The Stata Journal (2005)
5, Number 1, pp. 83–91

Further processing of estimation results: Basic programming with matrices

Ian Watson
ACIRRT, University of Sydney
i.watson@econ.usyd.edu.au

Abstract. Rather than process estimation results in other applications, such as spreadsheets, this article shows how easy it is to process them inside Stata by undertaking some basic programming with matrices. Spreadsheet visualization helps define the task, but the steps are all core Stata: macros, loops, and matrices. The programming challenge is only a modest one for novices, while the benefits of converting do-files into ado programs can be considerable.

Keywords: pr0015, matrices, programming, estimation results

1 Introduction

Many of Stata's commands are able to leave their results behind in matrices. This applies to simple descriptive statistics, such as those produced by `tabulate`, through to the more complex estimation results produced by procedures like `regress` and `logit`. It makes sense for those Stata users who wish to further process their results to become acquainted with the basics of programming with matrices. For those already familiar with Stata's macros and loops, the next step into matrices is only a few keystrokes away. Even for novices, the task of programming with matrices is very accessible. Indeed, this article is pitched at those Stata users who do not think of themselves as programmers at all, but who would benefit from processing their results inside Stata, instead of copying and pasting them into other applications, such as spreadsheets.

Before I came to Stata, I used SPSS and routinely copied and pasted the SPSS output from my logistic regressions into a spreadsheet in order to produce predicted probabilities. I used one column for the coefficients, the next column for the values—sometimes means, sometimes specific values, sometimes a combination of both—and a third column for some formulas to multiply the two other columns together. Finally, there would be a cell with a formula to convert the result to a probability. This approach had many shortcomings: a change to the model specifications, and the whole copy and paste procedure needed to be repeated. Within Stata, this type of exercise was unnecessary: the `adjust` command did all of this and more (see [R] **adjust**). Despite its shortcomings, the spreadsheet exercise did have a positive side. By creating predicted probabilities manually, I was forced to make sure I knew what I was doing. I had to visualize plugging particular values into an equation and calculating a result.

I came to see spreadsheets as a useful starting point for planning further processing of estimation results, but not as a solution. Where a Stata command did not already

exist for the task at hand, manipulating Stata's matrices became the best solution. So while spreadsheets could help with visualizing the goal, the best approach entailed adopting a definite Stata mindset.

This mindset involves some specific Stata conventions. Stata does not use arrays, the grid-type containers favored by many programming languages. It offers matrices instead. But a warning: this article does not discuss the kinds of matrix operations familiar to statisticians, such as matrix multiplication. Rather, this article deals with the much simpler task of multiplying one element in an array (matrix) by another element in an array (matrix). It is this kind of multiplication that is central to the further processing of estimation results and that mimics multiplying the cells in one column by those in another column within a spreadsheet.

Two other Stata conventions are macros and loops. Like Stata's matrices, macros serve as containers. In the example used in this article, they are the tools for looping and for storing final results. In many cases scalars could be used, but for simplicity, macros are employed. The looping processes are managed with Stata's `forvalues` command (see [P] **forvalues**).

A simple example will be used to illustrate the basics of programming with matrices. It is a poor person's `adjust` in which the mean values of the variables in a regression are plugged into the regression equation to produce a predicted outcome, similar to the SPSS example mentioned earlier. To visualize this in terms of a spreadsheet: the estimation results are a column of coefficients. Next to them is another column, populated with means, and then in a third column is a set of calculations, the product of the coefficient and the mean. Finally, there will be a cell at the bottom which sums the products and provides a prediction.

To convert this spreadsheet visualization into a Stata program entails a number of steps:

1. obtaining the estimation data;
2. storing it in a useful form;
3. stepping through the data to do some calculations on it;
4. displaying the results; and
5. abstracting the code to make it more generalizable.

In this article, I will illustrate this program as a do-file and then at the end, turn it into an ado-file. I will use Stata's full names for commands to make it more comprehensible, although most programmers use the abbreviated command names. Similarly, some of the macro names are longer (and more readable) in the do-file example. I will also carry out some of the steps in a long-winded fashion but will include more efficient code at the end of the article.

2 Obtaining the data

The starting point is a simple regression:

```
. sysuse auto
(1978 Automobile Data)
. regress price weight length foreign
```

(partial output)

Source	SS	df	MS
Model	348565467	3	116188489
Residual	286499930	70	4092856.14
Total	635065396	73	8699525.97

price	Coef.	Std. Err.	t	P>|t|
weight	5.774712	.9594168	6.02	0.000
length	-91.37083	32.82833	-2.78	0.007
foreign	3573.092	639.328	5.59	0.000
_cons	4838.021	3742.01	1.29	0.200

When you ask for the results from this regression, using `ereturn list`, you obtain quite a lot of information, but the key piece you need is the matrix of coefficients `e(b)`, and it is this which you obtain and place in a matrix.

```
. ereturn list                                              ([P] page 149)
scalars:
                  e(N) =  74
              e(df_m) =  3
              e(df_r) =  70
                  e(F) =  28.38811943068356
                 e(r2) =  .5488654690386703
               e(rmse) =  2023.080852875841
                e(mss) =  348565466.5125227
                e(rss) =  286499929.6090988
               e(r2_a) =  .5295311319974705
                 e(ll) =  -666.2612504003735
              e(ll_0) =  -695.7128688987766
macros:
            e(depvar) : "price"
               e(cmd) : "regress"
           e(predict) : "regres_p"
             e(model) : "ols"
matrices:
                  e(b) :  1 x 4
                  e(V) :  4 x 4
functions:
            e(sample)
. matrix coeffs=e(b)
. matrix list coeffs
coeffs[1,4]
       weight     length    foreign      _cons
y1  5.7747117  -91.37083  3573.0919  4838.0206
```

The column of coefficients is displayed horizontally as a row vector. To recast it in the more familiar Stata output form and to match the spreadsheet visualization, you need to flip it around. The matrix transpose operator does this:

```
. matrix coeffs = coeffs'                                    ([P] page 269)
. matrix list coeffs
coeffs[4,1]
                y1
 weight  5.7747117
 length  -91.37083
foreign  3573.0919
  _cons  4838.0206
```

The next step involves preparing your "containers"—the imaginary spreadsheet columns into which you will place the means and your calculations. There are various ways to do this, some more efficient than others. For heuristic purposes, I will create two new column vectors, although in practice I would probably work with one matrix made up of two columns. (This is the approach shown in the sample ado-file at the end of the article.)

I create each vector with the same number of rows as the coefficient vector, using the handy `J()` function, which creates a matrix of specified dimensions and contents. I also obtain the names of the rows to be used for calculating the means.

```
. local varnames : rownames(coeffs)                          ([P] page 293)
. local k = rowsof(coeffs)                                   ([P] page 275)
. matrix means = J(`k',1,1)                                  ([P] page 271)
. matrix calcs = J(`k',1,1)
. matrix list means
means[4,1]
    c1
r1   1
r2   1
r3   1
r4   1
```

3 Stepping through

At this stage, the data have been captured from the regression, and the "containers" are set up, ready to be filled. The next part of this example consists of stepping through the second vector and filling it with means, and stepping through the third vector and filling it with the product of the calculations. The `forvalues` loop is used for these stepping processes. Because this is probably the most complicated part of the process, I have broken it into two stages. Normally, for efficiency, you would combine obtaining the means and multiplying them by the coefficients into one stage (this is illustrated in the sample ado-file).

```
. local k1 = `k'-1
. forvalues r = 1/`k1'{
  2.   local varnm: word `r' of `varnms'
  3.   quietly sum `varnm' if e(sample)
  4.   local varmn = r(mean)
  5.   matrix means[`r',1] = `varmn'
  6. }
. matrix list means
means[4,1]
           c1
r1  3019.4595
r2  187.93243
r3   .2972973
r4          1
```

There are two issues worth noting here. First, the independent variable names (which were captured earlier) are stored in `varnames` and are used to retrieve the means using the `summarize` command. The `meanonly` option is specified because it produces faster code.

Second, it is only the independent variables that are used at this stage—the constant is left out—and this is done by using one row fewer than the total number of rows. This entails using the macro `k1` rather than `k`. Leaving the constant in would produce an error message from Stata's `summarize` command when it tried to find a variable called `_cons`.

The next stage involves stepping through the third vector and carrying out the calculations at each step. The running total will be stored in a macro, `total`. Notice that this time the stepping involves all four rows, using `k` rather than `k1`. This is done in order to pick up the constant. Because the value of its 'mean' has been left at 1, the result in the calculations vector is simply the value of the constant in the coefficient vector. If you were doing a different kind of calculation, which involved transforming the constant in some fashion, you would need to modify either this stage, or the earlier stage when it was left out. (A useful device is to set the value of empty matrices—see the `J()` function used earlier—to some value, such as -1, that you can test against during these loops.)

```
. forvalues r = 1/`k'{
  2.   matrix calcs[`r',1] = coeffs[`r',1]*means[`r',1]    ([P] page 277)
  3.   local total = `total'+calcs[`r',1]
  4. }
. matrix list calcs
calcs[4,1]
            c1
r1   17436.508
r2  -17171.542
r3   1062.2706
r4   4838.0206
. display `total'
6165.2568
```

It is essential to check your results manually when you first start programming matrices. You can either do this in a spreadsheet or make use of one of Stata's own commands (if available). In this case, there is `adjust`:

```
. adjust weight length foreign

    Dependent variable: price     Command: regress
Covariates set to mean: weight = 3019.4595, length = 187.93243, foreign = .2972973

------------------------
      All |         xb
----------+-------------
          |    6165.26
------------------------

     Key:  xb  =  Linear Prediction
```

4 Displaying results

Displaying the results depends on your purposes. If you just need to check your results, you will probably find the `matrix list` command suits, as well as `display` for any of your macro results. The results from this example are spread across three vectors, so they need to be combined into one matrix (called `final` in this example) for display purposes. This is shown below, and it is worth noting that the new matrix inherits the row names from the first matrix, although you can choose to rename these rows using `matrix rownames`. You can also name the columns with meaningful labels, using `matrix colnames`. Finally, you can display the single combined matrix, and the macro with the running total:

```
. matrix final = coeffs,means,calcs                          ([P] page 269)
. matrix colnames final = Coeffs Means Calcs                 ([P] page 293)
. matrix list final, format(%9.3f)
final[4,3]
            Coeffs      Means      Calcs
 weight      5.775   3019.459  17436.508
 length    -91.371    187.932  -1.72e+04
foreign   3573.092      0.297   1062.271
  _cons   4838.021      1.000   4838.021
. display "Prediction: " %9.3f `total'
Prediction:  6165.257
```

If your goal is to craft an ado program for continuing use, you might improve the presentation of the results. The Stata `display` command has many neat features and, when combined with SMCL, can produce more pleasing output. The ado code at the end of this paper illustrates this approach.

5 Abstraction—generalizing your code

At this stage, you still have a do-file, which essentially replicates what you would have done in a spreadsheet, but more efficiently. Turning it into an ado file involves just a few more steps. First, you need to add a header (`program spreadsht`) and a footer (`end`), and rename the file from `spreadsht.do` to `spreadsht.ado`. You might also strip out the various `matrix list` commands so that you are not displaying unnecessary output.

The more important steps, however, involve making your code more abstract so that it can deal with different variables. This entails passing arguments from the command line in Stata to the code in your ado-file so that it executes your code against whatever particular variables it suits you to use. You might also want to incorporate weighting and `if` and `in` conditions. All of these are demonstrated in this final section. Code similar to the following needs to be inserted at the top of your file:

```
program spreadsht
version 8.2
syntax varlist [if] [in] [fweight aweight iweight]        ([P] syntax)
tempvar touse
mark `touse' `if' `in'                                    ([P] mark)
regress `varlist' if `touse' [`weight' `exp']
and so on ...
end
```

Dealing with the `if` and `in` conditions is handled by the `mark` command. First, the code creates a temporary variable and stores the name in the macro `touse`. Next, `mark` fills this variable with 0s and 1s, depending on whether the observation is to be included or not, that is, whether it meets the conditions specified by `if` and `in`. Marking the sample in this way ensures that `if` and `in` conditions are maintained throughout the program by establishing the relevant sample at the outset. This is one of the reasons for using Stata's very useful `syntax` statement: it captures the `if` and `in` conditions that the user types and stores them in the macros `` `if' `` and `` `in' ``. These macros are then used in conjunction with the `mark` command to ensure that only the appropriate observations are included in the `touse` macro.

Now all that you need to do later is to issue your command with the expression `if `touse'` attached to the command name and you can be sure that the appropriate sample specified by the user is used in all subsequent processing (in this case, `regress`).

Weighting is also simple. Again, by making use of the `syntax` statement you can retrieve what the user entered. The macro `` `weight' `` (which contains the weight type) and the macro `` `exp' `` (which contains the equals sign, a space and the expression) can be combined within your code as `[`weight' `exp']`. When your program runs, this expression gets expanded to something like `[aweight = wt]`.

More often than not, your program will have bugs, and you will need to debug it. Usually the cause is quite simple: a typo; a missing ` or ' around a macro; a missing brace; or a failure to include a string quote (") where needed. To make life somewhat easier for programmers, Stata provides some useful tools. You can issue a `set trace on`

command, run your program, and then watch to see if it performs as you intend. It is worth issuing a `set tracedepth 1` command at the beginning, as this stops Stata going too deeply into its own code. When you are developing your program, sprinkle `mat list mymatname` commands liberally through your code so that you can check on the contents of the matrices. You will obviously remove these statements when the program is debugged.

My intention in this article has been to emphasize the advantages of spreadsheet visualization, followed by matrix programming, to carry out the further processing of estimation results. Further progress on building ado programs can be pursued with Stata's excellent programming manual or by simply peering under the bonnet of the numerous user ado-files available.

6 Acknowledgment

This article has benefited considerably from the comments of Nicholas J Cox.

7 Sample do-file

```
sysuse auto, clear
regress mpg weight length foreign
ereturn list
matrix coeffs = e(b)
matrix list coeffs
matrix coeffs = coeffs'
matrix list coeffs
local varnames : rownames(coeffs)
local k = rowsof(coeffs)
matrix means = J(`k',1,1)
matrix list means
matrix calcs = J(`k',1,1)
local k1 = `k'-1
forvalues r = 1/`k1'{
  local varname: word `r' of `varnames'
  quietly sum `varname' if e(sample), meanonly
  matrix means[`r',1] = r(mean)
}
matrix list means
local total = 0
forvalues r = 1/`k'{
  matrix calcs[`r',1] = coeffs[`r',1] * means[`r',1]
  local total = `total' + calcs[`r',1]
}
matrix list calcs
display `total'
adjust weight length foreign
matrix final = coeffs, means, calcs
matrix colnames final = Coeffs Means Calcs
matrix list final, format(%9.3f)
display "Prediction: " %9.3f `total'
```

8 Sample ado-file

```
program spreadsht
version 8.2
syntax varlist [if] [in] [fweight aweight iweight]
tempvar touse
mark ‘touse’ ‘if’ ‘in’
regress ‘varlist’ if ‘touse’ [‘weight’ ‘exp’]
mat coeffs = e(b)
mat coeffs = coeffs’
local varnms : rownames(coeffs)
local k = rowsof(coeffs)
mat results = J(‘k’,2,1)
local total = 0
forval r = 1/‘k’{
  local varnm: word ‘r’ of ‘varnms’
  if ‘r’ < ‘k’ {
    quietly sum ‘varnm’ if e(sample) [‘weight’ ‘exp’], meanonly
    mat results[‘r’,1] = r(mean)
  }
  mat results[‘r’,2] = coeffs[‘r’,1] * results[‘r’,1]
  local total = ‘total’ + results[‘r’,2]
}
mat final = coeffs, results
di
di "{title:Results of calculations}"
di
di as text "{hline 13}{c TT}{hline 40}"
di as text "{ralign 12: Variable} {c |} {ralign 13: Coefficient}" /*
     */ "{ralign 12: Mean} {ralign 13: Calculation}"
di in text "{hline 13}{c +{}hline 40}"
forval r = 1/‘k’ {
  local varnm: word ‘r’ of ‘varnms’
  local varnm=abbrev("‘varnm’",15)
  di as text "{ralign 12:‘varnm’} {c |} {col 20}" /*
     */ as result %9.3f final[‘r’,1] "{col 32}" /*
     */ as result %9.3f final[‘r’,2] "{col 46}" /*
     */ as result %9.3f final[‘r’,3]
}
di as text "{hline 13}{c BT}{hline 40}"
di
di as text "Prediction: " as result %9.3f ‘total’
end
```

About the Author

Ian Watson is Senior Researcher at the Australian Centre for Industrial Relations Research and Training (ACIRRT), University of Sydney.