The Stata Journal

The *Stata Journal*, electronic version (ISSN 1536-8734) is a publication of Stata Press. Stata and Mata are registered trademarks of StataCorp LP.

The Stata Journal (2007)
7, Number 1, pp. 106–116

Mata Matters: Subscripting

William Gould
StataCorp
College Station, TX
wgould@stata.com

Abstract. Mata is Stata's matrix language. In the Mata Matters column, we show how Mata can be used interactively to solve problems and as a programming language to add new features to Stata. Subscripting is the subject of this column. Stata has three subscripting modes, and two of them are about more than accessing an element of a vector or matrix. The advanced forms of subscripting can, by themselves, be the solution to some problems.

Keywords: pr0028, Mata, subscripts, list subscripts, range subscripts, sampling with replacement, permutation matrices and vectors

1 Introduction

In many Mata programs, dealing with matrices and vectors as a whole is enough, and one never needs to reach inside them to access the individual elements. For instance,

```
. sysuse auto
(1978 Automobile Data)
. tomata
. mata
                                       mata (type end to exit)
: mata describe
      # bytes   type                    name and extent
------------------------------------------------------------------------------
            8   real colvector          displacement[74]
            8   real colvector          foreign[74]
            8   real colvector          gear_ratio[74]
            8   real colvector          headroom[74]
            8   real colvector          length[74]
            8   string colvector        make[74]
            8   real colvector          mpg[74]
            8   real colvector          price[74]
            8   real colvector          rep78[74]
            8   real colvector          trunk[74]
            8   real colvector          turn[74]
            8   real colvector          weight[74]
------------------------------------------------------------------------------
: one = J(74, 1, 1)
: X = (weight, foreign, one)
: y = mpg
: b = invsym(X'X)*X'y
```



```
: b
                  1
    +---------------+
  1 |  -.0065878864 |
  2 |  -1.650029106 |
  3 |   41.67970233 |
    +---------------+
```

In the above, I use `tomata` to create Mata vectors from each variable in `auto.dta`. We discussed that approach in a previous "Mata Matters" column (Gould 2006). `tomata` is available from the SSC archives; type `ssc install tomata`. After that, you can type `help tomata` to learn more about it.

Sometimes, however, you must access elements individually. Most people know that you can type `mpg[5]` to access the fifth value of vector `mpg` or type `X[3,4]` to access the $(3, 4)$ element of `X`. But did you know that you could type `mpg[(1\2\3\4\5)]` or `mpg[(1,2,3,4,5)]` to obtain a vector of the first five elements of `mpg`? Or `mpg[1::5]` or `mpg[1..5]`? Or `mpg[|1\5|]`? Do you know the difference? Do you know when to use each?

Did you know that you could type `X[(1\2\3),(1,2,3)]` to obtain the top 3×3 submatrix of `X`? Or `X[1::3, 1..3]`? Or `X[|1,1 \ 3,3|]`?

Did you know that you can use subscripting to implement sampling with replacement? In one line?

Did you know that you can use subscripting as an alternative to permutation matrices?

Subscripting is the subject of this column, and not just its simple forms.

2 Simple subscripting

There is not much to say about simple subscripting such as `mpg[5]` or `X[3,4]`. How it works is pretty obvious. We do, however, need to discuss Mata's use of the comma because it plays a role in more advanced use and because I will confuse you if I try to explain subscripts and commas all at once.

The comma has two or three meanings in Mata, depending on how you count. First, the comma is the row-join operator. Typing `1,2,3` is like typing `1+2+3` except that the operator is different. The comma operator joins rows. The plus operator adds.

Both are binary operators, meaning that they work on two and only two values. When you type `1+2+3`, that is interpreted to mean `(1+2)+3`: `1+2` is 3, and then `3+3` is 6. It is the same with the comma operator. When you type `1,2,3`, that is interpreted to mean `(1,2),3`: `1,2` is $(1, 2)$, and `(1,2),3` is $(1, 2, 3)$.

You can code `x+2` to add 2 to `x`, assuming that `x` and 2 are conformable. Similarly, you can code `x,2` to join 2 to `x`, assuming that `x` and 2 are conformable. To add `x` and `y`, you code `x+y`, and `x` and `y` might be scalars, vectors, or matrices. To join `x` and `y`, you code `x,y`, and `x` and `y` might be scalars, vectors, or matrices. Regardless of operator, the only requirement is that `x` and `y` be conformable. The conformability rules are different for comma and plus, but so are the conformability rules different for `x+y` and `x*y`. For the comma, the conformability rule is merely that `x` and `y` have the same number of rows.

Parentheses are not required with the comma operator. Most of us type `(1,2,3)`, but we do that only because we are used to seeing the parentheses in mathematical texts—`1,2,3` means the same thing. Typing `(1,2,3)` instead of `1,2,3` is like typing `(1+2+3)` instead of `1+2+3`.

Comma differs from plus in one important way. In Mata, plus always means addition, regardless of context. Comma, however, sometimes means the operator comma and other times means the separator comma. If you code `ttail(15,1.8)`, you are not saying to join 15 and 1.8 to form (15, 1.8) and then pass that single vector to function `ttail()`. You are saying to pass to `ttail()` two arguments, 15 and 1.8. Comma works as a separator here.

Mata usually interprets comma to mean the operator. Whenever you are typing function arguments, however, Mata switches its interpretation of comma to separator. So once you have typed "`ttail(`", commas after that separate one argument from another until you close the parentheses. That is, commas separate unless you open another parenthesis, and then Mata goes back to its usual comma-is-operator rule. Say that we wanted the norm of `1,2,3`. Were we to code `norm(1,2,3)`, we would get an error because `norm()` requires one or two arguments, and we just supplied three, namely, 1, 2, and 3. If we code `norm((1,2,3))`, however, we obtain the desired result, 3.7416. The extra parentheses were required because we needed Mata to interpret comma as the join operator, not as the argument separator. Were we to include yet another pair of parentheses—were we to code `norm(((1,2,3)))`—that would also yield 3.7416. Mata does not follow an alternating rule. Mata's rule is that comma means join except in argument lists, and inside such lists comma means separator unless it is inside one or more parentheses.

There is one more part to the rule: comma is also interpreted as a separator inside subscripting brackets `[` and `]`. When you type `X[1,3]`, the comma separates the two values, treating them like arguments of a function. Everything just said about functions, commas, and parentheses also applies to subscripts, commas, and parentheses.

`[]` allows one or two arguments. One is enough for a vector; two are required for a matrix. If `y` is a column vector, you can refer to `x[3]` or `x[3,1]`; it makes no difference. If `z` is a row vector, you can refer to `z[3]` or `z[1,3]`; it makes no difference. You can even apply this rule to scalars. If `s` is a scalar, you can refer to `s`, `s[1]`, or `s[1,1]`.

If `X` is a matrix, however, you must specify two arguments, such as `X[3,2]`. Referring to `X[3]` by itself is not allowed. You can obtain the entire third row or the entire third

column of `X` by specifying the other subscript as missing value (`.`) as in `X[3,.]` or `X[.,3]`. You can even omit the `.` (but not the comma) and type `X[3,]` or `X[,3]`. That makes `[]` look pretty special, but it really is not. When you type `[` followed by `,` or type `,` followed by `]`, the Mata compiler fills in the `.` for you. What Mata in fact compiles has the `.` in it.

Speaking of operators: `[]` is itself an operator. What distinguishes an operator from mere notation is that operators can be applied to expressions. In Mata, `invsym(X)[1,2]` returns the (1,2) element of the matrix returned by `invsym(X)`. `(invsym(X'X)*X'y)[1]` returns the first element of the vector calculated by `invsym(X'X)*X'y`.

3 List subscripting

`[]` allows vector arguments as well as scalar ones.

Let `x` be a row vector containing $(7, 1, 6, 5)$. Then `x[1]` is 7. `x[(2,4,3,1)]` is $(1, 5, 6, 7)$. `x[(2,4,3)]` is $(1, 5, 6)$. `x[(2,4,4)]` is $(1, 5, 5)$. `x[(2,4,4,3)]` is $(1, 5, 5, 6)$.

Let `z` be a column vector containing $(7, 1, 6, 5)'$. `z[(2\4\3\1)]` is $(1, 5, 6, 7)'$. `z[(2\4\3)]` is $(1, 5, 6)'$. `z[(2\4\4)]` is $(1, 5, 5)'$. `z[(2\4\4\3)]` is $(1, 5, 5, 6)'$.

Whether the subscripts are row or column vectors does not matter; the shape of the result is determined by what is being subscripted, not by the subscripts. `x[(2,4,3,1)]` is $(1, 5, 6, 7)$; so is `x[(2\4\3\1)]`. `z[(2\4\3\1)]` is $(1, 5, 6, 7)'$; so is `z[(2,4,3,1)]`. Code will be more readable, however, if you subscript row vectors with rows and column vectors with columns.

Matrices may also be subscripted with vectors. Let `X` be `(7,3,4 \ 5,9,1)`. Then `X[(1\2),(1,2)]` is `(7,3 \ 5,9)`. As in the other cases, whether the subscripting vectors are rows or columns does not matter, so we could just as well code `X[(1,2),(1,2)]` or `X[(1\2),(1\2)]` or even `X[(1,2),(1\2)]`. As with vectors, elements may be repeated with matrices. `X[(1\1),(2,3)]` is `(3,4 \ 3,4)`. List subscripts can be used to create matrices of lesser or greater dimension than the original. `X[(1\1\2\2), (1,2,3,3)]` is `(7,3,4,4 \ 7,3,4,4 \ 5,9,1,1 \ 5,9,1,1)`.

3.1 List subscripting for sampling with replacement

So what can you do with list subscripts? One use is sampling with replacement. Let `X` be a data matrix, by which I mean that its rows are observations and its columns, variables. Let's say that we want to form a new data matrix `Z` containing observations drawn randomly from `X` with replacement.

The solution is to manufacture an observation vector `o`, each element of which contains an observation number from 1 to `rows(X)`, each element drawn from the rectangular distribution. Given `o`, we can then use subscripting to select those rows of `X`. That some observation numbers (row numbers) may be repeated in `o` will mean just that some rows will be repeated in the final result. If `X` has 10 rows, the code to produce `o` is

```
o = ceil(10*uniform(10,1))
```

and code for a general number of rows is

```
o = ceil(rows(X)*uniform(rows(X),1))
```

Let us take a moment to focus on what is going on here. `uniform()` produces random numbers between 0 and 1. Multiplying by `rows(X)` changes that range so that numbers are between 0 and `rows(X)`. We want integers from 1 to `rows(X)`, so that requires rounding up by using `ceil()`.

Now that we have `o`, the statement to select the data is simply

```
Z = X[o,.]
```

We specify the second subscript as `.` (missing value) because we want all the columns.

We could even put the whole thing together into one statement,

```
Z = X[ceil(rows(X)*uniform(rows(X),1)), .]
```

If we wished, we could omit the period:

```
Z = X[ceil(rows(X)*uniform(rows(X),1)),]
```

In the above, we are drawing a sample of size N from data of size N. If we wanted to draw a sample of size M—and whether $M < N$ or $M > N$ would not matter—the line would become

```
Z = X[ceil(rows(X)*uniform(M,1)),]
```

This is a case of something difficult to program in Stata being trivial in Mata. The following example is well worth understanding.

```
: X
       1    2    3
  +-------------------+
1 |   4    7    9   |
2 |   2   12    3   |
3 |   8    8    7   |
4 |   3    4    1   |
5 |   1    7    9   |
  +-------------------+

: uniformseed(39483)
: o = ceil(5*uniform(5,1))
: o
       1
  +-------+
1 |   2   |
2 |   1   |
3 |   5   |
4 |   5   |
5 |   2   |
  +-------+

: Z = X[o,]
```

```
: Z
       1    2    3
  +-------------------+
1 |   2   12    3     |
2 |   4    7    9     |
3 |   1    7    9     |
4 |   1    7    9     |
5 |   2   12    3     |
  +-------------------+
```

Below I use these ideas to perform a bootstrap of the regression of `mpg` on `weight` and `foreign`, using the automobile data:

```
. sysuse auto, clear
(1978 Automobile Data)
. mata:
------------------------------------------------- mata (type end to exit) -----
: st_view(datay=., ., "mpg")
: st_view(dataX=., ., tokens("weight foreign"))
: n = rows(datay)
: dataX = dataX, J(n, 1, 1)
:
: N = 10000                                     // number of replications
: uniformseed(47686)
: b = J(N, 3, .)
: for (i=1; i<=N; i++) {
>         o = ceil(n*uniform(n,1))
>         y = datay[o,]
>         X = dataX[o,]
>         b[i,] = (invsym(X'X)*X'y)'
> }
: variance(b)
[symmetric]
                  1              2              3
  +-------------------------------------------------+
1 |   3.10765e-07                                   |
2 |    .000172885     1.29395868                    |
3 |  -.0010054771    -.645488234    3.355736044     |
  +-------------------------------------------------+
```

These results are similar to those that would be produced in Stata by typing `estat vce` after `bootstrap, reps(10000): regress mpg weight foreign`.

Beyond sampling with replacement, the example above (1) uses subscripts to index view matrices (i.e., subscripting views is no different from subscripting ordinary matrices) and (2) uses subscripts on the left of the equal sign (which puts results into `b` one row at a time). Coding

```
: b = J(N, 3, .)
: for (...) {
:         ...
:         b[i,] = ...
: }
```

executes much faster than

```
: for (...) {
:         ...
:         b = b \ ...
: }
```

The inferior concatenate solution requires that Mata continually reallocate more memory for `b` and then copy the previous contents into the new, larger matrix. The `b[i,.] = ...` solution simply fills in a row of a preallocated matrix.

3.2 List subscripting with permutation vectors

List subscripts are often used with permutation vectors. A permutation vector is much like the observation vector `o` in the section above except that, rather than each element being just an integer from 1 to n, it is a permutation of the integers from 1 to n, so no rows (or columns) are duplicated or omitted. Permutation vectors arise in data processing, where they are associated with sorting, and in matrix calculation, where they are associated with pivoting.

Consider the vector `y = (7\1\6\5)`. The permutation vector that would put `y` in ascending order is `p = (2\4\3\1)` because `y[p]` is `(1\5\6\7)`. If we simply needed to put `y` into ascending order, easiest would be to code

```
: newy = sort(y, 1)
```

Let us assume, however, that going along with `y`, we have a matrix `X`, and its rows are in the same order as `y`'s. Imagine that we need not only to put `y` into ascending order but also to put `X` into the corresponding order. In the documentation on [M-5] **sort()**, there is another function, `order()`, that claims to be the equivalent of `sort()`. But rather than returning the sorted result, `order()` returns a permutation vector. The solution to our problem is

```
: p = order(y, 1)
: newy = y[p]
: newX = X[p,]
```

In fact, we might even code

```
: p = order(y, 1)
: y = y[p]
: X = X[p,]
```

because given `p`, putting `y` and `X` back in their original order, should that be required, is easy:

```
: y[p] = y
: X[p,] = X
```

That is, list subscripts can be used on the left of the assignment as well as on the right, and putting the subscript on the left inverts the reordering. To understand why,

let's consider the statement `newy=y[p]` and convince ourselves that the statement `y[p] = newy` would undo the mapping. Consider the first element of `p`, and imagine that its value is k. Then `newy = y[p]` assigns the kth value of `y` to the first element of `newy`. Now consider the corresponding actions of `y[p] = newy`. For the first element of `p`, it says to replace the kth element of `y` with the first element of `newy`. That exactly undoes the first operation performed by `newy=y[p]`. The same logic applies to each remaining element of `p`, assuming that the elements are unique, and it is uniqueness of the elements that distinguishes a permutation vector from an observation vector.

Proof aside, anytime you are thinking about using `sort()`, think about using `order()` instead. Having access to the permutation vector will often simplify the coding. By the way, the entire code for `sort(`*x*`, `*idx*`)` reads `return(`*x*`[order(`*x*`,`*idx*`),.])`.

Permutation vectors also arise in various mathematical matrix functions, although they are usually presented as permutation matrices in such cases. Permutation matrices are orthogonal matrices that reorder the rows or columns of another matrix via multiplication. If P were a permutation matrix, PX would be a row permutation of X and XP would be a column permutation. If X were $3 \times k$, the permutation matrix that would place the rows in the order 2, 3, and 1 is

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

This matrix is equivalent to the permutation vector `p = (2\3\1)`, so rather than coding P`*`A, one can code A`[p,]`. You should do this because A`[p,]` is faster and uses less memory than P`*`A. If we were coding with the column permutation A`*`P, we would code A`[,p]`.

Permutation matrices interest us because reordering the rows and/or columns of a matrix in some lengthy calculation can often improve numerical accuracy. One reorders early on, holding on to P (or `p`); solves the reordered problem; and then uses P (or `p`) to reorder the solution. For instance, LU decomposition is a popular ingredient in many matrix calculations. LU decomposition involves finding a lower triangular matrix L and an upper triangular matrix U such that $A = LU$. LU decomposition is seldom used in that form outside textbooks. Mata's LU decomposition routine `lud()` does not return L and U such that $A = LU$; it returns P, L, and U, such that $A = PLU$, where P is a permutation matrix. This amounts to finding the LU decomposition of $P^{-1}A$ or, as it is commonly written, $P'A$, because $P^{-1} = P'$ for permutation matrices. In any case, `lud()` solves a permuted problem and tells you the permutation it used. Do not immediately undo the reordering; the idea is to hold on to the reordering and then apply it at the end to transform results. For instance, if you were using LU decomposition to write a matrix inverter, you would proceed like this:

$$A^{-1} = (PLU)^{-1} = U^{-1}L^{-1}P^{-1} = U^{-1}L^{-1}P'$$

That is an easy enough calculation. You calculate $U^{-1}L^{-1}$ and then postmultiply by P'. But Mata did not return P; it returned `p`, the permutation vector equivalent. Postmultiplying by P' amounts to unpermuting the columns, so if you have `R` equal to $U^{-1}L^{-1}$, you code

```
Ainv = J(rows(R), cols(R), .)           // allocate Ainv
Ainv[,p] = R                            // and then fill in it
```

One cannot simply code `Ainv[,p] = R` if `Ainv` does not already exist because, to subscript on the left, the matrix must already exist. What the matrix contains does not matter since the statement `Ainv[,p] = R` will replace every element of it; `Ainv` need only initially be of the proper dimension. In the code above, I filled `Ainv` in with missing values. In practice, most people code

```
Ainv = R                                // allocate Ainv
Ainv[,p] = R                            // and then fill in it
```

because that takes less typing.

I am not nearly as facile with translating permutation matrix expressions into permutation vector code as I seem. I usually refer to the matrix expression/vector code equivalencies laid out in a table in [M-1] **permutation**. When I write numerical code using something like LU decomposition, I just hold on to the permutation vector `p` that Mata hands me early on. I think in permutation matrices and I often need to do derivations to determine where and how the permutation matrix fits in at the last step. When I get to that last step, I look at the table in [M-1] **permutation** and translate the mathematics to the permutation vector equivalent.

If you look up a Stata function and find it available in two forms, with and without permutation vectors, use the form with permutation vectors unless you are doing a classroom exercise.

4 Range subscripts

In addition to standard subscripts `A[i,j]` and list subscripts `A[i,j]`—which are really the same thing except that `i` and `j` are allowed to be vectors—Mata has range subscripts `A[|i,j|]`. That is, range subscripts are specified in `[|` and `|]` brackets rather than `[` and `]`. Range subscripts cannot do anything that list subscripts cannot do, so many programmers ignore them. For certain problems, however, range subscripts execute faster than list subscripts.

Say that you have 9,000 × 1,000 matrix `A` and you need to obtain the submatrix containing its first 8,999 rows and 999 columns. One way you could code that is

```
B = A[1::8999, 1..999]
```

Better, however, would be to code

```
B = A[|1,1 \ 8999,999|]
```

Range subscripts are for obtaining (and filling in) submatrices and subvectors quickly.

Were you to code `B = A[1::8999, 1..999]`, you would put Mata to a lot of work. Let's go through it:

1. Mata must produce `1::8999`. `::` is not notation; it is an operator, and `1::8999` produces the 8,999-element vector `(1\2\...\8999)`.
2. Similarly, Mata must produce `1..999`, a 999-element vector `(1,2,...,999)`.
3. Finally, one element at a time, Mata must copy indexed-by-vector elements from `A` into `B`. Mata performs $999 \times 8{,}999 = 8{,}990{,}001$ separate 8-byte moves.

When you code `B = A[|1,1 \ 8999,999|]`, on the other hand, Mata proceeds differently. Mata knows that you want to copy a submatrix of `A` to `B` and goes about it efficiently. Mata stores matrices rowwise, so it performs 8,999 separate $999 \times 8 = 7{,}992$ moves, one for each row. If you had coded `B = A[|1,1 \ 5000,1000|]`, Mata would have performed one 40,000,000-byte move. Mata always works out the most efficient way to achieve your desire.

Range subscripts are efficient.

Never code `x[5::1000]` or `x[5..1000]`. Code `x[|5\1000|]`.

Never code `X[3, 5..100]`. Code `X[|3,5 \ 3,100|]`.

Never code `X[5::100, 3]`. Code `X[|5,3 \ 100,3|]`.

Never code `X[5::100, 3..20]`. Code `X[|5,3 \ 100,20|]`.

The commas and backslash inside range subscripts are not notation; they are the row-join and column-join operators. Inside `[|` and `|]` you may specify a scalar, a 1×2 or a 2×1 vector, or a 2×2 matrix. Thus, you may code

```
B = A[|1,1 \ 8999,999|]
```

or you may code

```
r = (1,1 \ 8999,999)
B = A[|r|]
```

Here are the rules and guidelines:

1. `x[2]` and `x[|2|]` mean the same thing. Which you use does not matter.
2. `X[1,2]` and `X[|1,2|]` mean the same thing. `X[1,2]` is faster unless, in a tight loop, you are going to refer to `X[1,2]` often, in which case setting `idx=(1,2)` and

then referring to `X[|idx|]` is faster. (If you are going to refer to `X[1,2]` often and only on the right-hand side of expressions, coding `s=X[1,2]` and then referring to `s` is even faster.)

3. To obtain elements from a vector, whether a row or column, you specify a 2×1 argument containing the range. `x[|4\500|]` specifies elements 4–500. What is typed inside the brackets is a standard, make-a-2×1 expression. Thus, you can code things like `x[|k+1\length(x)-1|]`.

4. To obtain a submatrix, specify a 2×2 argument. The first row specifies the index of the top-left element. The second row specifies the index of the bottom-right element. `X[|`a`,`b`\`c`,`d`|]` specifies the submatrix (a, b) through (c, d).

5. Range subscripts may be used on the left of the assignment to fill in pieces of matrices. In particular, you may code `x[|`a`\`c`|] = z` and `X[|`a`,`b`\`c`,`d`|] = Z`. a, b, c, and d may of course be expressions.

5 Conclusion

Mata has two kinds of subscripts, called list and range. You will not use range subscripts often, but using them when extracting or filling in subvectors or submatrices is important because they are so much faster.

List subscripts include the simple `x[i]` and `X[i,j]` to obtain scalar elements. List subscripts also allow `i` and `j` to be vectors, and then vectors and matrices are returned. Good style is that `i` be a column vector and `j`, a row vector, but Mata does not require this.

Comma has two meanings in Mata. Usually comma is interpreted as the row-join operator, but inside function argument lists, and inside list subscripts (but not range subscripts), comma is interpreted as a separator. Within parentheses inside function argument lists and inside list subscripts, the usual row-join interpretation is restored.

The comma inside range subscripts is the row-join operator, whereas the comma inside list subscripts is a separator.

6 References

Gould, W. 2006. Mata Matters: Interactive use. *Stata Journal* 6: 387–396.

About the author

William Gould is President of StataCorp, head of development, and principal architect of Mata.