Title

stata.com

Glossary

Description

Commonly used terms are defined here.

Mata glossary

arguments

The values a function receives are called the function's arguments. For instance, in `lud(`*A*`, `*L*`, `*U*`)`, *A*, *L*, and *U* are the arguments.

array

An array is any indexed object that holds other objects as elements. Vectors are examples of 1-dimensional arrays. Vector $\mathbf{v}$ is an array, and $\mathbf{v}[1]$ is its first element. Matrices are 2-dimensional arrays. Matrix $\mathbf{X}$ is an array, and $\mathbf{X}[1,1]$ is its first element. In theory, one can have 3-dimensional, 4-dimensional, and higher arrays, although Mata does not directly provide them. See [M-2] **Subscripts** for more information on arrays in Mata.

Arrays are usually indexed by sequential integers, but in associative arrays, the indices are strings that have no natural ordering. Associative arrays can be 1-dimensional, 2-dimensional, or higher. If *A* were an associative array, then *A*["first"] might be one of its elements. See [M-5] **asarray()** for associative arrays in Mata.

binary operator

A binary operator is an operator applied to two arguments. In `2-3`, the minus sign is a binary operator, as opposed to the minus sign in `-9`, which is a unary operator.

broad type

Two matrices are said to be of the same broad type if the elements in each are numeric, are string, or are pointers. Mata provides two numeric types, real and complex. The term *broad type* is used to mask the distinction within numeric and is often used when discussing operators or functions. One might say, "The comma operator can be used to join the rows of two matrices of the same broad type," and the implication of that is that one could join a real to a complex. The result would be complex. Also see *type, eltype, and orgtype*.

c-conformability

Matrix, vector, or scalar A is said to be c-conformable with matrix, vector, or scalar B if they have the same number of rows and columns (they are *p-conformable*), or if they have the same number of rows and one is a vector, or if they have the same number of columns and one is a vector, or if one or the other is a scalar. c stands for colon; c-conformable matrices are suitable for being used with Mata's *:op* operators. A and B are c-conformable if and only if

A	B
$r \times c$	$r \times c$
$r \times 1$	$r \times c$
$1 \times c$	$r \times c$
1×1	$r \times c$
$r \times c$	$r \times 1$
$r \times c$	$1 \times c$
$r \times c$	1×1

The idea behind c-conformability is generalized elementwise operation. Consider C=A:$*B$. If A and B have the same number of rows and have the same number of columns, then $||C_{ij}|| = ||A_{ij}*B_{ij}||$. Now say that A is a column vector and B is a matrix. Then $||C_{ij}|| = ||A_i*B_{ij}||$: each element of A is applied to the entire row of B. If A is a row vector, each column of A is applied to the entire column of B. If A is a scalar, A is applied to every element of B. And then all the rules repeat, with the roles of A and B interchanged. See [M-2] **op_colon** for a complete definition.

class programming

See *object-oriented programming*.

colon operators

Colon operators are operators preceded by a colon, and the colon indicates that the operator is to be performed elementwise. A:$*B$ indicates element-by-element multiplication, whereas $A*B$ indicates matrix multiplication. Colons may be placed in front of any operator. Usually one thinks of elementwise as meaning $c_{ij} = a_{ij} <op> b_{ij}$, but in Mata, elementwise is also generalized to include c-conformability. See [M-2] **op_colon**.

column stripes

See *row and column stripes*.

column-major order

Matrices are stored as vectors. Column-major order specifies that the vector form of a matrix is created by stacking the columns. For instance,

```
: A
       1   2
    +-------+
  1 |  1   4|
  2 |  2   5|
  3 |  3   6|
    +-------+
```

is stored as

```
       1   2   3   4   5   6
    +-----------------------+
  1 |  1   2   3   4   5   6|
    +-----------------------+
```

in column-major order. The LAPACK functions use column-major order. Mata uses row-major order. See *row-major order*.

colvector

See *vector, colvector, and rowvector*.

complex

A matrix is said to be complex if its elements are complex numbers. Complex is one of two numeric types in Stata, the other being real. Complex is generally used to describe how a matrix is stored and not the kind of numbers that happen to be in it: complex matrix Z might happen to contain real numbers. Also see *type, eltype, and orgtype*.

condition number

The condition number associated with a numerical problem is a measure of that quantity's amenability to digital computation. A problem with a low condition number is said to be well conditioned, whereas a problem with a high condition number is said to be ill conditioned.

Sometimes reciprocals of condition numbers are reported and yet authors will still refer to them sloppily as condition numbers. Reciprocal condition numbers are often scaled between 0 and 1, with values near `epsilon(1)` indicating problems.

conformability

Conformability refers to row-and-column matching between two or more matrices. For instance, to multiply $A*B$, A must have the same number of columns as B has rows. If that is not true, then the matrices are said to be nonconformable (for multiplication).

Three kinds of conformability are often mentioned in the Mata documentation: *p-conformability*, *c-conformability*, and *r-conformability*.

conjugate

If $z = a + b\mathrm{i}$, the conjugate of z is $\mathrm{conj}(z) = a - b\mathrm{i}$. The conjugate is obtained by reversing the sign of the imaginary part. The conjugate of a real number is the number itself.

conjugate transpose

See *transpose*.

data matrix

A dataset containing n observations on k variables in often stored in an $n \times k$ matrix. An observation refers to a row of that matrix; a variable refers to a column. When the rows are observations and the columns are variables, the matrix is called a data matrix.

declarations

Declarations state the *eltype* and *orgtype* of functions, arguments, and variables. In

```
real matrix myfunc(real vector A, complex scalar B)
{
     real scalar i
     ...
}
```

the `real matrix` is a function declaration, the `real vector` and `complex scalar` are argument declarations, and `real scalar i` is a variable declaration. The `real matrix` states the function returns a real matrix. The `real vector` and `complex scalar` state the kind of arguments `myfunc()` expects and requires. The `real scalar i` helps Mata to produce more efficient compiled code.

Declarations are optional, so the above could just as well have read

```
function myfunc(A, B)
{
     ...
}
```

When you omit the function declaration, you must substitute the word `function`.

When you omit the other declarations, `transmorphic matrix` is assumed, which is fancy jargon for a matrix that can hold anything. The advantages of explicit declarations are that they reduce the chances you make a mistake either in coding or in using the function, and they assist Mata in producing more efficient code. Working interactively, most people omit the declarations.

See [M-2] **Declarations** for more information.

defective matrix

An $n \times n$ matrix is defective if it does not have n linearly independent eigenvectors.

dereference

Dereferencing is an action performed on pointers. Pointers contain memory addresses, such as 0x2a1228. One assumes something of interest is stored at 0x2a1228, say, a real scalar equal to 2. When one accesses that 2 via the pointer by coding $*p$, one is said to be dereferencing the pointer. Unary $*$ is the dereferencing operator.

diagonal matrix

A matrix is diagonal if its off-diagonal elements are zero; A is diagonal if $A[i,j]==0$ for $i!=j$. Usually, diagonal matrices are also *square*. Some definitions require that a diagonal matrix also be a square matrix.

diagonal of a matrix

The diagonal of a matrix is the set of elements $A[i,j]$.

dyadic operator

Synonym for binary operator.

eigenvalues and eigenvectors

A scalar, λ, is said to be an eigenvalue of square matrix $\mathbf{A}$: $n \times n$ if there is a nonzero column vector $\mathbf{x}$: $n \times 1$ (called an eigenvector) such that

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x} \tag{1}$$

Equation (1) can also be written as

$$(\mathbf{A} - \lambda\mathbf{I})\mathbf{x} = 0$$

where $\mathbf{I}$ is the $n \times n$ identity matrix. A nontrivial solution to this system of n linear homogeneous equations exists if and only if

$$\det(\mathbf{A} - \lambda\mathbf{I}) = 0 \tag{2}$$

This nth-degree polynomial in λ is called the characteristic polynomial or characteristic equation of $\mathbf{A}$, and the eigenvalues λ are its roots, also known as the characteristic roots.

The eigenvector defined by (1) is also known as the right eigenvector, because matrix $\mathbf{A}$ is postmultiplied by eigenvector $\mathbf{x}$. See [M-5] **eigensystem( )** and *left eigenvectors*.

eltype

See *type, eltype, and orgtype*.

epsilon(1), etc.

epsilon(1) refers to the unit roundoff error associated with a computer, also informally called machine precision. It is the smallest amount by which a number may differ from 1. For IEEE double-precision variables, epsilon(1) is approximately 2.22045e–16.

`epsilon(`*x*`)` is the smallest amount by which a real number can differ from *x*, or an approximation thereof; see [M-5] **epsilon( )**.

exp

exp is used in syntax diagrams to mean "any valid expression may appear here"; see [M-2] **exp**.

external variable

See *global variable*.

frames

Frames, also known as data frames, are in-memory areas where datasets are analyzed. Stata can hold multiple datasets in memory, and each dataset is held in a memory area called a frame. A variety of commands exist to manage frames and manipulate the data in them. See [D] **frames**.

function

The words *program* and *function* are used interchangeably. The programs that you write in Mata are in fact functions. Functions receive arguments and optionally return results.

Examples of functions that are included with Mata are `sqrt()`, `ttail()`, and `substr()`. Such functions are often referred to as the built-in functions or the library functions. Built-in functions refer to functions implemented in the C code that implements Mata, and library functions refer to functions written in the Mata programming language, but many users use the words interchangeably because how functions are implemented is of little importance. If you have a choice between using a built-in function and a library function, however, the built-in function will usually execute more quickly and the library function will be easier to use. Mostly, however, features are implemented one way or the other and you have no choice.

Also see *underscore functions*.

For a list of the functions that Mata provides, see [M-4] **Intro**.

generalized eigenvalues

A scalar, λ, is said to be a generalized eigenvalue of a pair of $n \times n$ square numeric matrices $\mathbf{A}$, $\mathbf{B}$ if there is a nonzero column vector $\mathbf{x}$: $n \times 1$ (called a generalized eigenvector) such that

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{B}\mathbf{x} \tag{1}$$

Equation (1) can also be written as

$$(\mathbf{A} - \lambda\mathbf{B})\mathbf{x} = 0$$

A nontrivial solution to this system of n linear homogeneous equations exists if and only if

$$\det(\mathbf{A} - \lambda\mathbf{B}) = 0 \tag{2}$$

In practice, the generalized eigenvalue problem for the matrix pair $(\mathbf{A}, \mathbf{B})$ is usually formulated as finding a pair of scalars (w, b) and a nonzero column vector $\mathbf{x}$ such that

$$w\mathbf{A}\mathbf{x} = b\mathbf{B}\mathbf{x}$$

The scalar w/b is a generalized eigenvalue if b is not zero.

Infinity is a generalized eigenvalue if b is zero or numerically close to zero. This situation may arise if $\mathbf{B}$ is singular.

The Mata functions that compute generalized eigenvalues return them in two complex vectors, $\mathbf{w}$ and $\mathbf{b}$ of length n. If $\mathbf{b}[i] = 0$, the ith generalized eigenvalue is infinite, otherwise the ith generalized eigenvalue is $\mathbf{w}[i]/\mathbf{b}[i]$.

global variable

Global variables, also known as external variables and as global external variables, refer to variables that are common across programs and which programs may access without the variable being passed as an argument.

The variables you create interactively are global variables. Even so, programs cannot access those variables without engaging in another step, and global variables can be created without your creating them interactively.

To access (and create if necessary) global external variables, you declare the variable in the body of your program:

```
function myfunction(...)
{
     external real scalar globalvar
     ...
}
```

See *Linking to external globals* in [M-2] **Declarations**.

There are other ways of creating and accessing global variables, but the declaration method is recommended. The alternatives are `crexternal()`, `findexternal()`, and `rmexternal()` documented in [M-5] **findexternal( )** and `valofexternal()` documented in [M-5] **valofexternal( )**.

hashing, hash functions, and hash tables

Hashing refers to a technique for quickly finding information corresponding to an identifier. The identifier might be a name, a Social Security number, fingerprints, or anything else on which the information is said to be indexed. The hash function returns a many-to-one mapping of identifiers onto a dense subrange of the integers. Those integers, called hashes, are then used to index a hash table. The selected element of the hash table specifies a list containing identifiers and information. The list is then searched for the particular identifier desired. The advantage is that rather than searching a single

large list, one need only search one of K smaller lists. For this to be fast, the hash function must be quick to compute and produce roughly equal frequencies of hashes over the range of identifiers likely to be observed.

Hermitian matrix

Matrix A is Hermitian if it is equal to its conjugate transpose; $A = A'$; see *transpose*. This means that each off-diagonal element a_{ij} must equal the conjugate of a_{ji}, and that the diagonal elements must be real. The following matrix is Hermitian:

$$\begin{bmatrix} 2 & 4+5i \\ 4-5i & 6 \end{bmatrix}$$

The definition $A = A'$ is the same as the definition for a symmetric matrix, although usually the word *symmetric* is reserved for real matrices and Hermitian, for complex matrices. In this manual, we use the word *symmetric* for both; see *symmetric matrices*.

Hessenberg decomposition

The Hessenberg decomposition of a matrix, $\mathbf{A}$, can be written as

$$\mathbf{Q}'\mathbf{A}\mathbf{Q} = \mathbf{H}$$

where $\mathbf{H}$ is in upper Hessenberg form and $\mathbf{Q}$ is orthogonal if $\mathbf{A}$ is real or unitary if $\mathbf{A}$ is complex. See [M-5] **hessenbergd( )**.

Hessenberg form

A matrix, $\mathbf{A}$, is in upper Hessenberg form if all entries below the first subdiagonal are zero: $A_{ij} = 0$ for all $i > j + 1$.

A matrix, $\mathbf{A}$, is in lower Hessenberg form if all entries above the first superdiagonal are zero: $A_{ij} = 0$ for all $j > i + 1$.

instance and realization

Instance and realization are synonyms for variable, as in Mata variable. For instance, consider a real scalar variable X. One can equally well say that X is an instance of a real scalar or a realization of a real scalar. Authors represent a variable this way when they wish to emphasize that X is not representative of all real scalars but is just one of many real scalars. Instance is often used with structures and classes when the writer wishes to emphasize the difference between the values contained in the variable and the definition of the structure or the class. It is confusing to say that V is a class C, even though it is commonly said, because the reader might confuse the definition of C with the specific values contained in V. Thus careful authors say that V is an instance of class C.

istmt

An *istmt* is an interactive statement, a statement typed at Mata's colon prompt.

J(r, c, value)

`J()` is the function that returns an $r \times c$ matrix with all elements set to *value*; see [M-5] **J()**. Also, `J()` is often used in the documentation to describe the various types of *void* matrices; see *void matrix*. Thus the documentation might say that such-and-such returns `J(0, 0, .)` under certain conditions. That is another way of saying that such-and-such returns a 0×0 real matrix.

When r or c is 0, there are no elements to be filled in with *value*, but even so, *value* is used to determine the type of the matrix. Thus `J(0, 0, 1i)` refers to a 0×0 complex matrix, `J(0, 0, "")` refers to a 0×0 string matrix, and `J(0, 0, NULL)` refers to a 0×0 *pointer* matrix.

In the documentation, `J()` is used for more than describing 0×0 matrices. Sometimes, the matrices being described are $r \times 0$ or are $0 \times c$. Say that a function `example(`X`)` is supposed to return a column vector; perhaps it returns the last column of X. Now say that X is 0×0. Function `example()` still should return a column vector, and so it returns a 0×1 matrix. This would be documented by noting that `example()` returns `J(0, 1, .)` when X is 0×0.

LAPACK

LAPACK stands for Linear Algebra PACKage and forms the basis for many of Mata's linear algebra capabilities; see [M-1] **LAPACK**.

left eigenvectors

A vector $\mathbf{x}$: $n \times 1$ is said to be a left eigenvector of square matrix $\mathbf{A}$: $n \times n$ if there is a nonzero scalar, λ, such that

$$\mathbf{xA} = \lambda\mathbf{x}$$

lval

lval stands for left-hand-side value and is defined as the property of being able to appear on the left-hand side of an equal-assignment operator. Matrices are *lvals* in Mata, and thus

```
X = ...
```

is valid. Functions are not *lvals*; thus, you cannot code

```
substr(mystr,1,3) = "abc"
```

lvals would be easy to describe except that *pointers* can also be lvals. Few people ever use pointers. See [M-2] **op_assignment** for a complete definition.

machine precision

See *epsilon(1), etc.*

.mata source code file

By convention, we store the Mata source code for function *function*`()` in file *function*`.mata`; see [M-1] **Source**.

matrix

The most general organization of data, containing r rows and c columns. Vectors, column vectors, row vectors, and scalars are special cases of matrices.

.mlib library

The object code of functions can be collected and stored in a library. Most Mata functions, in fact, are located in the official libraries provided with Stata. You can create your own libraries. See [M-3] **mata mlib**.

.mo file

The object code of a function can be stored in a `.mo` file, where it can be later reused. See [M-1] **How** and [M-3] **mata mosave**.

monadic operator

Synonym for unary operator.

NaN

NaN stands for Not a Number and is a special computer floating-point code used for results that cannot be calculated. Mata (and Stata) do not use NaNs. When NaNs arise, they are converted into `.` (missing value).

norm

A norm is a real-valued function $f(x)$ satisfying

$$\begin{aligned} f(0) &= 0 \\ f(x) &> 0 \qquad \text{for all } x \neq 0 \\ f(cx) &= |c|f(x) \\ f(x+y) &\leq f(x)+f(y) \end{aligned}$$

The word *norm* applied to a vector x usually refers to its Euclidean norm, $p = 2$ norm, or length: the square root of the sum of its squared elements. The are other norms, the popular ones being $p = 1$ (the sum of the absolute values of its elements) and $p =$ infinity (the maximum element). Norms can also be generalized to deal with matrices. See [M-5] **norm( )**.

NULL

A special value for a *pointer* that means "points to nothing". If you list the contents of a pointer variable that contains `NULL`, the address will show as `0x0`. See *pointer*.

numeric

A matrix is said to be numeric if its elements are real or complex; see *type, eltype, and orgtype*.

object code

Object code refers to the binary code that Mata produces from the source code you type as input. See [M-1] **How**.

object-oriented programming

Object-oriented programming is a programming concept that treats programming elements as objects and concentrates on actions affecting those objects rather than merely on lists of instructions. Object-oriented programming uses classes to describe objects. Classes are much like structures with a primary difference being that classes can contain functions (known as methods) as well as variables. Unlike structures, however, classes may inherit variables and functions from other classes, which in theory makes object-oriented programs easier to extend and modify than non–object-oriented programs.

observations and variables

A dataset containing n observations on k variables in often stored in an $n \times k$ matrix. An observation refers to a row of that matrix; a variable refers to a column.

operator

An operator is `+`, `-`, and the like. Most operators are binary (or dyadic), such as `+` in *A*`+`*B* and `*` in *C*`*`*D*. Binary operators also include logical operators such as `&` and `|` ("and" and "or") in *E*`&`*F* and *G*`|`*H*. Other operators are unary (or monadic), such as `!` (not) in `!`*J*, or both unary and binary, such as `-` in `-`*K* and in *L*`-`*M*. When we say "operator" without specifying which, we mean binary operator. Thus colon operators are in fact colon binary operators. See [M-2] **exp**.

optimization

Mata compiles the code that you write. After compilation, Mata performs an *optimization* step, the purpose of which is to make the compiled code execute more quickly. You can turn off the optimization step—see [M-3] **mata set**—but doing so is not recommended.

orgtype

See *type, eltype, and orgtype*.

orthogonal matrix and unitary matrix

A is orthogonal if *A* is *square* and $A'A==I$. The word orthogonal is usually reserved for real matrices; if the matrix is complex, it is said to be unitary (and then transpose means conjugate-transpose). We use the word orthogonal for both real and complex matrices.

If *A* is orthogonal, then $\det(A) = \pm 1$.

p-conformability

Matrix, vector, or scalar A is said to be p-conformable with matrix, vector, or scalar B if `rows(`A`)==rows(`B`)` and `cols(`A`)==cols(`B`)`. p stands for plus; p-conformability is one of the properties necessary to be able to add matrices together. p-conformability, however, does not imply that the matrices are of the same type. Thus `(1,2,3)` is p-conformable with `(4,5,6)` and with `("this","that","what")` but not with `(4\5\6)`.

permutation matrix and permutation vector

A *permutation matrix* is an $n \times n$ matrix that is a row (or column) permutation of the identity matrix. If P is a permutation matrix, then $P*A$ permutes the rows of A and $A*P$ permutes the columns of A. Permutation matrices also have the property that $P^{-1} = P'$.

A *permutation vector* is a $1 \times n$ or $n \times 1$ vector that contains a permutation of the integers 1, 2, ..., n. Permutation vectors can be used with subscripting to reorder the rows or columns of a matrix. Permutation vectors are a memory-conserving way of recording permutation matrices; see [M-1] **Permutation**.

pointer

A matrix is said to be a pointer matrix if its elements are pointers.

A pointer is the address of a *variable*. Say that variable X contains a matrix. Another variable p might contain 137,799,016 and, if 137,799,016 were the address at which X were stored, then p would be said to point to X. Addresses are seldom written in base 10, and so rather than saying p contains 137,799,016, we would be more likely to say that p contains 0x836a568, which is the way we write numbers in base 16. Regardless of how we write addresses, however, p contains a number and that number corresponds to the address of another variable.

In our program, if we refer to p, we are referring to p's contents, the number 0x836a568. The monadic operator * is defined as "refer to the address" or "dereference": $*p$ means X. We could code `Y = *p` or `Y = X`, and either way, we would obtain the same result. In our program, we could refer to X`[`i`,`j`]` or `(*`p`)[`i`,`j`]`, and either way, we would obtain the i, j element of X.

The monadic operator `&` is how we put addresses into p. To load p with the address of X, we code p `= &`X.

The special address 0 (zero, written in hexadecimal as 0x0), also known as `NULL`, is how we record that a pointer variable points to nothing. A pointer variable contains `NULL` or it contains a valid address of another variable.

See [M-2] **pointers** for a complete description of pointers and their use.

pragma

"(Pragmatic information) A standardised form of comment which has meaning to a compiler. It may use a special syntax or a specific form within the normal comment syntax. A pragma usually conveys non-essential information, often intended to help the compiler to optimise the program." See *The Free On-line Dictionary of Computing*, http://foldoc.org/, Editor Denis Howe. For Mata, see [M-2] **pragma**.

rank

Terms in common use are rank, row rank, and column rank. The row rank of a matrix A: $m \times n$ is the number of rows of A that are linearly independent. The column rank is defined similarly, as the number of columns that are linearly independent. The terms *row rank* and *column rank*, however, are used merely for emphasis; the ranks are equal and the result is simply called the rank of A.

For a square matrix A (where m==n), the matrix is invertible if and only if `rank(`A`)==`n. One often hears that A is of full rank in this case and rank deficient in the other. See [M-5] **rank( )**.

r-conformability

A set of two or more matrices, vectors, or scalars A, B, ..., are said to be r-conformable if each is *c-conformable* with a matrix of `max(rows(`A`), rows(`B`), ...)` rows and `max(cols(`A`), cols(`B`), ...)` columns.

r-conformability is a more relaxed form of *c-conformability* in that, if two matrices are c-conformable, they are r-conformable, but not vice versa. For instance, A: 1×3 and B: 3×1 are r-conformable but not c-conformable. Also, c-conformability is defined with respect to a pair of matrices only; r-conformability can be applied to a set of matrices.

r-conformability is often required of the arguments for functions that would otherwise naturally be expected to require scalars. See *R-conformability* in [M-5] **normal( )** for an example.

real

A matrix is said to be a real matrix if its elements are all reals and it is stored in a `real` matrix. Real is one of the two numeric types in Mata, the other being complex. Also see *type, eltype, and orgtype*.

row and column stripes

Stripes refer to the labels associated with the rows and columns of a Stata matrix; see *Stata matrix*.

row-major order

Matrices are stored as vectors. Row-major order specifies that the vector form of a matrix is created by stacking the rows. For instance,

```
: A
        1   2   3
    +-------------+
  1 |   1   2   3 |
  2 |   4   5   6 |
    +-------------+
```

is stored as

```
        1   2   3   4   5   6
    +-----------------------------+
  1 |   1   2   3   4   5   6     |
    +-----------------------------+
```

in row-major order. Mata uses row-major order. The LAPACK functions use column-major order. See *column-major order*.

rowvector

See *vector, colvector, and rowvector*.

scalar

A special case of a *matrix* with one row and one column. A scalar may be substituted anywhere a matrix, vector, column vector, or row vector is required, but not vice versa.

Schur decomposition

The Schur decomposition of a matrix, $\mathbf{A}$, can be written as

$$\mathbf{Q}'\mathbf{A}\mathbf{Q} = \mathbf{T}$$

where $\mathbf{T}$ is in Schur form and $\mathbf{Q}$, the matrix of Schur vectors, is orthogonal if $\mathbf{A}$ is real or unitary if $\mathbf{A}$ is complex. See [M-5] **schurd()**.

Schur form

There are two Schur forms: real Schur form and complex Schur form.

A real matrix is in Schur form if it is block upper triangular with 1×1 and 2×2 diagonal blocks. Each 2×2 diagonal block has equal diagonal elements and opposite sign off-diagonal elements. The real eigenvalues are on the diagonal and complex eigenvalues can be obtained from the 2×24 diagonal blocks.

A complex square matrix is in Schur form if it is upper triangular with the eigenvalues on the diagonal.

source code

Source code refers to the human-readable code that you type into Mata to define a function. Source code is compiled into object code, which is binary. See [M-1] **How**.

square matrix

A matrix is square if it has the same number of rows and columns. A 3×3 matrix is square; a 3×4 matrix is not.

Stata matrix

Stata itself, separate from Mata, has matrix capabilities. Stata matrices are separate from those of Mata, although Stata matrices can be gotten from and put into Mata matrices; see [M-5] **st_matrix()**. Stata matrices are described in [P] **matrix** and [U] **14 Matrix expressions**.

Stata matrices are exclusively numeric and contain real elements only. Stata matrices also differ from Mata matrices in that, in addition to the matrix itself, a Stata matrix has text labels on the rows and columns. These labels are called row stripes and column stripes. One can think of rows and columns as having names. The purpose of these names is discussed in [U] **14.2 Row and column names**. Mata matrices have no such labels. Thus three steps are required to get or to put all the information recorded in a Stata matrix: 1) getting or putting the matrix itself; 2) getting or putting the row stripe from or into a string matrix; and 3) getting or putting the column stripe from or into a string matrix. These steps are discussed in [M-5] **st_matrix( )**.

string

A matrix is said to be a string matrix if its elements are strings (text); see *type, eltype, and orgtype*. In Mata, a string may be text or binary and may be up to 2,147,483,647 characters (bytes) long.

structure

A structure is an *eltype*, indicating a set of variables tied together under one name. `struct mystruct` might be

```
struct mystruct {
        real scalar     n1, n2
        real matrix     X
}
```

If variable `a` was declared a `struct mystruct scalar`, then the scalar `a` would contain three pieces: two real scalars and one real matrix. The pieces would be referred to as `a.n1`, `a.n2`, and `a.X`. If variable `b` were also declared a `struct mystruct scalar`, it too would contain three pieces, `b.n1`, `b.n2`, and `b.X`. The advantage of structures is that they can be referred to as a whole. You can code `a.n1=b.n1` to copy one piece, or you can code `a=b` if you wanted to copy all three pieces. In all ways, `a` and `b` are variables. You may pass `a` to a subroutine, for instance, which amounts to passing all three values.

Structures variables are usually scalar, but they are not limited to being so. If `A` were a `struct mystruct matrix`, then each element of `A` would contain three pieces, and one could refer, for instance, to `A[2,3].n1`, `A[2,3].n2`, and `A[2,3].X`, and even to `A[2,3].X[3,2]`.

See [M-2] **struct**.

subscripts

Subscripts are how you refer to an element or even a submatrix of a matrix.

Mata provides two kinds of subscripts, known as list subscripts and range subscripts.

In list subscripts, *A*`[2,3]` refers to the (2,3) element of *A*. *A*`[(2\3), (4,6)]` refers to the submatrix made up of the second and third rows, fourth and sixth columns, of *A*.

In range subscripts, *A*`[|2,3|]` also refers to the (2,3) element of *A*. *A*`[|2,3\4,6|]` refers to the submatrix beginning at the (2,3) element and ending at the (4,6) element.

See [M-2] **Subscripts** for more information.

symmetric matrices

Matrix A is symmetric if $A = A'$. The word *symmetric* is usually reserved for real matrices, and in that case, a symmetric matrix is a square matrix with a_{ij}==a_{ji}.

Matrix A is said to be Hermitian if $A = A'$, where the transpose operator is understood to mean the conjugate-transpose operator; see *Hermitian matrix*. In Mata, the $'$ operator is the conjugate-transpose operator, and thus, in this manual, we will use the word *symmetric* both to refer to real, symmetric matrices and to refer to complex, Hermitian matrices.

Sometimes, you will see us follow the word *symmetric* with a parenthesized Hermitian, as in, "the resulting matrix is symmetric (Hermitian)". That is done only for emphasis.

The inverse of a symmetric (Hermitian) matrix is symmetric (Hermitian).

symmetriconly

Symmetriconly is a word we have coined to refer to a square matrix whose corresponding off-diagonal elements are equal to each other, whether the matrix is real or complex. Symmetriconly matrices have no mathematical significance, but sometimes, in data-processing and memory-management routines, it is useful to be able to distinguish such matrices.

time-series–operated variable

Time-series–operated variables are a Stata concept. The term refers to *op*`.`*varname* combinations such as `L.gnp` to mean the lagged value of variable `gnp`. Mata's [M-5] **st_data()** function works with time-series–operated variables just as it works with other variables, but many other Stata-interface functions do not allow *op*`.`*varname* combinations. In those cases, you must use [M-5] **st_tsrevar()**.

titlecase

Titlecasing is a Unicode concept implemented in Mata in the `ustrtitle()` function. To "titlecase" a phrase means to convert to Unicode titlecase the first letter of each Unicode word. This is almost, but not exactly, like capitalizing the first letter of each Unicode word. Like capitalization, titlecasing letters is locale-dependent, which means that the same letter might have different titlecase forms in different locales. In some locales, the titlecase form of a letter is different than the capital form of that same letter. For example, in some locales, capital letters at the beginning of words are not supposed to have accents on them, even if that capital letter by itself would have an accent.

traceback log

When a function fails—either because of a programming error or because it was used incorrectly—it produces a traceback log:

```
: myfunction(2,3)
              solve():  3200  conformability error
              mysub():     -  function returned error
         myfunction():     -  function returned error
              <istmt>:     -  function returned error
r(3200);
```

The log says that `solve()` detected the problem—arguments are not conformable—and that `solve()` was called by `mysub()` was called by `myfunction()` was called by what you typed at the keyboard. See [M-2] **Errors** for more information.

transmorphic

Transmorphic is an *eltype*. A scalar, vector, or matrix can be transmorphic, which indicates that its elements may be real, complex, string, pointer, or even a structure. The elements are all the same type; you are just not saying which they are. Variables that are not declared are assumed to be transmorphic, or a variable can be explicitly declared to be `transmorphic`. Transmorphic is just fancy jargon for saying that the elements of the scalar, vector, or matrix can be anything and that, from one instant to the next, the scalar, vector, or matrix might change from holding elements of one type to elements of another.

See [M-2] **Declarations**.

transpose

The transpose operator is written different ways in different books, including $'$, superscript *, superscript T, and superscript H. Here we use the $'$ notation: A' means the transpose of A, A with its rows and columns interchanged.

In complex analysis, the transpose operator, however it is written, is usually defined to mean the conjugate transpose; that is, one interchanges the rows and columns of the matrix and then one takes the conjugate of each element, or one does it in the opposite order—it makes no difference. Conjugation simply means reversing the sign of the imaginary part of a complex number: the conjugate of `1+2i` is `1-2i`. The conjugate of a real is the number itself; the conjugate of 2 is 2.

In Mata, $'$ is defined to mean conjugate transpose. Since the conjugate of a real is the number itself, A' is regular transposition when A is real. Similarly, we have defined $'$ so that it performs regular transposition for string and pointer matrices. For complex matrices, however, $'$ also performs conjugation.

If you have a complex matrix and simply want to transpose it without taking the conjugate of its elements, see [M-5] **transposeonly( )**. Or code `conj(A')`. The extra `conj()` will undo the undesired conjugation performed by the transpose operator.

Usually, however, you want transposition and conjugation to go hand in hand. Most mathematical formulas, generalized to complex values, work that way.

triangular matrix

A triangular matrix is a matrix with all elements equal to zero above the diagonal or all elements equal to zero below the diagonal.

A matrix A is *lower triangular* if all elements are zero above the diagonal, that is, if $A[i,j]==0$, $j > i$.

A matrix A is *upper triangular* if all elements are zero below the diagonal, that is, if $A[i,j]==0$, $j < i$.

A *diagonal matrix* is both lower and upper triangular. That is worth mentioning because any function suitable for use with triangular matrices is suitable for use with diagonal matrices.

A triangular matrix is usually *square*.

The inverse of a triangular matrix is a triangular matrix. The determinant of a triangular matrix is the product of the diagonal elements. The eigenvalues of a triangular matrix are the diagonal elements.

type, eltype, and orgtype

The *type* of a matrix (or vector or scalar) is formally defined as the matrix's *eltype* and *orgtype*, listed one after the other—such as `real vector`—but it can also mean just one or the other—such as the *eltype* `real` or the *orgtype* `vector`.

eltype refers to the type of the elements. The *eltypes* are

`real`	numbers such as 1, 2, 3.4
`complex`	numbers such as 1+2i, 3+0i
`string`	strings such as `"bill"`
`pointer`	pointers such as *&varname*
`struct`	structures
`numeric`	meaning `real` or `complex`
`transmorphic`	meaning any of the above

orgtype refers to the organizational type. *orgtype* specifies how the elements are organized. The *orgtypes* are

`matrix`	two-dimensional arrays
`vector`	one-dimensional arrays
`colvector`	one-dimensional column arrays
`rowvector`	one-dimensional row arrays
`scalar`	single items

The fully specified type is the element and organization types combined, as in `real vector`.

unary operator

A unary operator is an operator applied to one argument. In `-2`, the minus sign is a unary operator. In `!(`*a*`==`*b* `|` *a*`==`*c*`)`, `!` is a unary operator.

underscore functions

Functions whose names start with an underscore are called underscore functions, and when an underscore function exists, usually a function without the underscore prefix also exists. In those cases, the function is usually implemented in terms of the underscore function, and the underscore function is harder to use but is faster or provides greater control. Usually, the difference is in the handling of errors.

For instance, function `fopen()` opens a file. If the file does not exist, execution of your program is aborted. Function `_fopen()` does the same thing, but if the file cannot be opened, it returns a special value indicating failure, and it is the responsibility of your program to check the indicator and to take the appropriate action. This can be useful when the file might not exist, and if it does not, you wish to take a different action. Usually, however, if the file does not exist, you will wish to abort, and use of `fopen()` will allow you to write less code.

unitary matrix

See *orthogonal matrix*.

UTF-8

UTF-8 is the way of encoding Unicode characters chosen by Stata for its strings. It is backward compatible with ASCII encoding in the sense that plain ASCII characters are encoded the same in UTF-8 as in ASCII and that strings are still null terminated. Characters beyond plain ASCII are encoded using two to four bytes per character. As with other Unicode encodings, all possible Unicode characters (code points) can be represented by UTF-8.

variable

In a program, the entities that store values (a, b, c, ..., x, y, z) are called variables. Variables are given names of 1 to 32 characters long. To be terribly formal about it: a variable is a container; it contains a matrix, vector, or scalar and is referred to by its variable name or by another variable containing a *pointer* to it.

Also, *variable* is sometimes used to refer to columns of data matrices; see *data matrix*.

vector, colvector, and rowvector

A special case of a matrix with either one row or one column. A vector may be substituted anywhere a matrix is required. A matrix, however, may not be substituted for a vector.

A `colvector` is a vector with one column.

A `rowvector` is a vector with one row.

A `vector` is either a `rowvector` or `colvector`, without saying which.

view

A view is a special type of matrix that appears to be an ordinary matrix, but in fact the values in the matrix are the values of certain or all variables and observations in the Stata dataset that is currently in memory. Its values are not just equal to the dataset's values; they are the dataset's values: if an element of the matrix is changed, the corresponding variable and observation in the Stata dataset also changes. Views are obtained by `st_view()` and are efficient; see [M-5] **st_view( )**.

void function

A function is said to be void if it returns nothing. For instance, the function [M-5] **printf( )** is a void function; it prints results, but it does not return anything in the sense that, say, [M-5] **sqrt( )** does. It would not make any sense to code `x = printf("hi there")`, but coding `x = sqrt(2)` is perfectly logical.

void matrix

A matrix is said to be void if it is 0×0, $r \times 0$, or $0 \times c$; see [M-2] **void**.

Also see

[M-0] **Intro** — Introduction to the Mata manual

[M-1] **Intro** — Introduction and advice