NDMatrix( ) — N-dimensional matrix+

+This function is part of StataNow.

Description Syntax Remarks and examples Conformability Diagnostics Also see

Description

The `NDMatrix()` class defines a multidimensional matrix of transmorphic type.

Syntax

Syntax is presented under the following headings:

Step 1: Initialization
Step 2: Set data and dimension
Step 3: Manipulate data
Definition of q
Set data and dimension
  q.setData( ) and q.getData( )
  q.setSize( ) and q.getSize( )
Manipulate data
  q.ndReshape( )
  q.setElement( ) and q.getElement( )
  q.get1dIndices( )
  q.set1dData( ) and q.get1dData( )
  q.to1did( ) and q.tomultid( )

Step 1: Initialization

q = `NDMatrix()`

Step 2: Set data and dimension

void	*q*`.setData(`*transmorphic vector rawdata*`)`
void	*q*`.setSize(`*real vector s*`)`
transmorphic colvector	*q*`.getData()`
real colvector	*q*`.getSize()`

Step 3: Manipulate data

void	*q*`.ndReshape(`*real vector newsz*`)`
void	*q*`.setElement(`*real vector id*`,` *transmorphic scalar val*`)`
transmorphic scalar	*q*`.getElement(`*real vector id*`)`
real colvector	*q*`.get1dIndices(`*real scalar dim*`,` *real scalar id*`)`
void	*q*`.set1dData(`*real scalar dim*`,` *real scalar id*`,` *transmorphic vector val*`)`
transmorphic colvector	*q*`.get1dData(`*real scalar dim*`,` *real scalar id*`)`
real scalar	*q*`.to1did(`*real vector id*`)`
real colvector	*q*`.tomultid(`*real scalar id1d*`)`

Definition of q

A variable of type `NDMatrix` is called an instance of the `NDMatrix()` class. *q* is an instance of `NDMatrix()`, a vector of instances, or a matrix of instances. If you are working interactively, you can create an instance of `NDMatrix()` by typing

```
q = NDMatrix()
```

For a row vector of *n* `NDMatrix()` instances, type

```
q = NDMatrix(n)
```

For an $m \times n$ matrix of `NDMatrix()` instances, type

```
q = NDMatrix(m, n)
```

In a function, you would declare one instance of the `NDMatrix()` class *q* as a `scalar`.

```
void myfunc()
{
    class NDMatrix scalar   q
    q = NDMatrix()
    ...
}
```

Again within a function, you can declare *q* as a row vector of *n* instances by typing

```
void myfunc()
{
    class NDMatrix rowvector   q
    q = NDMatrix(n)
    ...
}
```

For an $m \times n$ matrix of instances in a function, type

```
void myfunc()
{
    class NDMatrix matrix   q
    q = NDMatrix(m, n)
    ...
}
```

Set data and dimension

At a minimum, you need to tell the `NDMatrix()` class about the multidimensional data as a vector and the dimension of those data as a vector of integers.

Each pair of functions includes a *q*`.set` function that specifies a setting and a *q*`.get` function that returns the current setting.

q.setData( ) and q.getData( )

q`.setData(`*rawdata*`)` sets the multidimensional data as a vector. The data can be of any type.

q`.getData()` returns the data as a column vector (or an empty vector if the input data name is not specified).

q.setSize( ) and q.getSize( )

q`.setSize(`*s*`)` sets the lengths for each of the dimensions in *s* as a vector. All the values in *s* must be positive integers.

This function also allows one of the lengths to be determined automatically by leaving the corresponding value in *s* as missing.

Note that this function must be called after *q*`.setData()`. If the automatic dimension setting is not used, the total length of the data stored must be the same as the product of all the elements in *s*. If autodetermination is used, then the quotient of the total length of the data stored and the product of all the other elements in *s* must be a positive integer.

q`.getSize()` returns the lengths for each of the dimensions as a column vector.

Manipulate data

q.ndReshape( )

q`.ndReshape(`*newsz*`)` sets the data to a new size, *newsz*.

For example, if the data are a (2, 3) matrix, we can use this function to reshape it to a (1, 6) matrix.

All the requirements stated in `q.setSize()` and `q.getSize()` also apply here.

q.setElement( ) and q.getElement( )

q`.setElement(`*id*`, `*val*`)` sets the value of the element at *id* as *val*. Here the matrix index in *id* is given as a vector of the same dimension as the data.

q`.getElement(`*id*`)` returns the value of the element at *id*.

q.get1dIndices( )

q`.get1dIndices(`*dim*`, `*id*`)` returns a column vector of all the indices for a specific *id* in a specific *dim*.

q.set1dData( ) and q.get1dData( )

q`.set1dData(`*dim*`, `*id*`, `*val*`)` sets the values of all the elements at a specific *id* in a specific *dim* as *val*. Here the length of *val* must be the same as the number of elements at *id* in *dim*.

q`.get1dData(`*dim*`, `*id*`)` returns the values of all the elements at a specific *id* in a specific *dim*.

q.to1did( ) and q.tomultid( )

Because the data are represented as a column vector internally, the multidimensional index must be transformed into the corresponding one-dimensional index to set or get elements from the data.

q`.to1did(`*id*`)` returns the multidimensional index corresponding to one-dimensional index *id*.

q`.tomultid(`*id1d*`)` returns the one-dimensional index corresponding to multidimensional index *id1d*.

Remarks and examples

The `NDMatrix()` class is a Mata class for multidimensional matrices.

Normally, a Mata matrix is at most two dimensional. Using this class, we can handle multidimensional matrices. The data are stored as a column vector, and a corresponding size vector is also stored to indicate the actual dimension.

▷ Example 1: Class usage example

Consider the following three-dimensional matrix of size $(2 \times 3 \times 4)$:

$$\begin{bmatrix} \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \\ \begin{bmatrix} m & n & o & p \\ q & r & s & t \\ u & v & w & x \end{bmatrix} \end{bmatrix}$$

We first define an instance of the `NDMatrix()` class:

```
: q = NDMatrix()
```

Then we set the matrix element values and the size of the matrix:

```
: data = ("a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k",
>         "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x")
: size = (2, 3, 4)
: q.setData(data)
: q.setSize(size)
```

Recall that values and size are stored in the NDMatrix() class as column vectors, so we have the following:

```
      : q.getData()
               1
          +-------+
        1 |   a   |
        2 |   b   |
        3 |   c   |
        4 |   d   |
        5 |   e   |
        6 |   f   |
        7 |   g   |
        8 |   h   |
        9 |   i   |
       10 |   j   |
       11 |   k   |
       12 |   l   |
       13 |   m   |
       14 |   n   |
       15 |   o   |
       16 |   p   |
       17 |   q   |
       18 |   r   |
       19 |   s   |
       20 |   t   |
       21 |   u   |
       22 |   v   |
       23 |   w   |
       24 |   x   |
          +-------+

      : q.getSize()
             1
         +-----+
       1 |  2  |
       2 |  3  |
       3 |  4  |
         +-----+
```

Because the data are actually stored as a column vector, it may be useful to transform an N-dimensional index to the one-dimensional index used internally. For example, the index $(2, 1, 3)$ corresponds to the one-dimensional index:

```
      : q.to1did((2, 1, 3))
        15
```

To obtain the N-dimensional index that corresponds to the one-dimensional index 15, we type

```
      : q.tomultid(15)
             1
         +-----+
       1 |  2  |
       2 |  1  |
       3 |  3  |
         +-----+
```

We can obtain the element at $(2, 1, 3)$ by typing

```
      : q.getElement((2, 1, 3))
        o
```

We can set and then get the element at $(2, 1, 3)$ to oo by typing

```
: q.setElement((2, 1, 3), "oo")
: q.getElement((2, 1, 3))
  oo
```

We may want to get the data for a specified dimension all at once. For example, to get all the elements at index 3 of dimension 3, we type

```
: q.get1dData(3, 3)
        1
    +-------+
  1 |    c  |
  2 |    g  |
  3 |    k  |
  4 |   oo  |
  5 |    s  |
  6 |    w  |
    +-------+
```

In addition, to obtain all the indices instead of the actual data at index 3 of dimension 3 in one-dimensional form, we type

```
: q.get1dIndices(3, 3)
        1
    +-------+
  1 |    3  |
  2 |    7  |
  3 |   11  |
  4 |   15  |
  5 |   19  |
  6 |   23  |
    +-------+
```

We can set these elements all at once by typing

```
: q.set1dData(3, 3, ("cc", "gg", "kk", "oo", "ss", "ww"))
: q.get1dData(3, 3)
        1
    +-------+
  1 |   cc  |
  2 |   gg  |
  3 |   kk  |
  4 |   oo  |
  5 |   ss  |
  6 |   ww  |
    +-------+
```

We can reshape the matrix to a new dimension. For example, to set the new size of our matrix to $(4, 3, 2)$, we type

```
: q.ndReshape((4, 3, .))
: q.getSize()
       1
    +-----+
  1 |  4  |
  2 |  3  |
  3 |  2  |
    +-----+
```

Recall that `setSize()` and `ndReshape()` allow one of the lengths to be determined automatically by leaving the corresponding value as missing, as in the previous example.

Note that if the automatic dimension setting is not used, the total length of the data stored must be the same as the product of all the elements in s. If autodetermination is used, then the quotient of the total length of the data stored and the product of all the other elements in s must be a positive integer.

◁

Conformability

`NDMatrix()`:

input:

	void

output:

result:	1×1

`NDMatrix(`*n*`)`:

input:

n:	1×1

output:

result:	$1 \times n$

`NDMatrix(`*m*`,` *n*`)`:

input:

m:	1×1
n:	1×1

output:

result:	$m \times n$

`setData(`*rawdata*`)`:

input:

rawdata:	$1 \times M$ or $M \times 1$

output:

result:	*void*

`getData()`:

input:

	void

output:

result:	$M \times 1$

`setSize(`*s*`)`:

input:

object:	$1 \times N$ or $N \times 1$

output:

result:	*void*

`getSize()`:

input:

	void

output:

result:	$N \times 1$

`ndReshape(`*newsz*`)`:

input:

newsz:	$1 \times N1$ or $N1 \times 1$

output:

result:	*void*

`setElement(`*id*`, `*val*`)`:

input:

id:	$1 \times N2$ or $N2 \times 1$
val:	1×1

output:

result:	*void*

`getElement()`:

input:

id:	$1 \times N2$ or $N2 \times 1$

output:

result:	1×1

`get1dIndices(`*dim*`, `*id*`)`:

input:

dim:	1×1
id:	1×1

output:

result:	$N3 \times 1$

`set1dData(`*dim*`, `*id*`, `*val*`)`:

input:

dim:	1×1
id:	1×1
val:	$1 \times N3$ or $N3 \times 1$

output:

result:	*void*

`get1dData(`*dim*`,` *id*`):`

input:		
	dim:	1×1
	id:	1×1
output:		
	result:	$N3 \times 1$

`to1did(`*id*`):`

input:		
	id:	$1 \times N2$ or $N2 \times 1$
output:		
	result:	1×1

`tomultid(`*id1d*`):`

input:		
	id:	1×1
output:		
	result:	$N2 \times 1$

Diagnostics

`NDMatrix()`, *q*`.set*()`, *q*`.get*()`, *q*`.ndReshape()`, *q*`.to1did()`, and *q*`.tomultid()` functions abort with an error message when used incorrectly.

Also see

[M-4] **Programming** — Programming functions