Using Mata to work more effectively in Stata

Christopher F Baum

Boston College and DIW Berlin

FNASUG, San Francisco, November 2008

Since the release of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with most of the capabilities of MATLAB, R, Ox or GAUSS. You can use Mata interactively, or you can develop Mata functions to be called from Stata. In this talk, we emphasize the latter use of Mata.

Mata functions may be particularly useful where the algorithm you wish to implement already exists in matrix-language form. It is quite straightforward to translate the logic of other matrix languages into Mata: much more so than converting it into Stata's matrix language.

A large library of mathematical and matrix functions is provided in Mata, including optimization routines, equation solvers, decompositions, eigensystem routines and probability density functions (enhanced in version 10.1). 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.

Since the release of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with most of the capabilities of MATLAB, R, Ox or GAUSS. You can use Mata interactively, or you can develop Mata functions to be called from Stata. In this talk, we emphasize the latter use of Mata.

Mata functions may be particularly useful where the algorithm you wish to implement already exists in matrix-language form. It is quite straightforward to translate the logic of other matrix languages into Mata: much more so than converting it into Stata's matrix language.

A large library of mathematical and matrix functions is provided in Mata, including optimization routines, equation solvers, decompositions, eigensystem routines and probability density functions (enhanced in version 10.1). 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.

Since the release of version 9, Stata contains a full-fledged matrix programming language, *Mata*, with most of the capabilities of MATLAB, R, Ox or GAUSS. You can use Mata interactively, or you can develop Mata functions to be called from Stata. In this talk, we emphasize the latter use of Mata.

Mata functions may be particularly useful where the algorithm you wish to implement already exists in matrix-language form. It is quite straightforward to translate the logic of other matrix languages into Mata: much more so than converting it into Stata's matrix language.

A large library of mathematical and matrix functions is provided in Mata, including optimization routines, equation solvers, decompositions, eigensystem routines and probability density functions (enhanced in version 10.1). 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 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 or Stata/MP, 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 at least twice the memory needed for that set of variables.

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 or Stata/MP, 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 at least twice the memory needed for that set of variables.

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, regardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements (but not both). This implies that you can use Mata productively in a list processing environment as well as in a numeric context.

For example, a prominent list-handling command, Bill Gould's `adoupdate`, is written almost entirely in Mata. `viewsource adoupdate.ado` reveals that only 22 lines of code (out of 1,193 lines) are in the ado-file language. The rest is 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, regardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements (but not both). This implies that you can use Mata productively in a list processing environment as well as in a numeric context.

For example, a prominent list-handling command, Bill Gould's `adoupdate`, is written almost entirely in Mata. `viewsource adoupdate.ado` reveals that only 22 lines of code (out of 1,193 lines) are in the ado-file language. The rest is 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, regardless of their size.

Unlike some matrix programming languages, Mata matrices can contain either numeric elements or string elements (but not both). This implies that you can use Mata productively in a list processing environment as well as in a numeric context.

For example, a prominent list-handling command, Bill Gould's `adoupdate`, is written almost entirely in Mata. `viewsource adoupdate.ado` reveals that only 22 lines of code (out of 1,193 lines) are in the ado-file language. The rest is Mata.

Last but by no means least, ado-file code written in the matrix language with explicit subscript references is *slow*. Even if such a routine avoids explicit subscripting, its performance may be unacceptable. For instance, David Roodman's `xtabond2` can run in version 7 or 8 without Mata, or in version 9 or 10 with Mata. The non-Mata version is an order of magnitude slower when applied to reasonably sized estimation problems.

In contrast, 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.

Last but by no means least, ado-file code written in the matrix language with explicit subscript references is *slow*. Even if such a routine avoids explicit subscripting, its performance may be unacceptable. For instance, David Roodman's `xtabond2` can run in version 7 or 8 without Mata, or in version 9 or 10 with Mata. The non-Mata version is an order of magnitude slower when applied to reasonably sized estimation problems.

In contrast, 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 interfaced with Stata provides for an efficient division of labour. 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.

Mata interfaced with Stata provides for an efficient division of labour. 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 the rest of this talk, I will discuss:

- Basic elements of Mata syntax
- Design of a Mata function
- Mata's interface functions
- Some examples of Stata–Mata routines

In the rest of this talk, I will discuss:

- Basic elements of Mata syntax
- Design of a Mata function
- Mata's interface functions
- Some examples of Stata–Mata routines

In the rest of this talk, I will discuss:

- Basic elements of Mata syntax
- Design of a Mata function
- Mata's interface functions
- Some examples of Stata–Mata routines

In the rest of this talk, I will discuss:

- Basic elements of Mata syntax
- Design of a Mata function
- Mata's interface functions
- Some examples of Stata–Mata routines

To understand Mata syntax, you must be familiar with its operators. The comma is the *column-join* operator, so

```
: r1 = ( 1, 2, 3 )
```

creates a three-element row vector. We could also construct this vector using the *row range operator* (..) as

```
: r1 = (1..3)
```

The backslash is the *row-join* operator, so

```
c1 = ( 4 \ 5 \ 6 )
```

creates a three-element column vector. We could also construct this vector using the *column range operator* (::) as

```
: c1 = (4::6)
```

To understand Mata syntax, you must be familiar with its operators. The comma is the *column-join* operator, so

```
: r1 = ( 1, 2, 3 )
```

creates a three-element row vector. We could also construct this vector using the *row range operator* (..) as

```
: r1 = (1..3)
```

The backslash is the *row-join* operator, so

```
c1 = ( 4 \ 5 \ 6 )
```

creates a three-element column vector. We could also construct this vector using the *column range operator* (::) as

```
: c1 = (4::6)
```

We may combine the column-join and row-join operators:

```
m1 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
```

creates a 3 $\times$ 3 matrix.

The matrix could also be constructed with the row range operator:

```
m1 = ( 1..3 \ 4..6 \ 7..9 )
```

We may combine the column-join and row-join operators:

```
m1 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
```

creates a 3 $\times$ 3 matrix.

The matrix could also be constructed with the row range operator:

```
m1 = ( 1..3 \ 4..6 \ 7..9 )
```

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

```
r2 = ( 1 \ 2 \ 3 )'
```

is a row vector.

The comma and backslash operators can be used on vectors and matrices as well as scalars, so

```
r3 = r1, c1'
```

will produce a six-element row vector, and

```
c2 = r1' \ c1
```

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

```
r2 = ( 1 \ 2 \ 3 )'
```

is a row vector.

The comma and backslash operators can be used on vectors and matrices as well as scalars, so

```
r3 = r1, c1'
```

will produce a six-element row vector, and

```
c2 = r1' \ c1
```

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

```
r2 = ( 1 \ 2 \ 3 )'
```

is a row vector.

The comma and backslash operators can be used on vectors and matrices as well as scalars, so

```
r3 = r1, c1'
```

will produce a six-element row vector, and

```
c2 = r1' \ c1
```

creates 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 = r1' + c1
h = r1 * c1
j = c1 * r1
```

In this example `h` will be the 1×1 dot product of vectors `r1, c1` while `j` is their 3×3 outer product.

One of Mata's most powerful features is the *colon operator*. Mata's algebraic operators, including the forward slash (/) for division, also can be used in element-by-element computations when preceded by a colon:

```
k = r1' :* c1
```

will produce a three-element column vector, with elements as the product of the respective elements: $k_i = r1_i \; c1_i, \; i = 1, \ldots, 3$.

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

```
r4 = ( 1, 2, 3 )
m2 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
m3 = r4 :+ m2
m4 = m1 :/ r1
```

adds the row vector `r4` to each row of the 3 $\times$ 3 matrix `m2` to form `m3`, and divides the elements of each row of matrix `m1` by the corresponding elements of row vector `r1` to form `m4`.

Mata'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:

```
r4 = ( 1, 2, 3 )
m2 = ( 1, 2, 3 \ 4, 5, 6 \ 7, 8, 9 )
m3 = r4 :+ m2
m4 = m1 :/ r1
```

adds the row vector `r4` to each row of the 3 $\times$ 3 matrix `m2` to form `m3`, and divides the elements of each row of matrix `m1` by the corresponding elements of row vector `r1` to form `m4`.

Mata'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.

As in Stata, the logical and (`&`) and or (`|`) operators may 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.

As in Stata, the logical and (`&`) and or (`|`) operators may 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.

As in Stata, the logical and (`&`) and or (`|`) operators may 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.

As in Stata, the logical and (`&`) and or (`|`) operators may 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)` references row `i` and those three columns of `x`. Missing values (dots) 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),.]` 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)` references row `i` and those three columns of `x`. Missing values (dots) 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),.]` 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)` references row `i` and those three columns of `x`. Missing values (dots) 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),.]` 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,. |]` specifies the submatrix ending in the last column and `x[| 2,2 \ .,.|]` discards 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 \ .,.|]` discards 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,. |]` specifies the submatrix ending in the last column and `x[| 2,2 \ .,.|]` discards 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 \ .,.|]` discards 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 the C language:

for (*starting_value; ending_value; incr*) {
    statements
}

where the three elements define the starting value, ending value or bound and increment or decrement of the loop. For instance:

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

prints the integers 1 to 10 on separate lines.

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 (exp1) {
    statements1
}
   else if (exp2) {
    statements2
}
   else {
    statements3
}
```

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

if (*exp*) *statement*

if (*exp*) *statement1*
else *statement2*

```
if (exp1) {
    statements1
}
   else if (exp2) {
    statements2
}
   else {
    statements3
}
```

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

if (*exp*) *statement*

if (*exp*) *statement1*
  else *statement2*

if (*exp1*) {
  statements1
}
  else if (*exp2*) {
  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. They may precede or follow the variable.

The operator `A # B` produces the Kronecker or direct product of `A` and `B`.

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. They may precede or follow the variable.

The operator `A # B` produces the Kronecker or direct product of `A` and `B`.

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. They may precede or follow the variable.

The operator `A # B` produces the Kronecker or direct product of `A` and `B`.

To call Mata code within an ado-file, you must define a Mata function, which is the equivalent of a Stata ado-file program. Unlike a Stata program, a Mata function has an explicit *return type* and a set of *arguments*. A function may be of return type `void` if it does not need a return statement. Otherwise, a function is typed in terms of two characteristics: its *element type* and their *organization type*. For instance,

```
real scalar calcsum(real vector x)
```

declares that the Mata `calcsum` function will return a real scalar. It has one argument: an object `x`, which must be a `real vector`.

Element types may be `real, complex, numeric, string, pointer, transmorphic`. A `transmorphic` object may be filled with any of the other types. A `numeric` object may be either `real` or `complex`. Unlike Stata, Mata supports complex arithmetic.

There are five organization types: `matrix, vector, rowvector, colvector, scalar`. Strictly speaking the latter four are just special cases of `matrix`. In Stata's matrix language, all matrices have two subscripts, neither of which can be zero. In Mata, all but the `scalar` may have zero rows and/or columns. Three- (and higher-) dimension matrices can be implemented by the use of the `pointer` element type, not to be discussed further in this talk.

Element types may be `real`, `complex`, `numeric`, `string`, `pointer`, `transmorphic`. A `transmorphic` object may be filled with any of the other types. A `numeric` object may be either `real` or `complex`. Unlike Stata, Mata supports complex arithmetic.

There are five organization types: `matrix`, `vector`, `rowvector`, `colvector`, `scalar`. Strictly speaking the latter four are just special cases of `matrix`. In Stata's matrix language, all matrices have two subscripts, neither of which can be zero. In Mata, all but the `scalar` may have zero rows and/or columns. Three- (and higher-) dimension matrices can be implemented by the use of the `pointer` element type, not to be discussed further in this talk.

A Mata function definition includes an *argument list*, which may be blank. The names of arguments are required and arguments are positional. The order of arguments in the calling sequence must match that in the Mata function. If the argument list includes a vertical bar (|), following arguments are optional.

Within a function, variables may be explicitly declared (and must be declared if `matastrict` mode is used). It is good programming practice to do so, as then variables cannot be inadvertently misused. Variables within a Mata function have *local scope*, and are not accessible outside the function unless declared as `external`.

A Mata function may only return one item (which could, however, be a multi-element *structure*. If the function is to return multiple objects, Mata's `st_...` functions should be used, as we will demonstrate.

A Mata function definition includes an *argument list*, which may be blank. The names of arguments are required and arguments are positional. The order of arguments in the calling sequence must match that in the Mata function. If the argument list includes a vertical bar (|), following arguments are optional.

Within a function, variables may be explicitly declared (and must be declared if `matastrict` mode is used). It is good programming practice to do so, as then variables cannot be inadvertently misused. Variables within a Mata function have *local scope*, and are not accessible outside the function unless declared as `external`.

A Mata function may only return one item (which could, however, be a multi-element *structure*. If the function is to return multiple objects, Mata's `st_...` functions should be used, as we will demonstrate.

A Mata function definition includes an *argument list*, which may be blank. The names of arguments are required and arguments are positional. The order of arguments in the calling sequence must match that in the Mata function. If the argument list includes a vertical bar (|), following arguments are optional.

Within a function, variables may be explicitly declared (and must be declared if `matastrict` mode is used). It is good programming practice to do so, as then variables cannot be inadvertently misused. Variables within a Mata function have *local scope*, and are not accessible outside the function unless declared as `external`.

A Mata function may only return one item (which could, however, be a multi-element *structure*. If the function is to return multiple objects, Mata's `st_...` functions should be used, as we will demonstrate.

If you're using Mata functions in conjunction with Stata's ado-file language, one of the most important set of tools are Mata's interface functions: the `st_` functions.

The first category of these functions provide access to data. Stata and Mata have separate workspaces, and these functions allow you to access and update Stata's workspace from inside Mata. For instance, `st_nobs(), st_nvar()` provide the same information as `describe` in Stata, which returns `r(N), r(k)` in its return list. Mata functions `st_data(), st_view()` allow you to access any rectangular subset of Stata's numeric variables, and `st_sdata(), st_sview()` do the same for string variables.

If you're using Mata functions in conjunction with Stata's ado-file language, one of the most important set of tools are Mata's interface functions: the `st_` functions.

The first category of these functions provide access to data. Stata and Mata have separate workspaces, and these functions allow you to access and update Stata's workspace from inside Mata. For instance, `st_nobs(), st_nvar()` provide the same information as `describe` in Stata, which returns `r(N), r(k)` in its return list. Mata functions `st_data(), st_view()` allow you to access any rectangular subset of Stata's numeric variables, and `st_sdata(), st_sview()` do the same for string variables.

One of the most useful Mata concepts is the *view matrix*, which as its name implies is a view of some of Stata's variables for specified observations, created by a call to `st_view()`. Unlike most Mata functions, `st_view()` does not return a result. It requires 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.

```
st_view(x, ., varname, touse)
```

States that the previously-declared Mata vector `x` should be created from all the observations (specified by the missing second argument) of `varname`, as modified by the contents of `touse`. In the Stata code, the `marksample` command imposes any `if` or `in` conditions by setting the indicator variable `touse`.

The Mata statements

```
real matrix Z
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. For instance, if `varlist` is a `string scalar` argument containing Stata variable names,

```
void foo( string scalar varlist )
...
st_view(X=., ., tokens(varlist), touse)
```

creates matrix `X` containing those 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:

```
X = st_data(., .)
```

places a copy of all variables in Stata's memory into matrix `X`. However, this operation requires at least twice as much memory as consumed by the Stata variables, as Mata does not have Stata's full set of 1-, 2-, and 4-byte data types. Thus, although a view matrix can reference any 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.

Similar to `st_view()`, an optional third argument to `st_data()` can mark out desired observations.

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:

```
X = st_data(., .)
```

places a copy of all variables in Stata's memory into matrix `X`. However, this operation requires at least twice as much memory as consumed by the Stata variables, as Mata does not have Stata's full set of 1-, 2-, and 4-byte data types. Thus, although a view matrix can reference any 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.

Similar to `st_view()`, an optional third argument to `st_data()` can mark out desired observations.

A very important aspect of views: using a view matrix rather than copying data into Mata with `st_data()` implies that any changes made to the view matrix will be reflected in Stata's variables' contents. This is a very powerful feature that allows us to easily return information generated in Mata back to Stata's variables, or create new content in existing variables.

This may or may not be what you want to do. Keep in mind that any alterations to a view matrix will change Stata's variables, just as a `replace` command in Stata would. If you want to ensure that Mata computations cannot alter Stata's variables, avoid the use of views, or use them with caution. You may use `st_addvar()` to explicitly create new Stata variables, and `st_store()` to populate their contents.

A very important aspect of views: using a view matrix rather than copying data into Mata with `st_data()` implies that any changes made to the view matrix will be reflected in Stata's variables' contents. This is a very powerful feature that allows us to easily return information generated in Mata back to Stata's variables, or create new content in existing variables.

This may or may not be what you want to do. Keep in mind that any alterations to a view matrix will change Stata's variables, just as a `replace` command in Stata would. If you want to ensure that Mata computations cannot alter Stata's variables, avoid the use of views, or use them with caution. You may use `st_addvar()` to explicitly create new Stata variables, and `st_store()` to populate their contents.

A Mata function may take one (or several) existing variables and create a transformed variable (or set of variables). To do that with views, create the new variable(s), pass the name(s) as a *newvarlist* and set up a view matrix.

```
st_view(Z=., ., tokens(newvarlist), touse)
```

Then compute the new content as:

```
Z[., .] = result of computation
```

It is very important to use the `[., .]` construct as shown. `Z =` will cause a new matrix to be created and break the link to the view.

You may also create new variables and fill in their contents by combining these techniques:

```
st_view(Z, ., st_addvar(("int", "float"), ///
        ("idnr", "bp")))
Z[., .] = result of computation
```

In this example, we create two new Stata variables, of data type `int` and `float`, respectively, named `idnr` and `bp`.

You may also use *subviews* and, for panel data, *panelsubviews*. We will not discuss those here.

You may also create new variables and fill in their contents by combining these techniques:

```
st_view(Z, ., st_addvar(("int", "float"), ///
        ("idnr", "bp")))
Z[., .] = result of computation
```

In this example, we create two new Stata variables, of data type `int` and `float`, respectively, named `idnr` and `bp`.

You may also use *subviews* and, for panel data, *panelsubviews*. We will not discuss those here.

You 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 is commonly found in the `return list` from an *r*-class program—we use the appropriate *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*.

You 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 is commonly found in the `return list` from an *r*-class program—we use the appropriate *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.

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

```
program varextrema, rclass
    version 10.1
    syntax varname(numeric) [if] [in]
    marksample touse
    mata: calcextrema( "`varlist'", "`touse'" )
    display as txt " min ( `varlist' ) = " as res r(min)
    display as txt " max ( `varlist' ) = " as res r(max)
    return scalar min = r(min)
    return scalar max = r(max)
end
```

Our ado-language code creates a Stata command, `varextrema`, which requires the name of a single numeric Stata variable. You may specify `if` *exp* or `in` *range* conditions. The Mata function `calcextrema` 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 two numeric scalars: `r(min)`, `r(max)`. Those are returned in turn to the calling program in the `varextrema` return list.

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

```
version 10.1
mata:
mata set matastrict on
void calcextrema(string scalar varname, ///
                 string scalar touse)
{
  real colvector x, cmm
  st_view(x, ., varname, touse)
  cmm = colminmax(x)
  st_numscalar("r(min)", cmm[1])
  st_numscalar("r(max)", cmm[2])
}
end
```

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 need to send two results back to 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 `colminmax()` function computes the extrema of its arguments as a two-element vector, and `st_numscalar()` returns each of them to Stata as `r(min)`, `r(max)` respectively.

Now let's consider a slightly more ambitious task. Say that you would like to *center* a number of variables on their means, creating a new set of transformed variables. Surprisingly, official Stata does not have such a command, although Ben Jann's `center` command does so. Accordingly, we write Stata command `centervars`, employing a Mata function to do the work.

The Stata code:

```
program centervars, rclass
    version 10.1
    syntax varlist(numeric) [if] [in], ///
                GENerate(string) [DOUBLE]
    marksample touse
    quietly count if `touse'
    if `r(N)' == 0  error 2000
    foreach v of local varlist {
        confirm new var `generate'`v'
    }
    foreach v of local varlist {
        qui generate `double' `generate'`v' = .
        local newvars "`newvars' `generate'`v'"
    }
    mata: centerv( "`varlist'", "`newvars'", "`touse'" )
end
```

The file `centervars.ado` contains a Stata command, `centervars`, that takes a list of numeric variables and a mandatory `generate()` option. The contents of that option are used to create new variable names, which then are tested for validity with `confirm new var`, and if valid generated as missing. The list of those new variables is assembled in local macro `newvars`. The original `varlist` and the list of `newvars` is passed to the Mata function `centerv()` along with `touse`, the temporary variable that marks out the desired observations.

The Mata code:

```
version 10.1
mata:
void centerv( string scalar varlist, ///
              string scalar newvarlist,
              string scalar touse)
{
  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ ., . ] = X :- mean(X)
}
end
```

In 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 observations selected by `if` and `in` conditions.

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 `Z[,]=` notation, the Stata variables themselves are modified. When the Mata function returns to Stata, the contents and descriptive statistics of the variables in *varlist* will be altered.

In 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 observations selected by `if` and `in` conditions.

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 `Z[,]=` notation, the Stata variables themselves are modified. When the Mata function returns to Stata, the contents and descriptive statistics of the variables in *varlist* will be altered.

One of the advantages of Mata use is evident here: we need not loop over the variables in order to demean them, as the operation can be written in terms of matrices, and the computation done very efficiently even if there are many variables and observations. Also note that performing these calculations in Mata incurs minimal overhead, as the matrix `Z` is merely a view on the Stata variables in `newvars`. One caveat: Mata's `mean()` function performs *listwise deletion*, like Stata's `correlate` command.

Let's consider adding a feature to `centervars`: the ability to transform variables before centering with one of several mathematical functions (`abs(), exp(), log(), sqrt()`). The user will provide the name of the desired transformation, which defaults to the identity transformation, and Stata will pass the name of the function (actually, a pointer to the function) to Mata. We call this new command `centertrans`.

The Stata code:

```
program centertrans, rclass
   version 10.1
   syntax varlist(numeric) [if] [in], ///
             GENerate(string) [TRans(string)] [DOUBLE]
   marksample touse
   quietly count if `touse'
   if `r(N)' == 0  error 2000
   foreach v of local varlist {
       confirm new var `generate'`v'
   }
   local trops abs exp log sqrt
   if "`trans'" == "" {
     local trfn  "mf_iden"
    }
    else {
     local ntr : list posof "`trans'" in trops
     if !`ntr' {
     display as err "Error: trans must be chosen from `trops'"
     error 198
     }
    local trfn : "mf_`trans'"
    }
   foreach v of local varlist {
       qui generate `double' `generate'`trans'`v' = .
       local newvars "`newvars' `generate'`trans'`v'"
   }
   mata: centertrans( "`varlist'", "`newvars'", &`trfn'(), "`touse'" )
end
```

In Mata, we must define “wrapper functions" for the transformations, as we cannot pass a pointer to a built-in function. We define trivial functions such as

```
function mf_log(x) return(log(x))
```

which defines the `mf_log()` scalar function as taking the log of its argument.

The Mata function `centertrans()` receives the function argument as

```
pointer(real scalar function) scalar f
```

To apply the function, we use

```
Z[ ., . ] = (*f)(X)
```

which applies the function referenced by `f` to the elements of the matrix `X`.

In Mata, we must define “wrapper functions" for the transformations, as we cannot pass a pointer to a built-in function. We define trivial functions such as

```
function mf_log(x) return(log(x))
```

which defines the `mf_log()` scalar function as taking the log of its argument.

The Mata function `centertrans()` receives the function argument as

```
pointer(real scalar function) scalar f
```

To apply the function, we use

```
Z[ ., . ] = (*f)(X)
```

which applies the function referenced by `f` to the elements of the matrix `X`.

In Mata, we must define “wrapper functions" for the transformations, as we cannot pass a pointer to a built-in function. We define trivial functions such as

```
function mf_log(x) return(log(x))
```

which defines the `mf_log()` scalar function as taking the log of its argument.

The Mata function `centertrans()` receives the function argument as

```
pointer(real scalar function) scalar f
```

To apply the function, we use

```
Z[ ., . ] = (*f)(X)
```

which applies the function referenced by `f` to the elements of the matrix `X`.

The Mata code:

```
version 10.1
mata:
function mf_abs(x) return(abs(x))
function mf_exp(x) return(exp(x))
function mf_log(x) return(log(x))
function mf_sqrt(x) return(sqrt(x))
function mf_iden(x) return(x)

void centertrans( string scalar varlist, ///
                  string scalar newvarlist,
                  pointer(real scalar function) scalar f,
                  string scalar touse)
{
  real matrix X, Z
  st_view(X=., ., tokens(varlist), touse)
  st_view(Z=., ., tokens(newvarlist), touse)
  Z[ , ] = (*f)(X)
  Z[ , ] = Z :- mean(Z)
}
end
```

Mata may prove particularly useful when you have an algorithm readily expressed in matrix form. Many estimation problems fall into that category. In this last example, I illustrate how "heteroskedastic OLS" (HOLS) can be easily implemented in Mata, with Stata code handling all of the housekeeping details. This section draws on joint work with Mark Schaffer.

HOLS is a form of Generalised Method of Moments (GMM) estimation in which you assert not only that the regressors X are uncorrelated with the error, but that you also have additional variables Z which are also uncorrelated with the error. Those additional "orthogonality conditions" serve to improve the efficiency of estimation when non-$i.i.d.$ errors are encountered. This estimator is described in `help ivreg2` and in Baum, Schaffer & Stillman, *Stata Journal*, 2007.

Mata may prove particularly useful when you have an algorithm readily expressed in matrix form. Many estimation problems fall into that category. In this last example, I illustrate how "heteroskedastic OLS" (HOLS) can be easily implemented in Mata, with Stata code handling all of the housekeeping details. This section draws on joint work with Mark Schaffer.

HOLS is a form of Generalised Method of Moments (GMM) estimation in which you assert not only that the regressors X are uncorrelated with the error, but that you also have additional variables Z which are also uncorrelated with the error. Those additional "orthogonality conditions" serve to improve the efficiency of estimation when non-$i.i.d.$ errors are encountered. This estimator is described in `help ivreg2` and in Baum, Schaffer & Stillman, *Stata Journal*, 2007.

The `hols` command takes a dependent variable and a set of regressors. The `exog()` option may be used to provide the names of additional variables uncorrelated with the error. By default, `hols` calculates estimates under the assumption of *i.i.d.* errors. If the `robust` option is used, the estimates' standard errors are robust to arbitrary heteroskedasticity.

Following estimation, the `estimates post` and `estimates display` commands are used to provide standard Stata estimation output. If the `exog` option is used, a Sargan–Hansen *J* test statistic is provided. A significant value of the *J* statistic implies rejection of the null hypothesis of orthogonality.

The `hols` command takes a dependent variable and a set of regressors. The `exog()` option may be used to provide the names of additional variables uncorrelated with the error. By default, `hols` calculates estimates under the assumption of *i.i.d.* errors. If the `robust` option is used, the estimates' standard errors are robust to arbitrary heteroskedasticity.

Following estimation, the `estimates post` and `estimates display` commands are used to provide standard Stata estimation output. If the `exog` option is used, a Sargan–Hansen *J* test statistic is provided. A significant value of the *J* statistic implies rejection of the null hypothesis of orthogonality.

The Stata code:

```
program hols, eclass
version 10.1
syntax varlist [if] [in] [, exog(varlist) robust ]
local depvar: word 1 of `varlist'
local regs: list varlist - depvar
marksample touse
markout `touse'  `exog'
tempname b V
mata: m_hols("`depvar'",  "`regs'", "`exog'", "`touse'", "`robust'")
mat `b' = r(beta)
mat `V' = r(V)
local vnames  `regs' _cons
matname `V' `vnames'
matname `b' `vnames', c(.)
local N = r(N)
ereturn post `b' `V', depname(`depvar') obs(`N') esample(`touse')
ereturn local depvar = "`depvar'"
ereturn scalar N = r(N)
ereturn scalar j = r(j)
ereturn scalar L = r(L)
ereturn scalar K = r(K)
if "`robust'" != "" {
    ereturn local vcetype "Robust"
}
local res = cond("`exog'" != "", "Heteroskedastic", "")
display _newline "`res' OLS results" _col(60) "Number of obs = " e(N)
ereturn display
display "Sargan-Hansen J statistic: " %7.3f e(j)
if ( e(L)-e(K) > 0 ) {
display "Chi-sq(" %3.0f e(L)-e(K) "  )       P-val = " ///
    %5.4f chiprob(e(L)-e(K), e(j)) _newline
}
end
```

The Mata code makes use of a function to compute the covariance matrix: either the classical, non-robust *VCE* or the heteroskedasticity-robust *VCE*. For ease of reuse, this logic is broken out into a standalone function.

```
version 10.1
mata:
real matrix m_myomega(real rowvector beta,
                      real colvector Y,
                      real matrix X,
                      real matrix Z,
                      string scalar robust)
{
        real matrix QZZ, omega
        real vector e, e2
        real scalar N, sigma2
// Calculate residuals from the coefficient estimates
N = rows(Z)
e = Y - X * beta'
if (robust=="") {
// Compute classical, non-robust covariance matrix
    QZZ = 1/N * quadcross(Z, Z)
    sigma2 = 1/N * quadcross(e, e)
    omega = sigma2 * QZZ
}
else {
// Compute heteroskedasticity-consistent covariance matrix
    e2 = e:^2
    omega = 1/N * quadcross(Z, e2, Z)
}
_makesymmetric(omega)
return (omega)
}
```

The main Mata code takes as arguments the dependent variable name, the list of regressors, the optional list of additional exogenous variables, the marksample indicator (`touse`) and the `robust` flag. The logic for linear GMM can be expressed purely in terms of matrix algebra.

The Mata code:

```
void m_hols(string scalar yname,
                string scalar inexognames,
                string scalar exexognames,
                string scalar touse,
                string scalar robust)
{
real matrix Y, X2, Z1, X, Z, QZZ, QZX, W, omega, V
real vector cons, beta_iv, beta_gmm, e, gbar
real scalar K, L, N, j
st_view(Y, ., tokens(yname), touse)
st_view(X2, ., tokens(inexognames), touse)
st_view(Z1, ., tokens(exexognames), touse)
// Constant is added by default.
cons = J(rows(X2), 1, 1)
X = X2, cons
Z = Z1, X
K = cols(X)
L = cols(Z)
N = rows(Y)
QZZ = 1/N * quadcross(Z, Z)
QZX = 1/N * quadcross(Z, X)
// First step of 2-step feasible efficient GMM.  Weighting matrix = inv(Z'Z)
W = invsym(QZZ)
beta_iv = (invsym(X'Z * W * Z'X) * X'Z * W * Z'Y)'
```

Mata code, continued...

```
// Use first-step residuals to calculate optimal weighting matrix for 2-step FEGMM
omega = m_myomega(beta_iv, Y, X, Z, robust)
// Second step of 2-step feasible efficient GMM
W = invsym(omega)
beta_gmm = (invsym(X'Z * W * Z'X) * X'Z * W * Z'Y)'
// Sargan-Hansen J statistic: first we calculate the second-step residuals
e = Y - X * beta_gmm'
// Calculate gbar = 1/N * Z'*e
gbar = 1/N * quadcross(Z, e)
j = N * gbar' * W * gbar
// Sandwich var-cov matrix (no finite-sample correction)
// Reduces to classical var-cov matrix if Omega is not robust form.
// But the GMM estimator is "root-N consistent", and technically we do
// inference on sqrt(N)*beta.  By convention we work with beta, so we adjust
// the var-cov matrix instead:
V = 1/N * invsym(QZX' * W * QZX)
// Return results to Stata as r-class macros.
st_matrix("r(beta)", beta_gmm)
st_matrix("r(V)", V)
st_numscalar("r(j)", j)
st_numscalar("r(N)", N)
st_numscalar("r(L)", L)
st_numscalar("r(K)", K)
}
end
```

Comparison of HOLS estimation results

```
. hols price mpg headroom, robust
OLS results                                     Number of obs = 74
```

price	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
mpg	-259.1057	66.15838	-3.92	0.000	-388.7737	-129.4376
headroom	-334.0215	314.9933	-1.06	0.289	-951.3971	283.3541
_cons	12683.31	2163.351	5.86	0.000	8443.224	16923.4

```
Sargan-Hansen J statistic:   0.000
. hols price mpg headroom, exog(trunk displacement weight) robust
Heteroskedastic OLS results                     Number of obs = 74
```

price	Coef.	Robust Std. Err.	z	P>|z|	[95% Conf.	Interval]
mpg	-287.4003	60.375	-4.76	0.000	-405.7331	-169.0675
headroom	-367.0973	313.6646	-1.17	0.242	-981.8687	247.6741
_cons	13136.54	2067.6	6.35	0.000	9084.117	17188.96

```
Sargan-Hansen J statistic:   4.795
Chi-sq(  3  )        P-val = 0.1874
```

As shown, this user-written estimation command can take advantage of all of the features of official estimation commands. There is even greater potential for using Mata with nonlinear estimation problems, as its new `optimize()` suite of commands allows easy access to an expanded set of optimization routines. For more information, see Austin Nichols' talk on *GMM estimation in Mata* from Summer NASUG 2008 and Colin Cameron's talk later this morning.

If you're serious about using Mata, you should familiarize yourself with Ben Jann's `moremata` package, available from SSC. The package contains a function library, `lmoremata`, as well as full documentation of all included routines (in the same style as Mata's on-line function descriptions).

Routines in `moremata` currently include kernel functions; statistical functions for quantiles, ranks, frequencies, means, variances and correlations; functions for sampling; density and distribution functions; root finders; matrix utility and manipulation functions; string functions; and input-output functions. Many of these functions provide functionality as yet missing from official Mata, and ease the task of various programming chores.

If you're serious about using Mata, you should familiarize yourself with Ben Jann's `moremata` package, available from SSC. The package contains a function library, `lmoremata`, as well as full documentation of all included routines (in the same style as Mata's on-line function descriptions).

Routines in `moremata` currently include kernel functions; statistical functions for quantiles, ranks, frequencies, means, variances and correlations; functions for sampling; density and distribution functions; root finders; matrix utility and manipulation functions; string functions; and input-output functions. Many of these functions provide functionality as yet missing from official Mata, and ease the task of various programming chores.

In summary, then, it should be apparent that gaining some familiarity with Mata will expand your horizons as a Stata programmer. Mata may be used effectively in either its interactive or function mode as an efficient adjunct to Stata's traditional command-line interface. We have not illustrated its usefulness for text processing problems (such as developing a concordance of words in a manuscript) but it could be fruitfully applied to such tasks as well. To echo my earlier UKSUG talk, **A little bit of Mata programming goes a long way!**

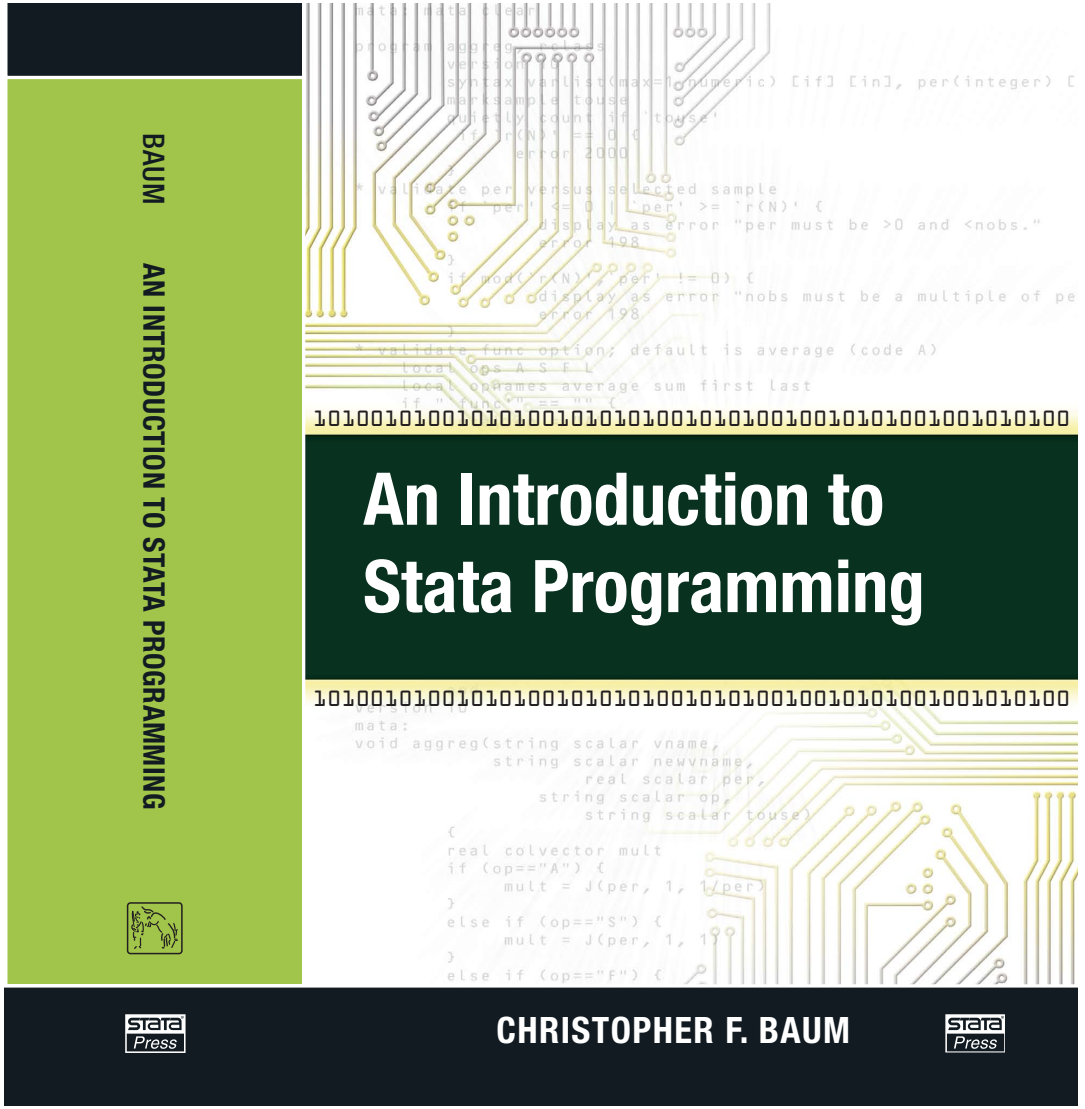
And if you're interesting in learning more about interfacing Mata and Stata...

In summary, then, it should be apparent that gaining some familiarity with Mata will expand your horizons as a Stata programmer. Mata may be used effectively in either its interactive or function mode as an efficient adjunct to Stata's traditional command-line interface. We have not illustrated its usefulness for text processing problems (such as developing a concordance of words in a manuscript) but it could be fruitfully applied to such tasks as well. To echo my earlier UKSUG talk, **A little bit of Mata programming goes a long way!**

And if you're interesting in learning more about interfacing Mata and Stata...

In summary, then, it should be apparent that gaining some familiarity with Mata will expand your horizons as a Stata programmer. Mata may be used effectively in either its interactive or function mode as an efficient adjunct to Stata's traditional command-line interface. We have not illustrated its usefulness for text processing problems (such as developing a concordance of words in a manuscript) but it could be fruitfully applied to such tasks as well. To echo my earlier UKSUG talk, **A little bit of Mata programming goes a long way!**

And if you're interesting in learning more about interfacing Mata and Stata...



See chapters 13–14 of my book, forthcoming this fall from Stata Press.