matrix define — Matrix definition, operators, and functions

Description

`matrix define` performs matrix computations. The word `define` may be omitted.

`matrix input` provides a method for inputting matrices. The word `input` may be omitted (see the discussion that follows).

For an introduction and overview of matrices in Stata, see **[U] 14 Matrix expressions**.

See [M-2] **exp** for matrix expressions in Mata.

Menu

matrix define

Data > Matrices, ado language > Define matrix from expression

matrix input

Data > Matrices, ado language > Input matrix by hand

Syntax

Perform matrix computations

`matrix` [`define`] *matname* = *matrix_expression*

Input matrices

`matrix` [`input`] *matname* = (# [,# ...] [\ # [, # ...] [\ [...]]])

Remarks and examples

Remarks are presented under the following headings:

Introduction

`matrix define` calculates matrix results from other matrices. For instance,

```
. matrix define D = A + B + C
```

creates `D` containing the sum of `A`, `B`, and `C`. The word `define` may be omitted,

```
. matrix D = A + B + C
```

and the command may be further abbreviated:

```
. mat D=A+B+C
```

The same matrix may appear on both the left and the right of the equal sign in all contexts, and Stata will not become confused. Complicated matrix expressions are allowed.

With `matrix input`, you define the matrix elements rowwise; commas are used to separate elements within a row, and backslashes are used to separate the rows. Spacing does not matter.

```
. matrix input A = (1,2\3,4)
```

The above would also work if you omitted the `input` subcommand.

```
. matrix A = (1,2\3,4)
```

There is a subtle difference: the first method uses the `matrix input` command, and the second uses the matrix expression parser. Omitting `input` allows expressions in the command. For instance,

```
. matrix X = (1+1, 2*3/4 \ 5/2, 3)
```

is understood but

```
. matrix input X = (1+1, 2*3/4 \ 5/2, 3)
```

would produce an error.

`matrix input`, however, has two advantages. First, it allows input of large matrices. (The expression parser is limited because it must "compile" the expressions and, if the result is too long, will produce an error.) Second, `matrix input` allows you to omit the commas.

Inputting matrices by hand

Before turning to operations on matrices, let's examine how matrices are created. Typically, at least in programming situations, you obtain matrices by accessing one of Stata's internal matrices (`e(b)` and `e(V)`; see [P] **matrix get**) or by accumulating it from the data (see [P] **matrix accum**). Nevertheless, the easiest way to create a matrix is to enter it using `matrix input`—this may not be the normal way to create matrices, but it is useful for performing small, experimental calculations.

▷ Example 1

To create the matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

type

```
. matrix A = (1,2 \ 3,4)
```

The spacing does not matter. To define the matrix

$$\mathbf{B} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & . & 6 \end{pmatrix}$$

type

```
. matrix B = (1,2,3 \ 4,.,6)
```

To define the matrix

$$\mathbf{C} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

type

```
. matrix C = (1,2 \ 3,4 \ 5,6)
```

If you need more than one line, and you are working interactively, just keep typing; Stata will wrap the line around the screen. If you are working in a do- or ado-file, see **[U] 16.1.3 Long lines in do-files**.

To create vectors, you enter the elements, separating them by commas or backslashes. To create the row vector

$$\mathbf{D} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$$

type

```
. matrix D = (1,2,3)
```

To create the column vector

$$\mathbf{E} = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

type

```
. matrix E = (1\2\3)
```

To create the 1×1 matrix $\mathbf{F} = (2)$, type

```
. matrix F = (2)
```

In these examples, we have omitted the `input` subcommand. They would work either way.

◁

Matrix operators

In what follows, uppercase letters **A**, **B**, ... stand for matrix names. The matrix operators are

`+`, meaning addition. `matrix` **C=A+B**, **A**: $r \times c$ and **B**: $r \times c$, creates **C**: $r \times c$ containing the elementwise addition $\mathbf{A} + \mathbf{B}$. An error is issued if the matrices are not conformable. Row and column names are obtained from **B**.

`-`, meaning subtraction or negation. `matrix` **C=A−B**, **A**: $r \times c$ and **B**: $r \times c$, creates **C** containing the elementwise subtraction $\mathbf{A} - \mathbf{B}$. An error is issued if the matrices are not conformable. `matrix` **C=−A** creates **C** containing the elementwise negation of **A**. Row and column names are obtained from **B**.

`*`, meaning multiplication. `matrix` **C=A*B**, **A**: $a \times b$ and **B**: $b \times c$, returns **C**: $a \times c$ containing the matrix product **AB**; an error is issued if **A** and **B** are not conformable. The row names of **C** are obtained from the row names of **A**, and the column names of **C** from the column names of **B**.

matrix **C=A***s or matrix **C=**s***A**, **A**: $a \times b$ and s a Stata scalar (see [P] **scalar**) or a literal number, returns **C**: $a \times b$ containing the elements of **A** each multiplied by s. The row and column names of **C** are obtained from **A**. For example, `matrix VC=MYMAT*2.5` multiplies each element of `MYMAT` by 2.5 and stores the result in `VC`.

`/`, meaning matrix division by scalar. matrix **C=A/**s, **A**: $a \times b$ and s a Stata scalar (see [P] **scalar**) or a literal number, returns **C**: $a \times b$ containing the elements of **A** each divided by s. The row and column names of **C** are obtained from **A**.

`#`, meaning the Kronecker product. matrix **C=A#B**, **A**: $a \times b$ and **B**: $c \times d$, returns **C**: $ac \times bd$ containing the Kronecker product **A** ⊗ **B**, all elementwise products of **A** and **B**. The upper-left submatrix of **C** is the product $A_{1,1}$**B**; the submatrix to the right is $A_{1,2}$**B**; and so on. Row and column names are obtained by using the subnames of **A** as resulting equation names and the subnames of **B** for the subnames of **C** in each submatrix.

Nothing, meaning copy. matrix **B=A** copies **A** into **B**. The row and column names of **B** are obtained from **A**. The `matrix rename` command (see [P] **matrix utility**) will rename instead of copy a matrix.

`'`, meaning transpose. matrix **B=A'**, **A**: $r \times c$, creates **B**: $c \times r$ containing the transpose of **A**. The row names of **B** are obtained from the column names of **A** and the column names of **B** from the row names of **A**.

`,`, meaning join columns by row. matrix **C=A,B**, **A**: $a \times b$ and **B**: $a \times c$, returns **C**: $a \times (b+c)$ containing **A** in columns 1 through b and **B** in columns $b + 1$ through $b + c$ (the columns of **B** are appended to the columns of **A**). An error is issued if the matrices are not conformable. The row names of **C** are obtained from **A**. The column names are obtained from **A** and **B**.

`\`, meaning join rows by column. matrix **C=A\B**, **A**: $a \times b$ and **B**: $c \times b$, returns **C**: $(a+c) \times b$ containing **A** in rows 1 through a and **B** in rows $a + 1$ through $a + c$ (the rows of **B** are appended to the rows of **A**). An error is issued if the matrices are not conformable. The column names of **C** are obtained from **A**. The row names are obtained from **A** and **B**.

`matrix define` allows complicated matrix expressions. Parentheses may be used to control the order of evaluation. The default order of precedence for the matrix operators (from highest to lowest) is

Matrix operator precedence

Operator	**Symbol**
parentheses	()
transpose	'
negation	-
Kronecker product	#
division by scalar	/
multiplication	*
subtraction	-
addition	+
column join	,
row join	\

▷ Example 2

The following examples are artificial but informative:

```
. matrix A = (1,2\3,4)
. matrix B = (5,7\9,2)
. matrix C = A+B
. matrix list C
C[2,2]
    c1  c2
r1   6   9
r2  12   6
. matrix B = A-B
. matrix list B
B[2,2]
    c1  c2
r1  -4  -5
r2  -6   2
. matrix X = (1,1\2,5\8,0\4,5)
. matrix C = 3*X*A'*B
. matrix list C
C[4,2]
      c1    c2
r1  -162    -3
r2  -612   -24
r3  -528    24
r4  -744   -18
. matrix D = (X'*X - A'*A)/4
. matrix rownames D = dog cat         // see [P] matrix rownames
. matrix colnames D = bark meow       // see [P] matrix rownames
. matrix list D
symmetric D[2,2]
      bark   meow
dog  18.75
cat   4.25   7.75
. matrix rownames A = aa bb           // see [P] matrix rownames
. matrix colnames A = alpha beta      // see [P] matrix rownames
. matrix list A
A[2,2]
    alpha   beta
aa      1      2
bb      3      4
. matrix D=A#D
. matrix list D
D[4,4]
         alpha:  alpha:   beta:   beta:
          bark    meow    bark    meow
aa:dog   18.75    4.25    37.5     8.5
aa:cat    4.25    7.75     8.5    15.5
bb:dog   56.25   12.75      75      17
bb:cat   12.75   23.25      17      31
. matrix G=A,B\D
. matrix list G

G[6,4]
          alpha    beta      c1      c2
     aa       1       2      -4      -5
     bb       3       4      -6       2
aa:dog    18.75    4.25    37.5     8.5
aa:cat     4.25    7.75     8.5    15.5
bb:dog    56.25   12.75      75      17
bb:cat    12.75   23.25      17      31

. matrix Z = (B - A)'*(B + A'*-B)/4

. matrix list Z

Z[2,2]
          c1     c2
alpha    -81   -1.5
 beta  -44.5    8.5
```

◁

❑ Technical note

Programmers: Watch out for confusion when combining ', meaning to transpose with local macros, where ' is one of the characters that enclose macro names: \`*mname*'. Stata will not become confused, but you might. Compare:

```
. matrix `new1' = `old'
```

and

```
. matrix `new2' = `old''
```

Matrix \``new2`' contains matrix \``old`', transposed. Stata will become confused if you type

```
. matrix `C' = `A'\`B'
```

because the backslash in front of the \``B`' makes the macro processor take the left quote literally. No substitution is ever made for \``B`'. Even worse, the macro processor assumes that the backslash was meant for it and so removes the character! Pretend that \``A`' contained `a`, \``B`' contained `b`, and \``C`' contained `c`. After substitution, the line would read

```
. matrix c = a`B'
```

which is not at all what was intended. To make your meaning clear, put a space after the backslash,

```
. matrix `C' = `A'\ `B'
```

which would then be expanded to read

```
. matrix c = a\ b
```

❑

Matrix functions returning matrices

In addition to matrix operators, Stata has matrix functions, which allow expressions to be passed as arguments. The following matrix functions are provided:

`matrix` **A**=`I(`*dim*`)` defines **A** as the *dim* × *dim* identity matrix, where *dim* is a scalar expression and will be rounded to the nearest integer. For example, `matrix` **A**=`I(3)` defines **A** as the 3 × 3 identity matrix.

`matrix` **A**=`J(`*r*`,`*c*`,`*z*`)` defines **A** as an $r \times c$ matrix containing elements *z*. *r*, *c*, and *z* are scalar expressions with *r* and *c* rounded to the nearest integer. For example, `matrix` **A**=`J(2,3,0)` returns a 2×3 matrix containing 0 for each element.

`matrix` **L**=`cholesky(`*mexp*`)` performs Cholesky decomposition. An error is issued if the matrix expression *mexp* does not evaluate to a square, symmetric matrix. For example, `matrix` **L**=`cholesky(`**A**`)` produces the lower triangular (square root) matrix **L**, such that $\mathbf{LL}' = \mathbf{A}$. The row and column names of **L** are obtained from **A**.

`matrix` **B**=`invsym(`*mexp*`)`, if *mexp* evaluates to a square, symmetric, and positive-definite matrix, returns the inverse. If *mexp* does not evaluate to a positive-definite matrix, rows will be inverted until the diagonal terms are zero or negative; the rows and columns corresponding to these terms will be set to 0, producing a g2-inverse. The row names of **B** are obtained from the column names of *mexp*, and the column names of **B** are obtained from the row names of *mexp*.

`matrix` **B**=`inv(`*mexp*`)`, if *mexp* evaluates to a square but not necessarily symmetric or positive-definite matrix, returns the inverse. A singular matrix will result in an error. The row names of **B** are obtained from the column names of *mexp*, and the column names of **B** are obtained from the row names of *mexp*. `invsym()` should be used in preference to `inv()`, which is less accurate, whenever possible. (Also see [P] **matrix svd** for singular value decomposition.)

`matrix` **B**=`sweep(`*mexp*`,`*n*`)` applies the sweep operator to the *n*th row and column of the square matrix resulting from the matrix expression *mexp*. *n* is a scalar expression and will be rounded to the nearest integer. The names of **B** are obtained from *mexp*, except that the *n*th row and column names are interchanged. For **A**: $n \times n$, $\mathbf{B} =$ `sweep(`**A**`,k)` produces **B**: $n \times n$, defined as

$$
\begin{aligned}
B_{kk} &= \frac{1}{A_{kk}} \\
B_{ik} &= -\frac{A_{ik}}{A_{kk}}, \qquad i \neq k \qquad \text{(}k\text{th column)} \\
B_{kj} &= \frac{A_{ij}}{A_{kk}}, \qquad j \neq k \qquad \text{(}j\text{th row)} \\
B_{ij} &= A_{ij} - \frac{A_{ik}A_{kj}}{A_{kk}}, \qquad i \neq k, j \neq k
\end{aligned}
$$

`matrix` **B**=`corr(`*mexp*`)`, where *mexp* evaluates to a covariance matrix, stores the corresponding correlation matrix in **B**. The row and column names are obtained from *mexp*.

`matrix` **B**=`diag(`*mexp*`)`, where *mexp* evaluates to a row or column vector ($1 \times c$ or $c \times 1$), creates **B**: $c \times c$ with diagonal elements from *mexp* and off-diagonal elements 0. The row and column names are obtained from the column names of *mexp* if *mexp* is a row vector or the row names if *mexp* is a column vector.

`matrix` **B**=`vec(`*mexp*`)`, where *mexp* evaluates to an $r \times c$ matrix, creates **B**: $rc \times 1$ containing the elements of *mexp* starting with the first column and proceeding column by column.

`matrix` **B**=`vecdiag(`*mexp*`)`, where *mexp* evaluates to a square $c \times c$ matrix, creates **B**: $1 \times c$ containing the diagonal elements from *mexp*. `vecdiag()` is the opposite of `diag()`. The row name is set to `r1`. The column names are obtained from the column names of *mexp*.

`matrix` **B**=`vech(`*mexp*`)`, where *mexp* evaluates to an $r \times r$ matrix, creates **B**: $r(r+1)/2 \times 1$ containing the lower triangle elements of *mexp*.

`matrix` **B**=`invvech(`*mexp*`)`, where *mexp* evaluates to an $r(r+1)/2 \times 1$ matrix, creates symmetric **B**: $r \times r$ from the elements of *mexp*.

`matrix` **B**=`vecp(`*mexp*`)`, where *mexp* evaluates to an $r \times r$ matrix, creates **B**: $r(r+1)/2 \times 1$ containing the upper triangle elements of *mexp*.

`matrix` **B**=`invvecp(`*mexp*`)`, where *mexp* evaluates to an $r(r+1)/2 \times 1$ matrix, creates symmetric **B**: $r \times r$ from the elements of *mexp*.

`matrix` **B**=`matuniform(`*r*`,`*c*`)` creates **B**: $r \times c$ containing uniformly distributed pseudorandom numbers on the interval $[\,0, 1\,]$.

`matrix` **B**=`hadamard(`*mexp*`,` *nexp*`)`, where *mexp* and *nexp* evaluate to $r \times c$ matrices, creates a matrix whose (i,j) element is *mexp*$[i,j] \cdot$ *nexp*$[i,j]$. If *mexp* and *nexp* do not evaluate to matrices of the same size, this function reports a conformability error.

`nullmat(`**B**`)` may only be used with the row-join (`,`) and column-join (`\`) operators and informs Stata that **B** might not exist. If **B** does not exist, the row-join or column-join operator simply returns the other matrix-operator argument. An example of the use of `nullmat()` is given in [FN] **Matrix functions**.

`matrix` **B**=`get(`*systemname*`)` returns in **B** a copy of the Stata internal matrix *systemname*; see [P] **matrix get**. You can obtain the coefficient vector and variance–covariance matrix after an estimation command either with `matrix get` or by reference to `e(b)` and `e(V)`.

▷ Example 3

The examples are, once again, artificial but informative.

```
. matrix myid = I(3)

. matrix list myid

symmetric myid[3,3]
    c1  c2  c3
r1   1
r2   0   1
r3   0   0   1

. matrix new = J(2,3,0)

. matrix list new

new[2,3]
    c1  c2  c3
r1   0   0   0
r2   0   0   0

. matrix A = (1,2\2,5)

. matrix Ainv = invsym(A)

. matrix list Ainv

symmetric Ainv[2,2]
    r1  r2
c1   5
c2  -2   1

. matrix L = cholesky(4*I(2) + A'*A)

. matrix list L

L[2,2]
           c1          c2
c1          3           0
c2          4   4.1231056

. matrix B = (1,5,9\2,1,7\3,5,1)
. matrix Binv = inv(B)

. matrix list Binv

Binv[3,3]
            r1          r2          r3
c1  -.27419355   .32258065   .20967742
c2   .15322581  -.20967742   .08870968
c3   .05645161   .08064516  -.07258065

. matrix C = sweep(B,1)

. matrix list C

C[3,3]
     r1   c2   c3
c1    1    5    9
r2   -2   -9  -11
r3   -3  -10  -26

. matrix C = sweep(C,1)

. matrix list C

C[3,3]
    c1  c2  c3
r1   1   5   9
r2   2   1   7
r3   3   5   1

. matrix Cov = (36.6598,-3596.48\-3596.48,604030)

. matrix R = corr(Cov)

. matrix list R

symmetric R[2,2]
           c1          c2
r1          1
r2  -.7642815           1

. matrix d = (1,2,3)

. matrix D = diag(d)

. matrix list D

symmetric D[3,3]
    c1  c2  c3
c1   1
c2   0   2
c3   0   0   3

. matrix e = vec(D)

. matrix list e

e[9,1]
       c1
c1:c1   1
c1:c2   0
c1:c3   0
c2:c1   0
c2:c2   2
c2:c3   0
c3:c1   0
c3:c2   0
c3:c3   3

. matrix f = vecdiag(D)

. matrix list f

f[1,3]
    c1  c2  c3
r1   1   2   3
```

```
. * matrix function arguments can be other matrix functions and expressions
. matrix G = diag(inv(B) * vecdiag(diag(d) + 4*sweep(B+J(3,3,10),2)'*I(3))')

. matrix list G

symmetric G[3,3]
            c1          c2          c3
c1  -3.2170088
c2           0   -7.686217
c3           0           0   2.3548387

. set seed 12345

. matrix U = matuniform(3,4)

. matrix list U

U[3,4]
           c1          c2          c3          c4
r1  .35762972  .40044262  .68938332  .55973557
r2  .57445129  .20769053   .0286627  .68892448
r3  .46934336   .2071526  .00393225  .01302971

. matrix H = hadamard(B,C)

. matrix list H

H[3,3]
    c1  c2  c3
r1   1  25  81
r2   4   1  49
r3   9  25   1
```

◁

Matrix functions returning scalars

In addition to the above functions used with `matrix define`, which can be described as matrix functions returning matrices, there are matrix functions that return mathematical scalars. The list of functions that follow should be viewed as a continuation of **[U] 13.3 Functions**. If the functions listed below are used in a scalar context (for example, used with `display` or `generate`), then **A**, **B**, … below stand for matrix names (possibly as a string literal or string variable name—details later). If the functions below are used in a matrix context (in `matrix define` for instance), then **A**, **B**, … may also stand for matrix expressions.

`rowsof(`**A**`)` and `colsof(`**A**`)` return the number of rows or columns of **A**.

`rownumb(`**A**`,`*string*`)` and `colnumb(`**A**`,`*string*`)` return the row or column number associated with the name specified by *string*. For instance, `rownumb(MYMAT,"price")` returns the row number (say, 3) in `MYMAT` that has the name `price` (subname `price` and equation name blank). `colnumb(MYMAT,"out2:price")` returns the column number associated with the name `out2:price` (subname `price` and equation name `out2`). If row or column name is not found, missing is returned.

`rownumb()` and `colnumb()` can also return the first row or column number associated with an equation name. For example, `colnumb(MYMAT,"out2:")` returns the first column number in `MYMAT` that has equation name `out2`. Missing is returned if the equation name `out2` is not found.

`trace(`**A**`)` returns the sum of the diagonal elements of square matrix **A**. If **A** is not square, missing is returned.

`det(`**A**`)` returns the determinant of square matrix **A**. The determinant is the volume of the $(p-1)$-dimensional manifold described by the matrix in p-dimensional space. If **A** is not square, missing is returned.

`diag0cnt(`**A**`)` returns the number of zeros on the diagonal of the square matrix **A**. If **A** is not square, missing is returned.

`issymmetric(`**A**`)` returns 1 if the matrix is symmetric and 0 otherwise.

`matmissing(`**A**`)` returns 1 if any elements of the matrix are missing and 0 otherwise.

`mreldif(`**A**`,`**B**`)` returns the relative difference of matrix **A** and **B**. If **A** and **B** do not have the same dimensions, missing is returned. The matrix relative difference is defined as

$$\max_{i,j}\left(\frac{|\mathbf{A}[i,j]-\mathbf{B}[i,j]|}{|\mathbf{B}[i,j]|+1}\right)$$

`el(`**A**`,`*i*`,`*j*`)` and **A**`[`*i*`,`*j*`]` return the (i,j) element of **A**. Usually either construct may be used; `el(MYMAT,2,3)` and `MYMAT[2,3]` are equivalent, although `MYMAT[2,3]` is more readable. For the second construct, however, **A** must be a matrix name—it cannot be a string literal or string variable. The first construct allows **A** to be a matrix name, string literal, or string variable. For instance, assume that `mymat` (as opposed to `MYMAT`) is a string variable in the dataset containing matrix names. `mymat[2,3]` refers to the $(2,3)$ element of the matrix named `mymat`, a matrix that probably does not exist, and so produces an error. `el(mymat,2,3)` refers to the data variable `mymat`; the contents of that variable will be taken to obtain the matrix name, and `el()` will then return the $(2,3)$ element of that matrix. If that matrix does not exist, Stata will not issue an error; because you referred to it indirectly, the `el()` function will return missing.

In either construct, *i* and *j* may be any expression (an *exp*) evaluating to a real. `MYMAT[2,3+1]` returns the $(2,4)$ element. In programs that loop, you might refer to `` MYMAT[`i',`j'+1] ``.

In a matrix context (such as `matrix define`), the first argument of `el()` may be a matrix expression. For instance, `matrix` **A** `=` **B**`*el(`**B**`-`**C**`,1,1)` is allowed, but `display el(`**B**`-`**C**`,1,1)` would be an error because `display` is in a scalar context.

The matrix functions returning scalars defined above can be used in any context that allows an expression—what is abbreviated *exp* in the syntax diagrams throughout this manual. For instance, `trace()` returns the (scalar) trace of a matrix. Say that you have a matrix called `MYX`. You could type

```
. generate tr = trace(MYX)
```

although this would be a silly thing to do. It would force Stata to evaluate the trace of the matrix many times, once for each observation in the data, and it would then store that same result over and over again in the new data variable `tr`. But you could do it because, if you examine the syntax diagram for `generate` (see [D] **generate**), `generate` allows an *exp*.

If you just wanted to see the trace of `MYX`, you could type

```
. display trace(MYX)
```

because the syntax diagram for `display` also allows an *exp*; see [P] **display**. You could do either of the following:

```
. local tr = trace(MYX)
. scalar tr = trace(MYX)
```

This is more useful because it will evaluate the trace only once and then store the result. In the first case, the result will be stored in a local macro (see [P] **macro**); in the second, it will be stored in a Stata scalar (see [P] **scalar**).

▷ Example 4

Storing the number as a scalar is better for two reasons: it is more accurate (scalars are stored in double precision), and it is faster (macros are stored as printable characters, and this conversion is a time-consuming operation). Not too much should be made of the accuracy issue; macros are stored with at least 13 digits, but it can sometimes make a difference.

In any case, let's demonstrate that both methods work by using the simple trace function:

```
. matrix A = (1,6\8,4)
. local tr = trace(A)
. display `tr'
5
. scalar sctr = trace(A)
. scalar list sctr
      sctr =          5
```

◁

Subscripting and element-by-element definition

`matrix` **B=A**`[`r_1`,`r_2`]`, for range expressions r_1 and r_2 (defined below), extracts a submatrix from **A** and stores it in **B**. Row and column names of **B** are obtained from the extracted rows and columns of **A**. In what follows, assume that **A** is $a \times b$.

A range expression can be a literal number. For example, `matrix` **B=A**`[1,2]` would return a 1×1 matrix containing $A_{1,2}$.

A range expression can be a number followed by two periods followed by another number, meaning the rows or columns from the first number to the second. For example, `matrix` **B=A**`[2..4,1..5]` would return a 3×5 matrix containing the second through fourth rows and the first through fifth columns of **A**.

A range expression can be a number followed by three periods, meaning all the remaining rows or columns from that number. For example, `matrix` **B=A**`[3,4...]` would return a $1 \times b - 3$ matrix (row vector) containing the fourth through last elements of the third row of **A**.

A range expression can be a quoted string, in which case it refers to the row or column with the specified name. For example, `matrix` **B=A**`["price","mpg"]` returns a 1×1 matrix containing the element whose row name is `price` and column name is `mpg`, which would be the same as `matrix` **B=A**`[2,3]` if the second row were named `price` and the third column `mpg`. `matrix` **B=A**`["price",1...]` would return the $1 \times b$ vector corresponding to the row named `price`. In either case, if there is no matrix row or column with the specified name, an error is issued, and the return code is set to 111. If the row or column names include both an equation name and a subname, the fully qualified name must be specified, as in `matrix` **B=A**`["eq1:price",1...]`.

A range expression can be a quoted string containing only an equation name, in which case it refers to all rows or columns with the specified equation name. For example, `matrix` **B=A**`["eq1:","eq1:"]` would return the submatrix of rows and columns that have equation names `eq1`.

A range expression containing a quoted string referring to an element (not to an entire equation) can be combined with the `..` and `...` syntaxes above: For example, `matrix` **B=A**`["price"...,"price"...]` would define **B** as the submatrix of **A** beginning with the rows and columns corresponding to `price`. `matrix` **B=A**`["price".."mpg","price".."mpg"]` would define **B** as the submatrix of **A** starting at rows and columns corresponding to `price` and continuing through the rows and columns corresponding to `mpg`.

A range expression can be mixed. For example, `matrix` **B=A**`[1.."price",2]` defines **B** as the column vector extracted from the second column of **A** containing the first element through the element corresponding to `price`.

Scalar expressions may be used in place of literal numbers. The resulting number will be rounded to the nearest integer. Subscripting with scalar expressions may be used in any expression context (such as `generate` or `replace`). Subscripting with row and column names may be used only in a matrix expression context. This is really not a constraint; see the `rownumb()` and `colnumb()` functions discussed previously in the section titled *Matrix functions returning scalars*.

`matrix` **A**`[`*r*`,`*c*`]=`*exp* changes the *r*,*c* element of **A** to contain the result of the evaluated scalar expression, as defined in **[U] 13 Functions and expressions**, and as further defined in *Matrix functions returning scalars*. *r* and *c* may be scalar expressions and will be rounded to the nearest integer. The matrix **A** must already exist; the matrix function `J()` can be used to achieve this.

`matrix` **A**`[`*r*`,`*c*`]=`*mexp* places the matrix resulting from the *mexp* matrix expression into the already existing matrix **A**, with the upper-left corner of the *mexp* matrix located at the *r*,*c* element of **A**. If there is not enough room to place the *mexp* matrix at that location, a conformability error will be issued, and the return code will be set to 503. *r* and *c* may be scalar expressions and will be rounded to the nearest integer.

▷ Example 5

Continuing with our artificial but informative examples,

```
. matrix A = (1,2,3,4\5,6,7,8\9,10,11,12\13,14,15,16)

. matrix rownames A = mercury venus earth mars

. matrix colnames A = poor average good exc

. matrix list A

A[4,4]
            poor   average      good       exc
mercury        1         2         3         4
  venus        5         6         7         8
  earth        9        10        11        12
   mars       13        14        15        16

. matrix b = A[1,2..3]

. matrix list b

b[1,2]
         average      good
mercury        2         3

. matrix b = A[2...,1..3]

. matrix list b

b[3,3]
          poor   average      good
venus        5         6         7
earth        9        10        11
 mars       13        14        15

. matrix b = A["venus".."earth","average"...]
. matrix list b

b[2,3]
       average     good      exc
venus        6        7        8
earth       10       11       12

. matrix b = A["mars",2...]

. matrix list b

b[1,3]
      average     good      exc
mars       14       15       16

. matrix b = A[sqrt(9)+1..substr("xmars",2,4),2.8..2*2] /* strange but valid */

. matrix list b

b[1,2]
      good   exc
mars    15    16

. matrix rownames A = eq1:alpha eq1:beta eq2:alpha eq2:beta

. matrix colnames A = eq1:one eq1:two eq2:one eq2:two

. matrix list A

A[4,4]
            eq1:  eq1:  eq2:  eq2:
            one   two   one   two
eq1:alpha     1     2     3     4
 eq1:beta     5     6     7     8
eq2:alpha     9    10    11    12
 eq2:beta    13    14    15    16

. matrix b = A["eq1:","eq2:"]

. matrix list b

b[2,2]
            eq2:  eq2:
            one   two
eq1:alpha     3     4
 eq1:beta     7     8

. matrix A[3,2] = sqrt(9)

. matrix list A

A[4,4]
            eq1:  eq1:  eq2:  eq2:
            one   two   one   two
eq1:alpha     1     2     3     4
 eq1:beta     5     6     7     8
eq2:alpha     9     3    11    12
 eq2:beta    13    14    15    16

. matrix X = (-3,0\-1,-6)

. matrix A[1,3] = X

. matrix list A

A[4,4]
            eq1:  eq1:  eq2:  eq2:
            one   two   one   two
eq1:alpha     1     2    -3     0
 eq1:beta     5     6    -1    -6
eq2:alpha     9     3    11    12
 eq2:beta    13    14    15    16
```

◁

❑ Technical note

`matrix` **A**`[`*i*`,`*j*`]=`*exp* can be used to implement matrix formulas that perhaps Stata does not have built in. Let's pretend that Stata could not multiply matrices. We could still multiply matrices, and after some work, we could do so conveniently. Given two matrices, **A**: $a \times b$ and **B**: $b \times c$, the (i, j) element of **C** = **AB**, **C**: $a \times c$, is defined as

$$C_{ij} = \sum_{k=1}^{b} A_{ik} B_{kj}$$

Here is a Stata program to make that calculation:

```
program matmult                                 // arguments A B C, creates C=A*B
        version 19.5      // (or version 19 if you do not have StataNow)
        args A B C                              // unload arguments into better names
        if colsof(`A')!=rowsof(`B') {           // check conformability
                error 503
        }
        local a = rowsof(`A')                   // obtain dimensioning information
        local b = colsof(`A')                   //    see Matrix functions returning
        local c = colsof(`B')                   //    scalars above
        matrix `C' = J(`a',`c',0)               // create result containing 0s
        forvalues i = 1/`a' {
                forvalues `j' = 1/`c' {
                        forvalues `k' = 1/`b' {
                                matrix `C'[`i',`j'] = `C'[`i',`j'] + /*
                                        */ `A'[`i',`k']*`B'[`k',`j']
                        }
                }
        }
end
```

Now if in some other program, we needed to multiply matrix `XXI` by `Xy` to form result `beta`, we could type `matmult XXI Xy beta` and never use Stata's built-in method for multiplying matrices (`matrix beta=XXI*Xy`). If we typed the program `matmult` into a file named `matmult.ado`, we would not even have to bother to load `matmult` before using it—it would be loaded automatically; see [U] **17 Ado-files**.

❑

Name conflicts in expressions (namespaces)

See [P] **matrix** for a description of namespaces. A matrix might have the same name as a variable in the dataset, and if it does, Stata might appear confused when evaluating an expression (an *exp*). When the names conflict, Stata uses the rule that it always takes the data-variable interpretation. You can override this.

First, when working interactively, you can avoid the problem by simply naming your matrices differently from your variables.

Second, when writing programs, you can avoid name conflicts by obtaining names for matrices from `tempname`; see [P] **macro**.

Third, whether working interactively or writing programs, when using names that might conflict, you can use the `matrix()` pseudofunction to force Stata to take the matrix-name interpretation.

`matrix(`*name*`)` says that *name* is to be interpreted as a matrix name. For instance, consider the statement `local new=trace(xx)`. This might work and it might not. If `xx` is a matrix and there is no variable named `xx` in your dataset, it will work. If there is also a numeric variable named `xx` in your dataset, it will not work. Typing the statement will produce a type-mismatch error—Stata assumes that when you type `xx`, you are referring to the data variable `xx` because there is a data variable `xx`. Typing `local new=trace(matrix(xx))` will then produce the desired result. When writing programs using matrix names not obtained from `tempname`, you are strongly advised to state explicitly that all matrix names are indeed matrix names by using the `matrix()` function.

The only exception to this recommendation has to do with the construct **A**`[`*i*`,`*j*`]`. The two subscripts indicate to Stata that **A** must be a matrix name and not an attempt to subscript a variable, so `matrix()` is not needed. This exception applies only to **A**`[`*i*`,`*j*`]`; it does not apply to `el(`**A**`,`*i*`,`*j*`)`, which would be more safely written as `el(matrix(`**A**`),`*i*`,`*j*`)`.

❑ Technical note

The `matrix()` and `scalar()` pseudofunctions (see [P] **scalar**) are really the same function, but you do not need to understand this fine point to program Stata successfully. Understanding this might, however, lead to producing more readable code. The formal definition is this:

`scalar(`*exp*`)` (and therefore `matrix(`*exp*`)`) evaluates *exp* but restricts Stata to interpreting all names in *exp* as scalar or matrix names. Scalars and matrices share the same namespace.

Therefore, because `scalar()` and `matrix()` are the same function, typing `trace(matrix(xx))` or `trace(scalar(xx))` would do the same thing, even though the second looks wrong. Because `scalar()` and `matrix()` allow an *exp*, you could also type `scalar(trace(xx))` and achieve the same result. `scalar()` evaluates the *exp* inside the parentheses: it merely restricts how names are interpreted, so now `trace(xx)` clearly means the trace of the matrix named `xx`.

How can you make your code more readable? Pretend that you wanted to calculate the trace plus the determinant of matrix `xx` and store it in the Stata scalar named `tpd` (no, there is no reason you would ever want to make such a silly calculation). You are writing a program and want to protect yourself from `xx` also existing in the dataset. One solution would be

```
    scalar tpd = trace(matrix(xx)) + det(matrix(xx))
```

Knowing the full interpretation rule, however, you realize that you can shorten this to

```
    scalar tpd = matrix(trace(xx) + det(xx))
```

and then, to make it more readable, you substitute `scalar()` for `matrix()`:

```
    scalar tpd = scalar(trace(xx) + det(xx))
```

❑

Macro functions

The following macro functions (see [P] **macro**) are also defined:

`rownames` **A** and `colnames` **A** return a list of all the row or column subnames (with time-series operators if applicable) of **A**, separated by single blanks. The equation names, even if present, are not included.

`roweq` **A** and `coleq` **A** return a list of all the row equation names or column equation names of **A**, separated by single blanks, and with each name appearing however many times it appears in the matrix.

`rowfullnames` **A** and `colfullnames` **A** return a list of all the row or column names, including equation names of **A**, separated by single blanks.

▷ Example 6

These functions are provided as macro functions and standard expression functions because Stata's expression evaluator works only with strings of no more than 2,045 characters, something not true of Stata's macro parser. A matrix with many rows or columns can produce an exceedingly long list of names.

In sophisticated programming situations, you sometimes want to process the matrices by row and column names rather than by row and column numbers. Assume that you are programming and have two matrices, `xx` and `yy`. You know that they contain the same column names, but they might be in a different order. You want to reorganize `yy` to be in the same order as `xx`. The following code fragment will create `‘newyy’` (a matrix name obtained from `tempname`) containing `yy` in the same order as `xx`:

```
tempname newyy newcol
local names : colfullnames(xx)
foreach name of local names {
        local j = colnumb(yy,"‘name’")
        if ‘j’>=. {
                display as error "column for ‘name’ not found"
                exit 111
        }
        matrix ‘newcol’ = yy[1...,‘j’]
        matrix ‘newyy’ = nullmat(‘newyy’),‘newcol’
}
```

◁

Reference

Spinelli, D. 2023. Improving flexibility and ease of matrix subsetting: The submatrix command. *Stata Journal* 23: 1045–1056.

Also see

[P] **macro** — Macro definition and manipulation

[P] **matrix** — Introduction to matrix commands

[P] **matrix get** — Access system matrices

[P] **matrix utility** — List, rename, and drop matrices

[P] **scalar** — Scalar variables

[U] **13.3 Functions**

[U] **14 Matrix expressions**

Mata Reference Manual