We now turn to a second way in which you may use Stata programming techniques: by taking advantage of Mata.

Since the release of version 9, Stata has contained 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.
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. 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.
Circumventing the limits of Stata’s matrix language

Mata circumvents the limitations of Stata’s traditional matrix commands. Stata matrices must obey the maximum `matsize`: 800 rows or columns in Stata/IC. 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.
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 onwards 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 labor. 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. Results produced by Mata may then be accessed by Stata and formatted with commands like `estimates display`.
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, 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:

\[ m1 = ( 1, 2, 3 \backslash 4, 5, 6 \backslash 7, 8, 9 ) \]

creates a $3 \times 3$ matrix.

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

\[ m1 = ( 1..3 \backslash 4..6 \backslash 7..9 ) \]
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 \(2 - 3i\) refers to a complex number \(2 - 3 \times \sqrt{-1}\).
The standard algebraic operators plus (+), minus (−) and multiply (*) work on scalars or matrices:

\[
\begin{align*}
g &= r1' + c1 \\
h &= r1 \times c1 \\
j &= c1 \times r1
\end{align*}
\]

In this example \( h \) will be the \( 1 \times 1 \) dot product of vectors \( r1, c1 \) while \( j \) is their \( 3 \times 3 \) 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:

\[ k = r_1 \colon : \ast \ c_1 \]

will produce a three-element column vector, with elements as the product of the respective elements: \( k_i = r_{1i} \ast c_{1i}, \ i = 1, \ldots , 3. \)
Mata’s colon operator is very powerful, in that it will work on nonconformable objects, or what Mata considers c-conformable objects. These include cases where two objects have the same number of rows (or the same number of columns), but one is a matrix and the other is a vector or scalar. 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 (\( >, \geq, <, \leq \)) 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.
Subscripting

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 can 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 can 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:

\[
v = \text{invsym}(xx)[| 2, 2 | ., . |]\]

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:

```plaintext
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:

```c
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 can also use `do`, which follows the syntax

```
do {
    statements
}
```

which will execute the `statements` at least once.

Alternatively, you can use `while`:

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

which could be used, for example, to loop until convergence.
To execute certain statements conditionally, you use `if`, `else`:

```mata
if (exp) statement

if (exp) statement1
    else statement2

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

\[
\begin{align*}
\text{if} \ (k == 0) & \quad \text{dof} = n-1 \\
\text{else} & \quad \text{dof} = n-k
\end{align*}
\]

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 can invoke Mata with a single Stata command, in either interactive mode or in a do-file or ado-file, with

```
.mata: one or more Mata commands, separated by semicolons
```

This context is most useful when you want to operate on one or more items in the Stata workspace, and return the results to the Stata workspace. In doing so, if you create items in Mata’s workspace, they will remain there until you give the command `mata: mata clear`. 
As a first example of the utility of this feature, consider a frequent comment on Statalist. After an estimation command, Stata returns \( e(b) \) and \( e(V) \), the vector of coefficients’ point estimates and their estimated covariance matrix, but does not return the \( t \)-statistics, \( p \)-values, confidence intervals, etc. that appear in the estimation output.

Mata can conveniently be used in this context. As an example, the following code produces a table of coefficients, standard errors and \( t \)-values, using the extended macro function `colnames` to label the columns.
. sysuse auto
(1978 Automobile Data)
. regress price displacement weight foreign

<table>
<thead>
<tr>
<th>Source</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
<th>Number of obs = 74</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>329060336</td>
<td>3</td>
<td>109686779</td>
<td>F( 3, 70) = 25.09</td>
</tr>
<tr>
<td>Residual</td>
<td>306005060</td>
<td>70</td>
<td>4371500.85</td>
<td>Prob &gt; F = 0.0000</td>
</tr>
<tr>
<td>Total</td>
<td>635065396</td>
<td>73</td>
<td>8699525.97</td>
<td>R-squared = 0.5182</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Adj R-squared = 0.4975</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Root MSE = 2090.8</td>
</tr>
</tbody>
</table>

| price       | Coef.     | Std. Err. | t     | P>|t|  | [95% Conf. Interval] |
|-------------|-----------|-----------|-------|------|---------------------|
| displacement| 10.25387  | 6.137677  | 1.67  | 0.099| -1.987346 22.49508  |
| weight      | 2.328626  | .7110004  | 3.28  | 0.002| .91058 3.746671   |
| foreign     | 3899.63   | 678.7615  | 5.75  | 0.000| 2545.883 5253.377  |
| _cons       | -4048.343 | 1432.739  | -2.83 | 0.006| -6905.852 -1190.834|
. mata: st_matrix("bst", (st_matrix("e(b)") \
  > sqrt(diagonal(st_matrix("e(V)")))´) \
  > st_matrix("e(b)") :/ sqrt(diagonal(st_matrix("e(V)")))´))
.
. matrix rownames bst = coeff stderr tstat
. matrix colnames bst = `: colnames e(b)´
. matrix list bst, format(%9.3f)

bst[3,4]

<table>
<thead>
<tr>
<th></th>
<th>displacement</th>
<th>weight</th>
<th>foreign</th>
<th>_cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>coeff</td>
<td>10.254</td>
<td>2.329</td>
<td>3899.630</td>
<td>-4048.343</td>
</tr>
<tr>
<td>stderr</td>
<td>6.138</td>
<td>0.711</td>
<td>678.761</td>
<td>1432.739</td>
</tr>
<tr>
<td>tstat</td>
<td>1.671</td>
<td>3.275</td>
<td>5.745</td>
<td>-2.826</td>
</tr>
</tbody>
</table>
We may also take advantage of an undocumented feature of the `regress` command. Like all estimation (e-class) commands, it returns results in the e-returns. However, it also leaves behind a matrix, `r(table)`, in the r-returns, accessible from `return list`. The first four rows of that matrix contain the estimated coefficients, standard errors, t-statistics and p-values, respectively. We construct a new Stata matrix with a on-line call to Mata.
. qui regress price displacement weight foreign
. mata: st_matrix("bstp", (st_matrix("r(table)"))[(1..4),.])
. matrix rownames bstp = coeff stderr tstat pvalue
. matrix colnames bstp = `: colnames e(b)`
. matrix list bstp, format(%9.3f)

bstp[4,4]

<table>
<thead>
<tr>
<th></th>
<th>displacement</th>
<th>weight</th>
<th>foreign</th>
<th>_cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>coeff</td>
<td>10.254</td>
<td>2.329</td>
<td>3899.630</td>
<td>-4048.343</td>
</tr>
<tr>
<td>stderr</td>
<td>6.138</td>
<td>0.711</td>
<td>678.761</td>
<td>1432.739</td>
</tr>
<tr>
<td>tstat</td>
<td>1.671</td>
<td>3.275</td>
<td>5.745</td>
<td>-2.826</td>
</tr>
<tr>
<td>pvalue</td>
<td>0.099</td>
<td>0.002</td>
<td>0.000</td>
<td>0.006</td>
</tr>
</tbody>
</table>
As a second example of a single-line Mata command, say that we have cross-sectional data on nominal expenditures for 20 hospitals for selected years. For comparability across years, we want to create real (inflation-adjusted) measures. We have a price deflator for health care expenditures, benchmarked at 100 in 2000, for each of these years.
. list exp*, sep(0)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>8.181849</td>
<td>8.803116</td>
<td>9.842918</td>
<td>5.858101</td>
<td>9.190242</td>
</tr>
<tr>
<td>2.</td>
<td>9.23138</td>
<td>9.526636</td>
<td>7.415146</td>
<td>10.4168</td>
<td>8.252012</td>
</tr>
<tr>
<td>3.</td>
<td>7.222272</td>
<td>6.583617</td>
<td>8.960152</td>
<td>6.53302</td>
<td>7.180707</td>
</tr>
<tr>
<td>4.</td>
<td>9.448738</td>
<td>7.972495</td>
<td>8.766521</td>
<td>5.63679</td>
<td>6.681845</td>
</tr>
<tr>
<td>5.</td>
<td>9.649901</td>
<td>6.769967</td>
<td>7.712472</td>
<td>9.241309</td>
<td>9.122219</td>
</tr>
<tr>
<td>6.</td>
<td>6.856859</td>
<td>9.078987</td>
<td>7.766836</td>
<td>8.037149</td>
<td>8.25163</td>
</tr>
<tr>
<td>7.</td>
<td>9.278242</td>
<td>7.264326</td>
<td>8.981012</td>
<td>8.23664</td>
<td>6.625589</td>
</tr>
<tr>
<td>8.</td>
<td>6.841638</td>
<td>7.565176</td>
<td>6.927269</td>
<td>8.598692</td>
<td>10.75326</td>
</tr>
<tr>
<td>10.</td>
<td>7.644565</td>
<td>7.221323</td>
<td>8.915318</td>
<td>8.676471</td>
<td>9.277551</td>
</tr>
<tr>
<td>11.</td>
<td>8.184382</td>
<td>9.080503</td>
<td>8.066475</td>
<td>8.371346</td>
<td>8.339171</td>
</tr>
<tr>
<td>12.</td>
<td>8.145008</td>
<td>9.001379</td>
<td>7.540075</td>
<td>7.305631</td>
<td>8.677146</td>
</tr>
<tr>
<td>14.</td>
<td>7.397709</td>
<td>10.08259</td>
<td>7.648673</td>
<td>7.524302</td>
<td>7.143608</td>
</tr>
<tr>
<td>15.</td>
<td>7.040617</td>
<td>8.258828</td>
<td>7.966858</td>
<td>10.83057</td>
<td>7.402671</td>
</tr>
<tr>
<td>16.</td>
<td>7.861959</td>
<td>7.856915</td>
<td>8.849826</td>
<td>7.502389</td>
<td>8.290931</td>
</tr>
<tr>
<td>17.</td>
<td>7.902071</td>
<td>9.412766</td>
<td>7.739596</td>
<td>7.646762</td>
<td>8.840546</td>
</tr>
<tr>
<td>18.</td>
<td>9.576131</td>
<td>7.904189</td>
<td>9.243579</td>
<td>8.642523</td>
<td>9.428108</td>
</tr>
<tr>
<td>19.</td>
<td>6.432031</td>
<td>8.056722</td>
<td>8.43427</td>
<td>8.755649</td>
<td>8.945447</td>
</tr>
</tbody>
</table>

. loc defl 87.6 97.4 103.5 110.1 117.4
In this example, we use a Mata view, labeled as \( X \), to transform the Stata variables in place. The local macro containing each year’s price deflator, \( \text{defl} \), is transformed into a row vector \( D \) and scaled by 100. The last Mata command uses the colon operator ( : / ) to divide each hospital’s nominal expenditures by the appropriate year’s deflator.

```
> D = 0.01 :* strtoreal(tokens(st_local("defl")))); X[.,.] = X :/ D
```
We rename the Stata variables to reflect their new definitions:

. rename exp* rexp*
. list rexp*, sep(0)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9.34001</td>
<td>9.038107</td>
<td>9.510066</td>
<td>5.32071</td>
<td>7.828145</td>
</tr>
<tr>
<td>3</td>
<td>8.244602</td>
<td>6.75936</td>
<td>8.657151</td>
<td>5.933715</td>
<td>6.116446</td>
</tr>
<tr>
<td>4</td>
<td>10.78623</td>
<td>8.185313</td>
<td>8.470069</td>
<td>5.1197</td>
<td>5.69152</td>
</tr>
<tr>
<td>5</td>
<td>11.01587</td>
<td>6.950685</td>
<td>7.451664</td>
<td>8.393559</td>
<td>7.770204</td>
</tr>
<tr>
<td>6</td>
<td>7.827465</td>
<td>9.321342</td>
<td>7.504189</td>
<td>7.299863</td>
<td>7.028646</td>
</tr>
<tr>
<td>7</td>
<td>10.5916</td>
<td>7.458241</td>
<td>8.677306</td>
<td>7.481053</td>
<td>5.643602</td>
</tr>
<tr>
<td>8</td>
<td>7.810089</td>
<td>7.767121</td>
<td>6.693013</td>
<td>7.809893</td>
<td>9.159508</td>
</tr>
<tr>
<td>9</td>
<td>9.893146</td>
<td>7.726305</td>
<td>9.011443</td>
<td>7.483504</td>
<td>7.738408</td>
</tr>
<tr>
<td>10</td>
<td>8.726672</td>
<td>7.414089</td>
<td>8.613833</td>
<td>7.880537</td>
<td>7.902514</td>
</tr>
<tr>
<td>13</td>
<td>11.00948</td>
<td>9.834132</td>
<td>8.252256</td>
<td>6.749656</td>
<td>7.728137</td>
</tr>
<tr>
<td>14</td>
<td>8.444874</td>
<td>10.35174</td>
<td>7.390022</td>
<td>6.834062</td>
<td>6.084845</td>
</tr>
<tr>
<td>15</td>
<td>8.037233</td>
<td>8.47929</td>
<td>7.697447</td>
<td>9.837034</td>
<td>6.305512</td>
</tr>
<tr>
<td>16</td>
<td>8.974838</td>
<td>8.066648</td>
<td>8.550556</td>
<td>6.814159</td>
<td>7.062121</td>
</tr>
<tr>
<td>18</td>
<td>10.93166</td>
<td>8.115184</td>
<td>8.930994</td>
<td>7.849703</td>
<td>8.030757</td>
</tr>
<tr>
<td>19</td>
<td>7.342501</td>
<td>8.271789</td>
<td>8.149053</td>
<td>7.952451</td>
<td>7.619631</td>
</tr>
<tr>
<td>20</td>
<td>9.393729</td>
<td>9.839087</td>
<td>8.039653</td>
<td>5.886891</td>
<td>8.716402</td>
</tr>
</tbody>
</table>
A note of caution: although Mata code may be interspersed in a do-file or ado-file, you may not place that code inside a `forvalues` or `foreach` loop unless it appears in a Mata single-line command or calls a Mata function.

The following code will not work, as the Mata `end` statement causes an exit from the `forvalues` loop:

```
. forv i=1/5 {
    2. mata
    3. j = log(strtoreal(st_local("i"))); j
    4. end
    —Break—
    r(1);
    end of do-file
    —Break—
    r(1);
```
If the same Mata commands are given as a single-line command, they work properly:

```
. forv i=1/5 {
  2.   mata: j = log(strtoreal(st_local("i")))); j
  3. }
0
  .6931471806
  1.098612289
  1.386294361
  1.609437912
```

If the Mata code was too lengthy to be placed in a single-line command, a Mata function could be defined and called as a single-line command within the loop.
Now let us turn from ‘one-liners’ to the development of Mata functions, which can carry out arbitrarily complex computations as part of an ado-file program.
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.
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, \texttt{st_nobs()}, \texttt{st_nvar()} provide the same information as \texttt{describe} in Stata, which returns \texttt{r(N)}, \texttt{r(k)} in its return list. Mata functions \texttt{st_data()}, \texttt{st_view()} allow you to access any rectangular subset of Stata’s numeric variables, and \texttt{st_sdata()}, \texttt{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.
The statement

\[ st\_view(x, ., \text{varname}, \text{touse}) \]

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

```mata
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 \texttt{Z=\textperiodcentered} specifies that the object is to be created as a void matrix, and then populated with contents. As \texttt{Z} is defined as a real matrix, columns associated with any string variables will contain all missing values. \texttt{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,

```mata
void foo( string scalar varlist )
...
st_view(X=., ., tokens(varlist),OURSE)
```
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:

\[ X = \text{st}_\text{data}(., .) \]

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.
Using views to update Stata variables

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
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 \textit{st\_functions}.

For local macros,

\begin{verbatim}
contents = st_local("macname")
st_local("macname", newvalue )
\end{verbatim}

The first command will return the contents of Stata local macro \textit{macname}. The second command will create and populate that local macro if it does not exist, or replace the contents if it does, with \textit{newvalue}.
Along the same lines, functions \texttt{st\_global}, \texttt{st\_numscalar} and \texttt{st\_strscalar} may be used to retrieve the contents, create, or replace the contents of global macros, numeric scalars and string scalars, respectively. Function \texttt{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 \texttt{r( )} or \texttt{e( )}: Stata’s \texttt{return list} and \texttt{ereturn list}. Functions \texttt{st\_rclear} and \texttt{st\_eclear} can be used to delete all entries in those lists. Read-only access to the \texttt{c( )} objects is also available.

The \texttt{stata( )} function can execute a Stata command from within Mata.
Stata’s local macros can be used to advantage in Mata functions, but with a very important caveat. Macro *evaluation* takes place in Mata as it does in Stata commands with one significant difference.

When a Mata function is first defined, it is compiled into bytecode. In that compilation process, the values of any local (or global) macros are substituted for their names, that is, the macros are evaluated. However, once the function is compiled, those values are hardcoded into the Mata function, just as any other definition would be.
For example,

```
. type gbpval.mata
version 13.1
mata:
real scalar function gbpval(real scalar dollar)
{
    real scalar usdpergbp
    usdpergbp = 2.08
    return( dollar / usdpergbp )
}
end
```

would define a function which converts U.S. dollars to British pounds sterling at a fixed exchange rate of U.S. $2.08 per pound sterling.
Rather than placing that constant in the Mata function, we could type

```
. type gbpval2.mata
local usdpergbp 2.08
mata:
real scalar function gbpval2(real scalar dollar)
{
    return( dollar / `usdpergbp´ )
}
end
```

We can invoke the function with

```
. mata: gbpval2(100) 48.07692308
```
What happens if we change the local macro?

. local usdpergbp 2.06 . mata: gbpval2(100) 48.07692308

Why does Mata ignore the new value of the macro? Unlike Stata commands, which would use the current value of the local macro, the macro’s value has been compiled into the `gbpval2` function, and can only be changed by recompiling that function. This is an important distinction, and explains why you cannot use local macros as counters within Mata functions as you can in ado-files or do-files.
The role of local macros in Mata functions is like that of a header file or .h file in the C language. Local macros in Mata functions serve to define objects, either numeric values or string values, that should be constants within the function.

One useful task that they can serve is the replacement of a lengthy string—such as the URL of a filename on the internet—with a symbol. The value of that symbol is the URL. If the filename changes, you need only update the local macro’s definition rather than change it in several places in the Mata function.
Local macros may also be used to good advantage to define abbreviations for commonly used words or phrases. For instance, we may use the abbreviation RS to avoid repeatedly typing real scalar:

```
. type gbpval3.mata
version 13.1
local RS  real scalar
mata:
`RS´ function gbpval3(`RS´ dollar)
{
    `RS´ usdpergbp
    usdpergbp = 2.08
    return( dollar / usdpergbp )
}
end
```

In summary, local macros can be used in Mata functions, but you must remember that they are expanded only when the function is first compiled by Mata.
We now give a simple illustration of how a Mata subroutine could be used to perform the computations in a do-file. Following our earlier examples, the ado-file `mysum3` takes a variable name and accepts optional `if` or `in` qualifiers. Its purpose is to compute the sum, average, and standard deviation of the variable.

Rather than computing statistics in the ado-file, we call the Mata `m_mysum` function with two arguments: the variable name and the ‘`touse`’ indicator variable.
A simple Mata function

program define mysum3, rclass
    version 14
    syntax varlist (max=1) [if] [in]
    return local varname `varlist´
    marksample touse
    mata: m_mysum("`varlist´", "`touse´")
    return scalar N = N
    return scalar sum = sum
    return scalar mean = mu
    return scalar sd = sigma
end
In the same ado-file, we include the Mata routine, prefaced by the `mata:` directive. This directive on its own line puts Stata into Mata mode until the `end` statement is encountered. The Mata routine creates a Mata view of the variable. A view of the variable is merely a reference to its contents, which need not be copied to Mata’s workspace. Note that the contents have been filtered for missing values and those observations specified in the optional `if` or `in` qualifiers.

That view, labeled as `x` in the Mata code, is then a matrix (or, in this case, a column vector) which may be used in various Mata functions that compute the vector’s descriptive statistics. The computed results are returned to the ado-file with the `st_numscalar()` function calls.
This function is considered a `void` function as it does not return a result when called; rather, it has side effects of defining several scalars in the Stata environment.

```mata
version 14
mata:
void m_mysum(string scalar vname, string scalar touse)

    st_view(X, ., vname, touse)
    mu = mean(X)
    st_numscalar("N", rows(X))
    st_numscalar("mu", mu)
    st_numscalar("sum", rows(X) * mu)
    st_numscalar("sigma", sqrt(variance(X)))

end
```
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:

```stata
program centervars, rclass
    version 14
    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:

```mata
version 14
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.
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 \textit{newvars}. One caveat: Mata’s \texttt{mean()} function performs \textit{listwise deletion}, like Stata’s \texttt{correlate} command.
Let’s consider adding a feature to \texttt{centervars}: the ability to transform variables before centering with one of several mathematical functions (\texttt{abs()}, \texttt{exp()}, \texttt{log()}, \texttt{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 \texttt{centertrans}.
The Stata code:

```stata
program centertrans, rclass
    version 14
    syntax varlist(numeric) [if] [in], ///
        GENERate(string) [TRANS(string)] [DOUBLE]
    marksample touse
    quietly count if `touse'
    if `r(N)' == 0 error 2000
    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 {
        confirm new var `generate’`trans’`v'
    }
    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

```plaintext
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

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

To apply the function, we use

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

which applies the function referenced by `f` to the elements of the matrix `X`. The `Z` matrix is then demeaned as before.
The Mata code:

version 14
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
Presently, the largest and most useful collection of user-written Mata routines is Ben Jann’s `moremata` package, available from the Statistical Software Components Archive. 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. Very importantly, the package also contains the full source code for each Mata routine, accessible with `viewsource`. Unlike ado-file code, which is accessible by its very nature, Mata functions may be distributed in object-code (or function-library) form.
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 that is currently missing from official Mata and ease the task of various programming chores.