functions can access Stata's variables and can work with virtual matrices (*views*) of a subset of the data in memory. Mata also supports file input/output.

Mata code is automatically compiled into bytecode, like Java, and can be stored in object form or included in-line in a Stata do-file or ado-file. Mata code runs many times faster than the interpreted ado-file language, providing significant speed enhancements to many computationally burdensome tasks.

Mata: Stata's matrix programming language

As of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with all of the capabilities of MATLAB, Ox or GAUSS. Mata can be used interactively, or Mata functions can be developed to be called from Stata. A large library of mathematical and matrix functions is provided in Mata, including equation solvers, decompositions, eigensystem routines and probability density functions. Mata functions can access Stata's variables and can work with virtual matrices (*views*) of a subset of the data in memory. Mata also supports file input/output.

Mata code is automatically compiled into bytecode, like Java, and can be stored in object form or included in-line in a Stata do-file or ado-file. Mata code runs many times faster than the interpreted ado-file language, providing significant speed enhancements to many computationally burdensome tasks.

Mata circumvents the limitations of Stata's traditional matrix commands. Stata matrices must obey the maximum *matsize:* 800 rows or columns in Intercooled Stata. Thus, code relying on Stata matrices is fragile. Stata's matrix language does contain commands such as `matrix accum` which can build a cross-product matrix from variables of any length, but for many applications the limitation of `matsize` is binding.

Even in Stata/SE with the possibility of a much larger `matsize`, Stata's matrices have another drawback. Large matrices consume large amounts of memory, and an operation that converts Stata variables into a matrix or vice versa will require twice the memory needed for that set of variables.

Last but surely not least, ado-file code written in the matrix language with explicit subscript references is slow.

Mata circumvents the limitations of Stata's traditional matrix commands. Stata matrices must obey the maximum *matsize:* 800 rows or columns in Intercooled Stata. Thus, code relying on Stata matrices is fragile. Stata's matrix language does contain commands such as `matrix accum` which can build a cross-product matrix from variables of any length, but for many applications the limitation of `matsize` is binding.

Even in Stata/SE with the possibility of a much larger `matsize`, Stata's matrices have another drawback. Large matrices consume large amounts of memory, and an operation that converts Stata variables into a matrix or vice versa will require twice the memory needed for that set of variables.

Last but surely not least, ado-file code written in the matrix language with explicit subscript references is slow.

Mata circumvents the limitations of Stata's traditional matrix commands. Stata matrices must obey the maximum *matsize:* 800 rows or columns in Intercooled Stata. Thus, code relying on Stata matrices is fragile. Stata's matrix language does contain commands such as `matrix accum` which can build a cross-product matrix from variables of any length, but for many applications the limitation of `matsize` is binding.

Even in Stata/SE with the possibility of a much larger `matsize`, Stata's matrices have another drawback. Large matrices consume large amounts of memory, and an operation that converts Stata variables into a matrix or vice versa will require twice the memory needed for that set of variables.

Last but surely not least, ado-file code written in the matrix language with explicit subscript references is slow.

The Mata programming language can sidestep these memory issues by creating matrices with contents that refer directly to Stata variables—no matter how many variables and observations may be referenced. These virtual matrices, or *views*, have minimal overhead in terms of memory consumption irregardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements. A single matrix may not mix those elements, but it may be declared generically to hold either type of data. This implies that Mata can be used productively in a list processing environment as well as in a numeric context. Indeed, a command such as Bill Gould's `adoupdate` is written almost completely in Mata.

The Mata programming language can sidestep these memory issues by creating matrices with contents that refer directly to Stata variables—no matter how many variables and observations may be referenced. These virtual matrices, or *views*, have minimal overhead in terms of memory consumption irregardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements. A single matrix may not mix those elements, but it may be declared generically to hold either type of data. This implies that Mata can be used productively in a list processing environment as well as in a numeric context. Indeed, a command such as Bill Gould's `adoupdate` is written almost completely in Mata.

Mata can be used very productively—like other matrix programming languages—in an interactive environment. Just entering `mata` at the Stata command dot-prompt puts you into the Mata environment, with the colon prompt. To exit Mata and return to Stata, enter `end`. However, the contents of your Mata environment will still exist for the remainder of your interactive Stata session. You may enter Mata again and take up where you left off.

In this presentation, we will not focus on interactive Mata use, but rather on the way in which Mata can be used as a valuable adjunct to Stata's ado-file language. Its advantages arise in two contexts: where computations may be done more efficiently in Mata due to its compiled bytecode, and where the algorithm you wish to implement already exists in matrix-language form. In many cases both of those rationales will make Mata an ideal solution to your programming problem.

Mata can be used very productively—like other matrix programming languages—in an interactive environment. Just entering `mata` at the Stata command dot-prompt puts you into the Mata environment, with the colon prompt. To exit Mata and return to Stata, enter `end`. However, the contents of your Mata environment will still exist for the remainder of your interactive Stata session. You may enter Mata again and take up where you left off.

In this presentation, we will not focus on interactive Mata use, but rather on the way in which Mata can be used as a valuable adjunct to Stata's ado-file language. Its advantages arise in two contexts: where computations may be done more efficiently in Mata due to its compiled bytecode, and where the algorithm you wish to implement already exists in matrix-language form. In many cases both of those rationales will make Mata an ideal solution to your programming problem.

In a pure matrix programming language, you must handle all of the housekeeping details involved with data organization, transformation and selection. In contrast, if you write an ado-file that calls one or more Mata functions, the ado-file will handle those housekeeping details with the convenience features of the `syntax` and `marksample` statements of the regular ado-file language. When the housekeeping chores are completed, the resulting variables can be passed on to Mata for processing.

Mata can access Stata variables, local and global macros, scalars and matrices, and modify the contents of those objects as needed. If Mata's view matrices are used, alterations to the matrix within Mata modifies the Stata variables that comprise the view.

In a pure matrix programming language, you must handle all of the housekeeping details involved with data organization, transformation and selection. In contrast, if you write an ado-file that calls one or more Mata functions, the ado-file will handle those housekeeping details with the convenience features of the `syntax` and `marksample` statements of the regular ado-file language. When the housekeeping chores are completed, the resulting variables can be passed on to Mata for processing.

Mata can access Stata variables, local and global macros, scalars and matrices, and modify the contents of those objects as needed. If Mata's view matrices are used, alterations to the matrix within Mata modifies the Stata variables that comprise the view.

To understand Mata syntax, we present several of its operators. The comma is the *column-join* operator, so

```
a = ( 1, 2, 3 )
```

creates a three-element row vector. The backslash is the *row-join* operator, so

```
b = ( 4 \ 5 \ 6 )
```

creates a three-element column vector, while

```
c = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
```

creates a 3 $\times$ 3 matrix.

The prime (or apostrophe) is the transpose operator, so

```
d = ( 1 \ 2 \ 3 )'
```

is a row vector. The comma and backslash operators can be used on vectors and matrices as well as scalars, so

```
e = a, b'
```

will produce a six-element row vector, and

```
f = a' \ b
```

a six-element column vector. Matrix elements can be real or complex, so `2 - 3 i` refers to a complex number $2 - 3 \times \sqrt{-1}$.

The standard algebraic operators plus, minus and multiply (*) work on scalars or matrices:

```
g = a'  + b
h = a * b
j = b * a
```

In this example `h` will be the dot product of vectors `a, b` while `j` is their outer product.

Stata's algebraic operators (including the slash for division) also can be used in element-by-element computations when preceded by a colon:

```
k = a' :* b
```

will produce the three-element column vector, with elements as the product of the respective elements.

The standard algebraic operators plus, minus and multiply (*) work on scalars or matrices:

```
g = a'  + b
h = a * b
j = b * a
```

In this example `h` will be the dot product of vectors `a, b` while `j` is their outer product.

Stata's algebraic operators (including the slash for division) also can be used in element-by-element computations when preceded by a colon:

```
k = a' :* b
```

will produce the three-element column vector, with elements as the product of the respective elements.

Mata's colon operator is very powerful, in that it will work on nonconformable objects. For example:

```
a = ( 1, 2, 3 )
c = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
m = a :+ c
n = c :/ a
```

adds the row vector `a` to each row of `c` to form `m`, and divides each row of `c` by the corresponding elements of `a` to form `n`.

Stata's scalar functions will also operate on elements of matrices:

```
d = sqrt(c)
```

will take the element-by-element square root, returning missing values where appropriate.

Mata's colon operator is very powerful, in that it will work on nonconformable objects. For example:

```
a = ( 1, 2, 3 )
c = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
m = a :+ c
n = c :/ a
```

adds the row vector `a` to each row of `c` to form `m`, and divides each row of `c` by the corresponding elements of `a` to form `n`.

Stata's scalar functions will also operate on elements of matrices:

```
d = sqrt(c)
```

will take the element-by-element square root, returning missing values where appropriate.

As in Stata, the equality logical operators are `a == b` and `a != b`. They will work whether or not `a` and `b` are conformable or even of the same type: `a` could be a vector and `b` a matrix. They return 0 or 1.

Unary not `!` returns 1 if a scalar equals zero, 0 otherwise, and may be applied in a vector or matrix context, returning a vector or matrix of 0, 1.

The remaining logical comparison operators (`>`, `>=`, `<`, `<=`) can only be used on objects that are conformable and of the same general type (numeric or string). They return 0 or 1.

The logical and (`&`) and or (`|`) operators, as in Stata, can only be applied to real scalars.

As in Stata, the equality logical operators are `a == b` and `a != b`. They will work whether or not `a` and `b` are conformable or even of the same type: `a` could be a vector and `b` a matrix. They return 0 or 1.

Unary not `!` returns 1 if a scalar equals zero, 0 otherwise, and may be applied in a vector or matrix context, returning a vector or matrix of 0, 1.

The remaining logical comparison operators (`>`, `>=`, `<`, `<=`) can only be used on objects that are conformable and of the same general type (numeric or string). They return 0 or 1.

The logical and (`&`) and or (`|`) operators, as in Stata, can only be applied to real scalars.

As in Stata, the equality logical operators are `a == b` and `a != b`. They will work whether or not `a` and `b` are conformable or even of the same type: `a` could be a vector and `b` a matrix. They return 0 or 1.

Unary not `!` returns 1 if a scalar equals zero, 0 otherwise, and may be applied in a vector or matrix context, returning a vector or matrix of 0, 1.

The remaining logical comparison operators (`>`, `>=`, `<`, `<=`) can only be used on objects that are conformable and of the same general type (numeric or string). They return 0 or 1.

The logical and (`&`) and or (`|`) operators, as in Stata, can only be applied to real scalars.

As in Stata, the equality logical operators are `a == b` and `a != b`. They will work whether or not `a` and `b` are conformable or even of the same type: `a` could be a vector and `b` a matrix. They return 0 or 1.

Unary not `!` returns 1 if a scalar equals zero, 0 otherwise, and may be applied in a vector or matrix context, returning a vector or matrix of 0, 1.

The remaining logical comparison operators (`>`, `>=`, `<`, `<=`) can only be used on objects that are conformable and of the same general type (numeric or string). They return 0 or 1.

The logical and (`&`) and or (`|`) operators, as in Stata, can only be applied to real scalars.

Subscripts in Mata utilize square brackets, and may appear on either the left or right of an algebraic expression. There are two forms: *list* subscripts and *range* subscripts.

With list subscripts, you can reference a single element of an array as `x[i,j]`. But `i` or `j` can also be a vector: `x[i,jvec]`, where `jvec=(4,6,8)` will reference row `i` and those three columns of `x`. Missing values (dots) will reference all rows or columns, so `x[i,.]` or `x[i,]` extracts row `i`, and `x[.,.]` or `x[,]` references the whole matrix.

You may also use *range operators* to avoid listing each consecutive element: `x[(1..4),.]` and `x[(1::4),.]` will both reference the first four rows of `x`. The double-dot range creates a row vector, while the double-colon range creates a column vector. Either may be used in a subscript expression. Ranges may also decrement, so `x[(3::1),.]` returns those rows in reverse order.

Subscripts in Mata utilize square brackets, and may appear on either the left or right of an algebraic expression. There are two forms: *list* subscripts and *range* subscripts.

With list subscripts, you can reference a single element of an array as `x[i,j]`. But `i` or `j` can also be a vector: `x[i,jvec]`, where `jvec=(4,6,8)` will reference row `i` and those three columns of `x`. Missing values (dots) will reference all rows or columns, so `x[i,.]` or `x[i,]` extracts row `i`, and `x[.,.]` or `x[,]` references the whole matrix.

You may also use *range operators* to avoid listing each consecutive element: `x[(1..4),.]` and `x[(1::4),.]` will both reference the first four rows of `x`. The double-dot range creates a row vector, while the double-colon range creates a column vector. Either may be used in a subscript expression. Ranges may also decrement, so `x[(3::1),.]` returns those rows in reverse order.

Subscripts in Mata utilize square brackets, and may appear on either the left or right of an algebraic expression. There are two forms: *list* subscripts and *range* subscripts.

With list subscripts, you can reference a single element of an array as `x[i,j]`. But `i` or `j` can also be a vector: `x[i,jvec]`, where `jvec=(4,6,8)` will reference row `i` and those three columns of `x`. Missing values (dots) will reference all rows or columns, so `x[i,.]` or `x[i,]` extracts row `i`, and `x[.,.]` or `x[,]` references the whole matrix.

You may also use *range operators* to avoid listing each consecutive element: `x[(1..4),.]` and `x[(1::4),.]` will both reference the first four rows of `x`. The double-dot range creates a row vector, while the double-colon range creates a column vector. Either may be used in a subscript expression. Ranges may also decrement, so `x[(3::1),.]` returns those rows in reverse order.

Range subscripts use the notation `[| |]`. They can reference single elements of matrices, but are not useful for that. More useful is the ability to say `x[| i,j \ m,n |]`, which creates a submatrix starting at `x[i,j]` and ending at `x[m,n]`. The arguments may be specified as missing (dot), so `x[| 1,2 \ 4,. |]` will specify the submatrix ending in the last column and `x[| 2,2 \ .,. |]` will discard the first row and column of `x`. They also may be used on the left hand side of an expression, or to extract a submatrix:
`v = invsym(xx)[| 2,2 \ .,. |]` will discard the first row and column of the inverse of `xx`.

You need not use range subscripts, as even the specification of a submatrix can be handled with list subscripts and range *operators*, but they are more convenient for submatrix extraction (and faster in terms of execution time).

Range subscripts use the notation `[| |]`. They can reference single elements of matrices, but are not useful for that. More useful is the ability to say `x[| i,j \ m,n |]`, which creates a submatrix starting at `x[i,j]` and ending at `x[m,n]`. The arguments may be specified as missing (dot), so `x[| 1,2 \ 4,. |]` will specify the submatrix ending in the last column and `x[| 2,2 \ .,. |]` will discard the first row and column of `x`. They also may be used on the left hand side of an expression, or to extract a submatrix:
`v = invsym(xx)[| 2,2 \ .,. |]` will discard the first row and column of the inverse of `xx`.

You need not use range subscripts, as even the specification of a submatrix can be handled with list subscripts and range *operators*, but they are more convenient for submatrix extraction (and faster in terms of execution time).

Several constructs support loops in Mata. As in any matrix language, explicit loops should not be used where matrix operations can be used. The most common loop construct resembles that of C:

```
for (exp1; exp2; exp3) {
    statements
}
```

where the three `exp`s define the lower limit, upper limit and increment of the loop. For instance:

```
for (i=1; i<=10; i++) {
    printf("i=%g \n", i)
}
```

If a single statement is to be executed, it may appear on the `for` statement.

You may also use `do`, which follows the syntax

```
do {
    statements
} while (exp)
```

which will execute the statements at least once.

Alternatively, you may use `while`:

```
while (exp) {
    statements
}
```

which could be used, for example, to loop until convergence.

You may also use `do`, which follows the syntax

```
do {
    statements
} while (exp)
```

which will execute the statements at least once.

Alternatively, you may use `while`:

```
while (exp) {
    statements
}
```

which could be used, for example, to loop until convergence.

To execute certain statements conditionally, you use `if, else`:

```
if (exp) statement

if (exp) statement1
else statement2

if (exp) {
    statements1
}
else if {
    statements2
}
else {
    statements3
}
```

You may also use the conditional `a ? b : c`, where `a` is a real scalar. If `a` evaluates to true (nonzero), the result is set to `b`, otherwise `c`. For instance,

```
if (k == 0)  dof = n-1
else         dof = n-k
```

can be written as

```
dof = ( k==0 ? n-1 : n-k )
```

The increment (++) and decrement (−−) operators can be used to manage counter variables. The operator `A # B` produces the Kronecker direct product of those objects.

You may also use the conditional `a ? b : c`, where `a` is a real scalar. If `a` evaluates to true (nonzero), the result is set to `b`, otherwise `c`. For instance,

```
if (k == 0)  dof = n-1
else         dof = n-k
```

can be written as

```
dof = ( k==0 ? n-1 : n-k )
```

The increment (++) and decrement (−−) operators can be used to manage counter variables. The operator `A # B` produces the Kronecker direct product of those objects.

For compatibility with old-style Fortran, there is a `goto` statement:

```
label: statement
    statements
    if (exp) goto label
}
```

Although such a construct can be rewritten in terms of `do`:

```
do {
    statements
    } while (exp)
```

The `goto` statement is more useful when there are long-range branches in a program being translated from old-style Fortran code.

We now consider a simple Mata function called from an ado-file. Imagine that we did not have an easy way of computing the sum of the elements of a Stata variable, and wanted to do so with Mata:

```
program varsum, rclass
    version 9.2
    syntax varname [if] [in]
    marksample touse
    mata: calcsum( "`varlist'", "`touse'" )
    display as txt " sum ( `varlist' ) = " ///
            as res r(sum)
    return scalar sum = r(sum)
end
```

This is the first part of the contents of `varsum.ado`. We define the Mata `calcsum` function next.

We then add the Mata function definition to `varsum.ado`:

```
version 9.2
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
```

Our `varsum` ado-code creates a Stata command, `varsum`, which requires the name of a single Stata variable. You may specify `if` or `in` conditions. The Mata function `calcsum` is called with two arguments: the name of the variable and the name of the `touse` temporary variable marking out valid observations. As we will see the Mata function returns its results in a scalar, `r(sum)`, which we print out and return to Stata.

The Mata code as shown is `strict`: all objects must be defined. The function is declared `void` as it does not return a result. A Mata function could return a single result to Mata, but we want the result back in Stata. The input arguments are declared as `string scalar` as they are variable names. We create a *view matrix*, colvector `x`, as the subset of *varname* for which `touse==1`. Mata's `colsum( )` function computes the sum of those elements, and `st_numscalar` returns it to Stata as `r(sum)`.

Our `varsum` ado-code creates a Stata command, `varsum`, which requires the name of a single Stata variable. You may specify `if` or `in` conditions. The Mata function `calcsum` is called with two arguments: the name of the variable and the name of the `touse` temporary variable marking out valid observations. As we will see the Mata function returns its results in a scalar, `r(sum)`, which we print out and return to Stata.

The Mata code as shown is `strict`: all objects must be defined. The function is declared `void` as it does not return a result. A Mata function could return a single result to Mata, but we want the result back in Stata. The input arguments are declared as `string scalar` as they are variable names. We create a *view matrix*, colvector `x`, as the subset of *varname* for which `touse==1`. Mata's `colsum( )` function computes the sum of those elements, and `st_numscalar` returns it to Stata as `r(sum)`.

This short example of Mata code uses two of the important *st_ functions*: the Mata functions that permit Mata to access any object (variable, local or global macro, scalar, matrix, label, etc.) in Stata. These functions allow those objects to be read, but also to be created (as is the scalar `r(sum)` in this example) or updated. This implies that Mata can both read Stata variables (as in the example) and modify their contents.

We consider a simple program that alters a set of Stata variables next.

```
program centervars, rclass
    version 9.2
    syntax varlist(numeric) [if] [in]
    marksample touse
    mata: centerv( "`varlist'", "`touse'" )
end
version 9.2
mata:
void centerv( string scalar varlist, ///
              string scalar touse)
{
  st_view(X=.,.,tokens(varlist),touse)
  X[,] = X :- mean(X)
}
end
```

The `centervars.ado` file contains a Stata command, `centervars`, that takes a list of numeric variables. That list is passed to the Mata function `centerv` along with `touse`, the temporary variable that marks out the desired observations. The Mata function `tokens( )` extracts the variable names from *varlist* and places them in a string rowvector, the form needed by `st_view` . The `st_view` function then creates a *view matrix*, `X`, containing those variables and the specified observations.

In this function, though, the view matrix allows us to both access the variables' contents, as stored in Mata matrix `X`, but also to *modify* those contents. The colon operator subtracts the vector of column means of `X` from the data. Using the `X[,]=` notation, the Stata variables themselves are modified. When the Mata function returns to Stata, the descriptive statistics of the variables in *varlist* will be alter

The `centervars.ado` file contains a Stata command, `centervars`, that takes a list of numeric variables. That list is passed to the Mata function `centerv` along with `touse`, the temporary variable that marks out the desired observations. The Mata function `tokens( )` extracts the variable names from *varlist* and places them in a string rowvector, the form needed by `st_view` . The `st_view` function then creates a *view matrix*, `X`, containing those variables and the specified observations.

In this function, though, the view matrix allows us to both access the variables' contents, as stored in Mata matrix `X`, but also to *modify* those contents. The colon operator subtracts the vector of column means of `X` from the data. Using the `X[,]=` notation, the Stata variables themselves are modified. When the Mata function returns to Stata, the descriptive statistics of the variables in *varlist* will be altered.

The `centervars` command is somewhat dangerous in that it alters the contents of existing variables without explicit mention (e.g., a required `replace` option). A better approach would be to allow the specification of a *prefix* such as `c_` to create a set of new variables, or a separate *newvarlist* of new variable names to store the modified variables.

But the function illustrates the power of Mata: rather than writing a loop in the ado-file language which operates on each variable, we may give a single command to transform the entire set of variables, irregardless of their number.

We now discuss the *st_ functions* and other sets of Mata functions more thoroughly.

In the previous examples we used `st_view` to access Stata variables from within Mata, and `st_numscalar` to define the contents of a Stata numeric scalar. These are two of a sizable number of *st_functions* that permit interchange of information between the Stata (`st`) and Mata environments.

First let us define the `st_view` function, as it is the most common method of accessing Stata variables. Unlike most Mata functions, it does not return a result. It takes three arguments: the name of the view matrix to be created, the observations (rows) that it is to contain, and the variables (columns). An optional fourth argument can specify `touse`: an indicator variable specifying whether each observation is to be included.

In the previous examples we used `st_view` to access Stata variables from within Mata, and `st_numscalar` to define the contents of a Stata numeric scalar. These are two of a sizable number of *st_functions* that permit interchange of information between the Stata (`st`) and Mata environments.

First let us define the `st_view` function, as it is the most common method of accessing Stata variables. Unlike most Mata functions, it does not return a result. It takes three arguments: the name of the view matrix to be created, the observations (rows) that it is to contain, and the variables (columns). An optional fourth argument can specify `touse`: an indicator variable specifying whether each observation is to be included.

Thus a Mata statement

```
st_view(Z=., .., .)
```

will create a view matrix of all observations and all variables in Stata's memory. The missing value (dot) specification indicates that all observations and all variables are included. The syntax `Z=.` specifies that the object is to be created as a void matrix, and then populated with contents. As `Z` is defined as a real matrix, columns associated with any string variables will contain all missing values. `st_sview` creates a view matrix of string variables.

If we want to specify a subset of variables, we must define a string vector containing their names (as the example in `centervars.ado` using the `tokens( )` function shows).

Thus a Mata statement

```
st_view(Z=., ., .)
```

will create a view matrix of all observations and all variables in Stata's memory. The missing value (dot) specification indicates that all observations and all variables are included. The syntax `Z=.` specifies that the object is to be created as a void matrix, and then populated with contents. As `Z` is defined as a real matrix, columns associated with any string variables will contain all missing values. `st_sview` creates a view matrix of string variables.

If we want to specify a subset of variables, we must define a string vector containing their names (as the example in `centervars.ado` using the `tokens( )` function shows).

As in `centervars.ado`, modifying the contents of a view matrix will alter the original variables. Those variables were defined in Stata, so altering their values will not change their data type. Although Stata's `generate` or `replace` commands will *promote* or *cast* a variable (for instance, from `int` to `real`) as needed, `centervars.ado` will return integer variables if applied to integer variables.

A good approach to this problem involves creating new variables of the appropriate data type in Stata and forming two view matrices within Mata: one that only accesses the original variables and a second that maps into the new variables. This will also ensure that the original variables are not altered by the Mata function. An example of this logic is contained in `hprescott.ado` (`findit hprescott`).

As in `centervars.ado`, modifying the contents of a view matrix will alter the original variables. Those variables were defined in Stata, so altering their values will not change their data type. Although Stata's `generate` or `replace` commands will *promote* or *cast* a variable (for instance, from `int` to `real`) as needed, `centervars.ado` will return integer variables if applied to integer variables.

A good approach to this problem involves creating new variables of the appropriate data type in Stata and forming two view matrices within Mata: one that only accesses the original variables and a second that maps into the new variables. This will also ensure that the original variables are not altered by the Mata function. An example of this logic is contained in `hprescott.ado` (`findit hprescott`).

An alternative to view matrices is provided by `st_data` and `st_sdata`, which copy data from Stata variables into Mata matrices, vectors or scalars. However, this operation duplicates the contents of those variables in Mata, and requires at least twice as much memory as consumed by the Stata variables (Mata does not have the full set of 1-, 2-, and 4-byte datatypes). Thus, although a view matrix can reference any and all variables currently in Stata's memory with minimal overhead, a matrix created by `st_data` will consume considerable memory (just as a matrix in Stata's own matrix language does).

As with `st_view`, dots may be used in `st_data` to specify all observations or all variables, and an optional *selectvar* can mark out desired observations. Otherwise, lists of variable names (or their indices in the dataset) are used to indicate the desired variables.

An alternative to view matrices is provided by `st_data` and `st_sdata`, which copy data from Stata variables into Mata matrices, vectors or scalars. However, this operation duplicates the contents of those variables in Mata, and requires at least twice as much memory as consumed by the Stata variables (Mata does not have the full set of 1-, 2-, and 4-byte datatypes). Thus, although a view matrix can reference any and all variables currently in Stata's memory with minimal overhead, a matrix created by `st_data` will consume considerable memory (just as a matrix in Stata's own matrix language does).

As with `st_view`, dots may be used in `st_data` to specify all observations or all variables, and an optional *selectvar* can mark out desired observations. Otherwise, lists of variable names (or their indices in the dataset) are used to indicate the desired variables.

We may also want to transfer other objects between the Stata and Mata environments. Although local and global macros, scalars and Stata matrices could be passed in the calling sequence to a Mata function, the function can only return one item. In order to return a number of objects to Stata—for instance, a list of macros, scalars and matrices as commonly found in `return list` from an *r*-class program—we use *st_functions*.

For local macros,

```
contents = st_local("macname")
st_local("macname", newvalue )
```

The first command will return the contents of Stata local macro *macname*. The second command will create and populate that local macro if it does not exist, or replace the contents if it does, with *newvalue*.

We may also want to transfer other objects between the Stata and Mata environments. Although local and global macros, scalars and Stata matrices could be passed in the calling sequence to a Mata function, the function can only return one item. In order to return a number of objects to Stata—for instance, a list of macros, scalars and matrices as commonly found in `return list` from an *r*-class program—we use *st_functions*.

For local macros,

```
contents = st_local("macname")
st_local("macname", newvalue )
```

The first command will return the contents of Stata local macro *macname*. The second command will create and populate that local macro if it does not exist, or replace the contents if it does, with *newvalue*.

Along the same lines, functions `st_global`, `st_numscalar` and `st_strscalar` may be used to retrieve the contents, create, or replace the contents of global macros, numeric scalars and string scalars, respectively. Function `st_matrix` performs these operations on Stata matrices.

All of these functions can be used to obtain the contents, create or replace the results in `r( )` or `e( )`: Stata's `return list` and `ereturn list`. Functions `st_rclear` and `st_eclear` can be used to delete all entries in those lists. Read-only access to the `c( )` objects is also available.

The `stata( )` function can execute a Stata command from within Mata.

Along the same lines, functions `st_global`, `st_numscalar` and `st_strscalar` may be used to retrieve the contents, create, or replace the contents of global macros, numeric scalars and string scalars, respectively. Function `st_matrix` performs these operations on Stata matrices.

All of these functions can be used to obtain the contents, create or replace the results in `r( )` or `e( )`: Stata's `return list` and `ereturn list`. Functions `st_rclear` and `st_eclear` can be used to delete all entries in those lists. Read-only access to the `c( )` objects is also available.

The `stata( )` function can execute a Stata command from within Mata.

Beyond the Stata interface functions, Mata contains a broad set of functions for matrix handling, mathematics and statistics, utility features, string handling and input-output.

Standard matrices can be defined with `I( )`, `e( )` (for unit vectors) and `J( )` (for constant matrices) with random matrices computed with `uniform( )`.

Matrix functions include `trace( )`, `det( )`, `norm( )`, `cond( )` and `rank`. A variety of functions provide decompositions, inversion and solution of linear systems, including Cholesky, LU, QR and SVD decompositions and solvers. The entire set of EISPACK/LAPACK routines are available for eigensystem analysis. Standard scalar functions are available and can be applied to vectors and matrices.

Beyond the Stata interface functions, Mata contains a broad set of functions for matrix handling, mathematics and statistics, utility features, string handling and input-output.

Standard matrices can be defined with `I( )`, `e( )` (for unit vectors) and `J( )` (for constant matrices) with random matrices computed with `uniform( )`.

Matrix functions include `trace( )`, `det( )`, `norm( )`, `cond( )` and `rank`. A variety of functions provide decompositions, inversion and solution of linear systems, including Cholesky, LU, QR and SVD decompositions and solvers. The entire set of EISPACK/LAPACK routines are available for eigensystem analysis. Standard scalar functions are available and can be applied to vectors and matrices.

Beyond the Stata interface functions, Mata contains a broad set of functions for matrix handling, mathematics and statistics, utility features, string handling and input-output.

Standard matrices can be defined with `I( )`, `e( )` (for unit vectors) and `J( )` (for constant matrices) with random matrices computed with `uniform( )`.

Matrix functions include `trace( )`, `det( )`, `norm( )`, `cond( )` and `rank`. A variety of functions provide decompositions, inversion and solution of linear systems, including Cholesky, LU, QR and SVD decompositions and solvers. The entire set of EISPACK/LAPACK routines are available for eigensystem analysis. Standard scalar functions are available and can be applied to vectors and matrices.

Matrix utility functions include `rows( )`, `cols( )`, `length( )` (of a vector), `issymmetric( )`, `isdiagonal( )` and `missing( )` (`nonmissing( )`) to count (non-)missing values. You can also use `rowmissing( )` and `colmissing` to analyze missingness.

A variety of row-wise and column-wise functions are available: `rowmin( )` and `colmin( )` and equivalent `...max`, `rowsum( )`, `colsum( )`, and overall `sum( )`. Routines for evaluating convergence include `reldif( )`, `mreldif( )` and `mreldifsym( )` (difference from symmetry).

Matrix utility functions include `rows( )`, `cols( )`, `length( )` (of a vector), `issymmetric( )`, `isdiagonal( )` and `missing( )` (`nonmissing( )`) to count (non-)missing values. You can also use `rowmissing( )` and `colmissing` to analyze missingness.

A variety of row-wise and column-wise functions are available: `rowmin( )` and `colmin( )` and equivalent `...max`, `rowsum( )`, `colsum( )`, and overall `sum( )`. Routines for evaluating convergence include `reldif( )`, `mreldif( )` and `mreldifsym( )` (difference from symmetry).

Statistical functions include `mean( )`, `variance( )` and `correlation`, as well as utility routines such as `cross( )` and `crossdev( )` to compute cross-products. Distribution-specific functions include, among many others, `lnfactorial( )`, `lngamma( )`, `normalden( )`, `normal( )`, `invnormal( )`, `binomial( )`.

For the χ^2, *t*, *F* and β distributions both PDFs and CDFs are available for the distribution and their inverses. Noncentral χ^2, *F* and β are also handled. The `logit( )` and `invlogit( )` functions are available for analysis of the logistic distribution.

Mathematical functions also are provided to handle Fourier transforms, creation of power spectra, cubic splines and polynomial arithmetic.

Statistical functions include `mean( )`, `variance( )` and `correlation`, as well as utility routines such as `cross( )` and `crossdev( )` to compute cross-products. Distribution-specific functions include, among many others, `lnfactorial( )`, `lngamma( )`, `normalden( )`, `normal( )`, `invnormal( )`, `binomial( )`.

For the χ^2, *t*, *F* and β distributions both PDFs and CDFs are available for the distribution and their inverses. Noncentral χ^2, *F* and β are also handled. The `logit( )` and `invlogit( )` functions are available for analysis of the logistic distribution.

Mathematical functions also are provided to handle Fourier transforms, creation of power spectra, cubic splines and polynomial arithmetic.

Mata's string functions largely parallel those available in Stata. As in Stata, the + operator is overloaded to denote string concatenation. In addition, the * operator can be used to duplicate strings.

A full set of input-output functions make Mata an easier environment to perform arbitrary I/O than Stata itself. Functions are available to query the local filesystem, create, change or remove directories and work with paths embedded in filenames or Stata's ADOPATH settings. You may read and write both ASCII and binary files as well as matrices: the latter a facility lacking from official Stata. You may also direct output to the Results window or read input from the Command window.

Mata's string functions largely parallel those available in Stata. As in Stata, the + operator is overloaded to denote string concatenation. In addition, the * operator can be used to duplicate strings.

A full set of input-output functions make Mata an easier environment to perform arbitrary I/O than Stata itself. Functions are available to query the local filesystem, create, change or remove directories and work with paths embedded in filenames or Stata's ADOPATH settings. You may read and write both ASCII and binary files as well as matrices: the latter a facility lacking from official Stata. You may also direct output to the Results window or read input from the Command window.

We present an example of constructing a Stata command that uses Mata to achieve a useful task. We often have timeseries data at a higher frequency (e.g., monthly) and want to work with it at a lower frequency (e.g., quarterly or annual). We may use Stata's `collapse` command to achieve this, or the author's `tscollap`. But both of those solutions destroy the current dataset. In some cases—for instance, for graphical or tabular presentation—we may want to retain the original (high-frequency) data and add the lower-frequency series to the dataset. Note that the computation of these series could also be handled with the `egen group( )` function, but that would intersperse the lower-frequency data with missing values.

We design a Stata command, `avgper`, which takes a single variable and optional `if` or `in` conditions along with a mandatory option `per( )`: the number of periods to be averaged into a lower-frequency series. We could handle multiple variables or alternative transformations (e.g., sums over the periods) with an expanded version of this routine.

We present an example of constructing a Stata command that uses Mata to achieve a useful task. We often have timeseries data at a higher frequency (e.g., monthly) and want to work with it at a lower frequency (e.g., quarterly or annual). We may use Stata's `collapse` command to achieve this, or the author's `tscollap`. But both of those solutions destroy the current dataset. In some cases—for instance, for graphical or tabular presentation—we may want to retain the original (high-frequency) data and add the lower-frequency series to the dataset. Note that the computation of these series could also be handled with the `egen group( )` function, but that would intersperse the lower-frequency data with missing values.

We design a Stata command, `avgper`, which takes a single variable and optional `if` or `in` conditions along with a mandatory option `per( )`: the number of periods to be averaged into a lower-frequency series. We could handle multiple variables or alternative transformations (e.g., sums over the periods) with an expanded version of this routine.

The Stata ado-file defines the program, then validates the `per( )` argument. We require that the number of high-frequency observations is a multiple of `per`.

```
program avgper, rclass
version 9.2
syntax varlist(max=1 numeric) [if] [in], per(integer)
marksample touse
qui summ `varlist' if `touse'
* validate per versus selected sample
if `per' <= 0 | `per' >= `r(N)' {
display as error "per must be >0 and <nobs."
error 198
}
if mod(`r(N)',`per' != 0) {
display as error "nobs must be a multiple of per."
error 198
}
```

We attempt to create a new variable named *vname*A`n`, where *vname* is the specified variable and *n* is the value of `per( )`. If that variable name is already in use, the routine exits with error. The variable is created with missing values, as it is only a placeholder. With successful validation, we pass the arguments to the Mata function `avgper`.

```
* validate the new varname
local newvar = "`varlist'"+"A"+string(`per')
qui gen `newvar' = .
* pass the varname and newvarname to mata
mata: avgper("`varlist'","`newvar'", ///
                `per',"`touse'")
end
```

The Mata function to achieve this task is quite succinct, requiring that we effectively reshape the data into a matrix with `per` columns using `colshape( )`, then scale by `1/per` to create averages:

```
version 9.2
mata:
void avgper(string scalar vname,
string scalar newvname,
        real scalar per,
        string scalar touse)
{
st_view(v1=.,.,vname,touse)
st_view(v2=.,.,newvname)
v3 = colshape(v1',per) * J(per,1,1/per)
v2[(1::rows(v3)),] = v3
}
end
```

Note that we make use of *view matrices* to access the contents of *vname*—the existing variable name specified in the `avgper` command—and to access *newvname* in Mata, which is our newly-created `` `newvar' `` in the Stata code. The `colshape` function creates a matrix which is $q \times per$, where q is the number of low-frequency observations to be created. Postmultiplying that matrix by a `per`–element column vector of `1/per` produces the desired result of a q-element column vector. That object—*v3* in Mata—is then written to the first q rows of view matrix *v2*, which corresponds to the Stata variable `` `newvar' ``.

By using Mata and a simple matrix expression, we have considerably simplified the computation of the lower-frequency series, and may apply the routine to any combination of data frequencies (e.g., business-daily data to weekly) without concern for Stata's support of a particular timeseries frequency.