The Stata Journal (2012)
**12**, Number 4, pp. 748–758

# Speaking Stata: Matrices as look-up tables

Nicholas J. Cox
Department of Geography
Durham University
Durham City, UK
n.j.cox@durham.ac.uk

**Abstract.** Matrices in Stata can serve as look-up tables. Because Stata will accept references to matrix elements within many commands, most notably `generate` and `replace`, users can access and use values from a table in either vector or full matrix form. Examples are given for entry of small datasets, recoding of categorical variables, and quantile-based or similar binning of counted or measured variables. In the last case, the device grants easy exploration of the consequences of different binning conventions and the instability of bin allocation.

**Keywords:** pr0054, matrices, vectors, data entry, recoding, quantiles, binning, look-up tables

## 1 Introduction

To many new users, and even to many more experienced users, the main data model of Stata no doubt seems uncompromisingly simple and severe. You may have precisely one dataset, a rectangular block of observations and variables, in memory at a time. Compared with, say, the engaging generosity of spreadsheets—put what you want where you want it, subject only to some size limitations—that standard appears Spartan. As usual, first appearances can be deceptive. Detailed acquaintance shows that Stata's facility for combining datasets, by use of `append`, `merge`, and other commands, imparts greater flexibility than is initially evident. Further, Stata offers a variety of facilities for storing both metadata, such as variable and value labels, and the results (numerical, textual, graphical) of various commands.

In this column, I focus on yet another extension of the main data model. Stata offers two ways of handling matrices (and, in the case of Mata, other data structures, too) within Stata sessions: its own matrix language and Mata as an embedded programming language. This column focuses on the use of Stata matrices as look-up tables, with a few asides on Mata. Various little techniques offer some ways of working with small data tables auxiliary to the main dataset. In each case, there are alternatives, but in each case the simplicity and generality of the approach can be refreshingly different and appealing. The approach is not intended to be systematic or comprehensive; the scope for using matrices in any mathematical or statistical language is, in practice, limitless. At most, the intent is to provoke interest in, and use of, a device that is sometimes overlooked.

Mata is now a much richer and deeper language than Stata's matrix language, and the difference will only get greater. A theme underlying this column is that nevertheless there remains much scope for applying the latter. That is particularly important for users who lack the time or inclination to start learning Mata. Although no user-written program is used here, note that various user-written programs have been written to go with Stata's matrix language (Weesie 1997; Cox 1999, 2000).

## 2 A prime example, and entering small datasets generally

Consider first Stata matrices as a way to input small datasets by hand. There are several ways to do that, the choice being largely a matter of taste or convenience. You could use `input` or the Data Editor `edit` command, the latter either directly or by copying and pasting from some other application. Here is the matrix way exemplified, assuming no data are currently in memory.

```
. matrix mydata = (2,3,5,7,11,13,17,19,23,29)´
. set obs `=rowsof(mydata)´
obs was 0, now 10
. generate myvar = mydata[_n, 1]
. list, sep(0)

      ┌───────┐
      │ myvar │
      ├───────┤
  1.  │     2 │
  2.  │     3 │
  3.  │     5 │
  4.  │     7 │
  5.  │    11 │
  6.  │    13 │
  7.  │    17 │
  8.  │    19 │
  9.  │    23 │
 10.  │    29 │
      └───────┘
```

The matrix here is first typed as a row vector. That is what the commas imply; in this example they join scalars columnwise. That row vector is promptly transposed to a column vector with the prime operator `'`. The mode of entry in the example is partly a personal quirk, because I prefer to use commas rather than the backslashes that join rows, but more importantly it shows some technique. You may use whichever style of entry you prefer, because at most a keystroke is needed to transpose.

The `set obs` command just given condenses two commands in one. We could in this example count the number of values directly and type `set obs 10`. We can also insist that Stata do the counting. The single command `set obs '=rowsof(mydata)'` combines the two steps

```
. local n = rowsof(mydata)
. set obs `n´
```

and indeed cuts out any need to create and use that local macro. Its name makes clear that `rowsof()` is a function that counts rows of matrices. We just entered a column vector, which to Stata is just a special case of a matrix. That is not surprising, although a quirk of Stata's matrix language is that you must always specify a column subscript 1 when referring to elements of a column vector. Similarly, you must always specify a row subscript 1 when referring to elements of a row vector. (Mata, in contrast, happily indulges single subscripts for referring to vectors.)

It is evident, if only from the `list` results that follow, that the `generate` command copies the values just entered to a new variable. Precisely how this works is key. As always, the principle is that Stata calculates the values of a new variable observation by observation, so `generate` necessarily loops automatically over observations. In this example, the observation number `_n` will vary from 1 to 10. (The `set obs` command was needed beforehand to make that so.) So the expression `mydata[_n, 1]` is evaluated in turn as `mydata[1, 1]`, `mydata[2, 1]`, and so forth, and the result is copied across—element by element, observation by observation—to the new variable.

The principle of such data input using a matrix extends easily to small datasets with a few variables, not just one. In developing and testing a program to calculate people's ages and other differences between daily dates, I needed a sandpit of sample dates to play in. (The solution to that problem is a little more complicated than just subtraction, given the occurrence of leap years; that is why I was writing a program to do this.)

```
. clear
. matrix values = (28,19,28,29,29\3,11,2,2,7\1952,1952,2011,2012,2012)
. set obs `=colsof(values)´
obs was 0, now 5
. generate bdate = mdy(values[2, _n], values[1, _n], values[3, _n])
. format %td bdate
. list

     ┌───────────┐
     │     bdate │
     ├───────────┤
  1. │ 28mar1952 │
  2. │ 19nov1952 │
  3. │ 28feb2011 │
  4. │ 29feb2012 │
  5. │ 29jul2012 │
     └───────────┘
```

Here the columns define observations, rather than the rows defined in the first example. No matter: we just use `colsof()` instead of `rowsof()` and switch subscripts. An earlier version of the code segment just given was not quite so slick (earlier versions rarely are). I originally defined separate variables for month, day, and year, and fed the three variables to the function `mdy()`; then I realized that the function could work directly on the elements of the matrix.

If you are familiar with the `svmat` command, you will know that this command offers an alternative method for putting a Stata matrix into variables. Please bear with me, because the approach here has applications and extensions that make it more widely useful.

## 3 Some recoding examples

Let's revisit a standard problem, recoding a categorical variable that is not suitably coded for a particular purpose. Although nothing in Stata makes it compulsory, we will assume a preference for codes that are consecutive integers, most obviously 1 and above, but possibly a shifted sequence such as −2, −1, 0, 1, 2. Unless the codes have some inherent meaning (in which case is the variable really categorical?), using consecutive integers causes no problems, while other codings can look untidy. For the most part, the codings should be hidden below value labels. However, there are contexts, most notably `twoway` graphs, where irregular integer sequences will be all too evident. Henceforth in this section, "integer sequence" will be short-hand for an increasing sequence of consecutive integers.

Stata's `auto.dta` includes a well-defined ordinal or graded variable, repair record `rep78`, but there are few observations of cars with grades 1 and 2. That is good news for car owners. But for some statistical purposes, we would think, say, of collapsing 1 and 2 into one category and renumbering to ensure a tidy order.

```
. sysuse auto, clear
(1978 Automobile Data)
. tabulate rep78
     Repair
Record 1978 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |          2        2.90        2.90
          2 |          8       11.59       14.49
          3 |         30       43.48       57.97
          4 |         18       26.09       84.06
          5 |         11       15.94      100.00
------------+-----------------------------------
      Total |         69      100.00
```

A matrix approach to solve this problem requires only a vector of new codes. This example is especially easy, because the existing codes run 1 to 5, and so they can be implicit as column or row indexes. The mapping 1 to 1, 2 to 1, 3 to 2, 4 to 3, 5 to 4 is concisely defined as, say, a row vector that we can use in a one-line `generate` statement. (Where I say index, some people say subscript, given the common use of subscript notation in matrix algebra to refer to matrix elements.)

```
. matrix newcodes = 1,1,2,3,4
. generate rep78_2 = newcodes[1, rep78]
(5 missing values generated)
```

Let's take that more slowly and emphasize details that are crucial for full understanding.

1. We put a matrix-like expression on the right-hand side of the `generate` command, but Stata will evaluate it observation by observation. That is, Stata will attempt to interpret the expression as referring to a matrix element. (Conversely, offering an entire matrix to `generate` will fail, even if exceptionally the matrix is $1 \times 1$, a scalar dressed up as a matrix.)
2. It is immaterial that `newcodes` is a $1 \times 5$ vector and that `rep78` and other variables in the dataset have 74 observations. The data entry examples in the previous section had the same number of rows or columns as there were to be observations. Such equality was entirely natural in that context, but it is not essential here.
3. It so happens that there are missing values in `rep78`, but that is taken care of without a need on our part for evasive or corrective actions. So long as a matrix exists, it is quite legal to refer to an element of that matrix that does not exist, because the row or column index is outside the range; or even to refer to one that cannot exist because the expression supplied for the index evaluates to zero, or a negative value, or missing. It is just that the result returned by such a reference will be missing.
4. Whenever `rep78` is equal to 1, the element used is `newcodes[1,1]`, and similarly for other values of `rep78`. The values of `rep78` in `auto.dta` run 3 3 `.` (missing) 4 3 (first five) through to 4 5 4 4 5 (last five). It is as if you are asking Stata to construct a vector of the same length as the dataset using those columns of the vector. The order of the chosen elements is what you say it is: repetitions, omissions, and impossible indexes (as just explained) are all fine. Stata will do that element by element. The "as if" here is, however, the essence of the Mata approach to the same question.

We can check with the `groups` command (Cox 2003b) whether values align one-to-one as they should. (Download the latest version of `groups` by using `ssc install groups, replace`.)

```
. groups rep78*

  +------------------------------------+
  | rep78   rep78_2   Freq.   Percent  |
  |------------------------------------|
  |     1         1       2      2.90  |
  |     2         1       8     11.59  |
  |     3         2      30     43.48  |
  |     4         3      18     26.09  |
  |     5         4      11     15.94  |
  +------------------------------------+
```

This approach has its downside, too. We would still need to take care of any value labels, which a command such as `recode` is willing to organize for us. Further, the technique at its simplest (as it is here) depends on the values being consecutive integers from 1 and above. What if the problem were the other way around so that we wanted to map, say, 2, 3, 5, 7, 11 to 1, 2, 3, 4, 5? We could set up a vector by typing

```
. matrix newcodes = 0,1,2,0,3,0,4,0,0,0,5
```

The idea used here is that the 2nd, 3rd, 5th, 7th, and 11th elements of `newcodes` give the new values of a variable, so we can map *oldvar* to *newvar* with the code

```
generate newvar = newcodes[1, oldvar]
```

It does not matter what goes in the other elements of `newcodes`. As long as *oldvar* is never any value other than 2, 3, 5, 7, or 11, those elements will never be used and could be anything legal; here 0 was used.

That said, this solution is now too tricky to be a serious proposal. It is tricky even for this example and does not extend well to more complicated examples. It will not work at all with recoding zero or negative integers, because they cannot be row or column indexes.

The look-up table approach is easily workable for more general problems so long as we are willing to loop over possibilities. In this column, modest familiarity is presumed with loops in Stata. If you need a tutorial, see a previous column (Cox 2002).

We will take this in two steps. First, if you want a mapping to integers in sequence, you can do something like this:

```
generate newvar = .
matrix oldcodes = 2,3,5,7,11
forvalues i = 1/`=colsof(oldcodes)´ {
      replace newvar = `i´ if oldvar == oldcodes[1, `i´]
}
```

A little thought or experiment makes it clear that this is quite general, at least for one-to-one mappings. (In practice, you might want to add some `quietly` prefixes to suppress return messages.)

1. The values of `oldcodes` do not need to be in order. It is thus easy to reorder the categories of a variable.
2. The loop is from 1 to the number of columns in the row vector, or the number of rows if we entered a column vector. The result of this code is that the values of the new variable are the same integer sequence, 1 and above. But any other integer sequence can be obtained by adding or subtracting a constant, either within the loop or after it.
3. The variable *newvar* is born missing and will remain so if *oldvar* takes any value other than those specified. This will usually be what is wanted.
4. The code is its own concise documentation. We could use `matrix list` to leave a clear record of what we did. An audit trail is always important but especially so for data manipulations.

For more arbitrary mappings, there is a fall-back device that is only a little more complicated.

```
generate newvar = .
matrix codes = [2,3,5,7,11\1,1,2,3,4]
forvalues i = 1/`=colsof(codes)´ {
      replace newvar = codes[2, `i´] if oldvar == codes[1, `i´]
}
```

Looked at differently, this technique is another solution to the problem of looping over parallel lists (Cox 2003a). You can store the lists in a matrix and then loop over the row or column index with `forvalues`.

The point of these examples is not to persuade you to change how you recode categorical variables. If you do that a lot, you are likely to be familiar with `recode`, which in a strong sense is optimized for the purpose. `recode` has added bells and whistles for handling complications and auxiliary tasks; it is likely to be especially appealing for interactive use. The point is just to give concrete examples of other ways to do it. Programmers will note that the approach here will often be easy to automate.

## 4 Quantile examples, and binning generally

Another standard problem is cutting the range of a variable into categories according to the values of certain quantiles, whether the variable's own quantiles or quantiles supplied from somewhere else. For example, if you have lower quartile, median, and upper quartile (25%, 50%, and 75% quantiles or percentiles), then you can code a new variable that is 1, 2, 3, or 4 according to whether values lie below the lower quartile (1) through to whether they lie above the upper quartile (4). Note here that three boundaries give rise to four classes or bins; the upper limit of the top class is just the maximum observed value, and the lower limit of the bottom class is just the minimum observed value. So for $k$ bins, we seek just $k - 1$ "internal" boundaries.

Quantile-based bins can provide simple frameworks for analysis that are easy to understand and easy to explain to others, including lay audiences. The last may help to explain the widespread use of such an idea in looking at the performance of firms in economics and business. Commonly, such bins span equal intervals on a cumulative probability scale, so in principle, the number of values falling into each should be the same. The example of median and quartiles is a case in point.

Indeed, we can imagine any kind of subdivision according to disjoint intervals, which need not be formally defined anywhere by quantiles. Classifications like this are common in all quantitative fields. Often they may be criticized because the cutoffs concerned are arbitrary or the classification is an unnecessary coarsening of data. In practice, there may be standard schemes beyond researchers' control that are firmly entrenched, regardless of their merits. Classifications of net income by tax brackets or of body mass index according to implicit degrees and kinds of risk are two of many examples.

In this section, I will maintain a focus on quantiles. It will be easy to see how the ideas extend to any set of bins defined by their boundaries.

Stata has a bundle of related commands in this area, including `pctile`, `_pctile`, and `xtile`. Let's start with `xtile` and look at the `age` variable in `nlswork.dta` that may be downloaded from StataCorp's website. This dataset includes data on 28,510 women from the U.S. National Longitudinal Survey.

```
. webuse nlswork, clear
(National Longitudinal Survey.  Young Women 14-26 years of age in 1968)
```

As is standard, `age` is reported in years. Applying `tabulate` to age reveals a very uneven distribution of ages from 14 to 46 within the dataset, with a mode at 24 years of 1,636 women. There is just one woman aged 15, two women aged 46, and three women aged 14. Although the ages can be used in many analyses precisely as recorded, this is clearly also a situation in which researchers might seek a categorization with bins that are equally populated. Choosing 10 bins, we can get such a categorization with the following command:

```
. xtile q_age = age, nquantiles(10)
```

Perhaps surprisingly, `xtile` does not display the quantiles it uses, nor does it leave them even temporarily in memory once it is done. However, the associated command `_pctile` can be used for this purpose.

```
. _pctile age, nquantiles(10)
. return list
scalars:
                 r(r1) =  21
                 r(r2) =  23
                 r(r3) =  24
                 r(r4) =  26
                 r(r5) =  28
                 r(r6) =  31
                 r(r7) =  33
                 r(r8) =  36
                 r(r9) =  38
```

Recall that the number of quantiles determined is one fewer than the number of bins. Such r-class results will be overwritten by the next r-class command, so if we want to keep an eye on them, we should put them somewhere safer. A vector is one such place. The `matrix` command understands about `r()` results, so we can type

```
. matrix q = r(r1), r(r2), r(r3), r(r4), r(r5), r(r6), r(r7), r(r8), r(r9)
```

In a large dataset, a variable containing only integer values will in practice be likely to have quantiles that are also integers. In a smaller dataset, it will be more common that the quantiles reported are interpolated values with fractional parts. This consideration brings to the fore the question of what `xtile` does when confronted with ties. A tabulation of `q_age` shows that frequencies are far from equal:

```
. tabulate q_age
         10 
  quantiles 
     of age |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |      4,122       14.46       14.46
          2 |      3,062       10.74       25.20
          3 |      1,636        5.74       30.94
          4 |      2,980       10.45       41.39
          5 |      2,567        9.00       50.39
          6 |      3,614       12.68       63.07
          7 |      2,357        8.27       71.34
          8 |      3,543       12.43       83.76
          9 |      1,824        6.40       90.16
         10 |      2,805        9.84      100.00
------------+-----------------------------------
      Total |     28,510      100.00
```

Researchers are often surprised at such inequality, but when ties are common, it is very likely. To spell out a rule that is used, it is vital that identical values are treated identically. The manual entry on `xtile` (see [D] **pctile**) makes it clear that its rule at boundaries is that upper limits are inclusive. So for example, bin 1 is all values less than or equal to the first quantile (decile, in this case). We should test our understanding by trying to reproduce that.

```
. generate q_age2 = 10 if age < .
(24 missing values generated)
. forvalues i = 9(-1)1 {
  2.         quietly replace q_age2 = `i´ if age <= q[1, `i´]
  3. }
```

A detail to note: We need to watch for missing values. Given the Stata rule that numeric missing is treated as larger than any nonmissing value, that also applies to any of the quantiles. If you want an exercise, you might like to work out code for classifying ages from the bottom upward, not the top downward as here.

We need to check that the two variables are identical:

```
. assert q_age == q_age2
```

No news is good news. Silence implies consent: the two variables are identical. But can we do better than `xtile` without cheating? The largest single class is the lowest; if we were to use the convention $<$ rather than $\leq$, we should be able to reduce that frequency, but how will that convention work elsewhere? `xtile` does not offer this option, but we can implement the rule for ourselves by repeating the code for `q_age2` but with a new variable name and a new operator inside the loop:

```
. generate q_age3 = 10 if age < .
(24 missing values generated)
. forvalues i = 9(-1)1 {
  2.        quietly replace q_age3 = `i´ if age < q[1, `i´]
  3. }
. tabulate q_age3
     q_age3 |      Freq.     Percent        Cum.
------------+-----------------------------------
          1 |      2,805        9.84        9.84
          2 |      2,775        9.73       19.57
          3 |      1,604        5.63       25.20
          4 |      3,202       11.23       36.43
          5 |      2,731        9.58       46.01
          6 |      3,662       12.84       58.85
          7 |      2,314        8.12       66.97
          8 |      3,677       12.90       79.87
          9 |      2,067        7.25       87.12
         10 |      3,673       12.88      100.00
------------+-----------------------------------
      Total |     28,510      100.00
. count if q_age != q_age3
11647
```

About 41% of values changed bin when we changed the convention about what to do when values tie with boundaries! Can either binning be said to be better?

A criterion for the equality of the bin numbers is the standard deviation of the bin frequencies. One way to get those for the two binned variables is to push the frequencies into a matrix, push the matrix into a variable, and then fire up `summarize` as usual. That is a little quick and dirty, but it works. (There naturally are other ways to do it, notably by doing more in Mata.)

```
. quietly tabulate q_age, matcell(freq)
. quietly tabulate q_age3, matcell(freq3)
. generate freq = freq[_n,1]
(28524 missing values generated)
. generate freq3 = freq3[_n,1]
(28524 missing values generated)
. summarize freq*
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        freq |        10        2851    788.4865       1636       4122
       freq3 |        10        2851    716.4114       1604       3677
```

By this criterion, the second binning has improved the degree of inequality.

This little exercise has shown two things, one positive and one negative. Positively, once we have values stored in a matrix as a look-up table, there is plenty of scope to do something different with basic Stata commands. Negatively, we have seen how capricious binning can be, given only a change of convention at boundaries. One clear implication is to report carefully the rule for binning so that other researchers can repeat your results.

## 5 Conclusion

Matrices in Stata can serve as look-up tables. You can first store and then access key values from a table in either vector or full matrix form.

The underlying principle is that Stata will accept references to matrix elements within many commands, most notably `generate` and `replace`. We have seen examples of how this can be exploited for entry of small datasets, recoding of categorical variables, and quantile-based or similar binning of counted or measured variables. In the last case, the device grants easy exploration of the consequences of different binning conventions and the instability of bin allocation.

## 6 Acknowledgments

William Gould and Vince Wiggins helped underline how general the idea of look-up tables is.

## 7 References

Cox, N. J. 1999. dm69: Further new matrix commands. *Stata Technical Bulletin* 50: 5–9. Reprinted in *Stata Technical Bulletin Reprints*, vol. 9, pp. 28–34. College Station, TX: Stata Press.

———. 2000. dm79: Yet more new matrix commands. *Stata Technical Bulletin* 56: 4–8. Reprinted in *Stata Technical Bulletin Reprints*, vol. 10, pp. 17–23. College Station, TX: Stata Press.

———. 2002. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2: 202–222.

———. 2003a. Speaking Stata: Problems with lists. *Stata Journal* 3: 185–202.

———. 2003b. Speaking Stata: Problems with tables, Part I. *Stata Journal* 3: 309–324.

Weesie, J. 1997. dm49: Some new matrix commands. *Stata Technical Bulletin* 39: 17–20. Reprinted in *Stata Technical Bulletin Reprints*, vol. 7, pp. 43–48. College Station, TX: Stata Press.

### About the author

Nicholas Cox is a statistically minded geographer at Durham University. He contributes talks, postings, FAQs, and programs to the Stata user community. He has also coauthored 15 commands in official Stata. He wrote several inserts in the *Stata Technical Bulletin* and is an editor of the *Stata Journal*.