Creating and Concatenating Matrices

MATLAB is a matrix-based computing environment. All of the data that you enter into MATLAB is stored in the form of a matrix or a multidimensional array. Even a single numeric value like `100` is stored as a matrix (in this case, a matrix having dimensions 1-by-1):

```
A = 100;

whos A
  Name      Size                    Bytes  Class

  A         1x1                         8  double array
```

Regardless of the data type being used, whether it is numeric, character, or logical `true` or `false` data, MATLAB stores this data in matrix (or array) form. For example, the string `'Hello World'` is a 1-by-11 matrix of individual character elements in MATLAB. You can also build matrices composed of more complex data types, such as MATLAB structures and cell arrays.

To create a matrix of basic data elements such as numbers or characters, see

- "Constructing a Simple Matrix" on page 1-4
- "Specialized Matrix Functions" on page 1-4

To build a matrix composed of other matrices, see

- "Concatenating Matrices" on page 1-7
- "Matrix Concatenation Functions" on page 1-8

This section also describes

- "Generating a Numeric Sequence" on page 1-10
- "Combining Unlike Data Types" on page 1-11

Constructing a Simple Matrix

The simplest way to create a matrix in MATLAB is to use the matrix constructor operator, `[]`. Create a row in the matrix by entering elements (shown as `E` below) within the brackets. Separate each element with a comma or space:

```
row = [E1, E2, ..., Em]          row = [E1 E2 ... Em]
```

For example, to create a one row matrix of five elements, type

```
A = [12 62 93 -8 22];
```

To start a new row, terminate the current row with a semicolon:

```
A = [row1; row2; ...; rown]
```

This example constructs a 3 row, 5 column (or 3-by-5) matrix of numbers. Note that all rows must have the same number of elements:

```
A = [12 62 93 -8 22; 16 2 87 43 91; -4 17 -72 95 6]
A =
    12    62    93    -8    22
    16     2    87    43    91
    -4    17   -72    95     6
```

The square brackets operator constructs two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To construct arrays of more than two dimensions, see “Creating Multidimensional Arrays” on page 1-39.

For instructions on how to read or overwrite any matrix element, see “Accessing Elements of a Matrix” on page 1-14.

Specialized Matrix Functions

MATLAB has a number of functions that create different kinds of matrices. Some create specialized matrices like the Hankel or Vandermonde matrix. The functions shown in the table below create a matrices for more general use.

Function	Description
`ones`	Create a matrix or array of all ones.
`zeros`	Create a matrix or array of all zeros.

Function	Description
`eye`	Create a matrix with ones on the diagonal and zeros elsewhere.
`accumarray`	Distribute elements of an input matrix to specified locations in an output matrix, also allowing for accumulation.
`diag`	Create a diagonal matrix from a vector.
`magic`	Create a square matrix with rows, columns, and diagonals that add up to the same number.
`rand`	Create a matrix or array of uniformly distributed random numbers.
`randn`	Create a matrix or array of normally distributed random numbers and arrays.
`randperm`	Create a vector (1-by-n matrix) containing a random permutation of the specified integers.

Most of these functions return matrices of type `double` (double-precision floating point). However, you can easily build basic arrays of any numeric type using the `ones`, `zeros`, and `eye` functions.

To do this, specify the MATLAB class name as the last argument:

```
A = zeros(4, 6, 'uint32')
A =
        0        0        0        0        0        0
        0        0        0        0        0        0
        0        0        0        0        0        0
        0        0        0        0        0        0
```

Examples

Here are some examples of how you can use these functions.

Creating a Magic Square Matrix. A magic square is a matrix in which the sum of the elements in each column, or each row, or each main diagonal is the same. To create a 5-by-5 magic square matrix, use the `magic` function as shown.

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

Note that the elements of each row, each column, and each main diagonal add up to the same value: 65.

Creating a Random Matrix. The `rand` function creates a matrix or array with elements uniformly distributed between zero and one. This example multiplies each element by 20:

```
A = rand(5) * 20
A =
    3.8686   13.9580    9.9310   13.2046   14.5423
   13.6445    7.5675   17.9954    6.8394    6.1858
    6.0553   17.2002   16.4326    5.7945   16.7699
   10.8335   17.0731   12.8982    6.8239   11.3614
    3.0175   11.8713   16.3595   10.6816    7.4083
```

Creating a Diagonal Matrix. Use `diag` to create a diagonal matrix from a vector. You can place the vector along the main diagonal of the matrix, or on a diagonal that is above or below the main one, as shown here. The `-1` input places the vector one row below the main diagonal:

```
A = [12 62 93 -8 22];

B = diag(A, -1)
B =
     0     0     0     0     0     0
    12     0     0     0     0     0
     0    62     0     0     0     0
     0     0    93     0     0     0
     0     0     0    -8     0     0
     0     0     0     0    22     0
```

Concatenating Matrices

Matrix concatenation is the process of joining one or more matrices to make a new matrix. The brackets `[]` operator discussed earlier in this section serves not only as a matrix constructor, but also as the MATLAB concatenation operator. The expression `C = [A B]` horizontally concatenates matrices `A` and `B`. The expression `C = [A; B]` vertically concatenates them.

This example constructs a new matrix `C` by concatenating matrices `A` and `B` in a vertical direction:

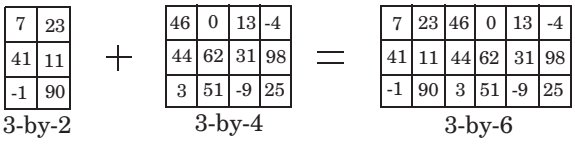
```
A = ones(2, 5) * 6;        % 2-by-5 matrix of 6 s
B = rand(3, 5);            % 3-by-5 matrix of random values

C = [A; B]                 % Vertically concatenate A and B
C =
    6.0000    6.0000    6.0000    6.0000    6.0000
    6.0000    6.0000    6.0000    6.0000    6.0000
    0.6154    0.7382    0.9355    0.8936    0.8132
    0.7919    0.1763    0.9169    0.0579    0.0099
    0.9218    0.4057    0.4103    0.3529    0.1389
```

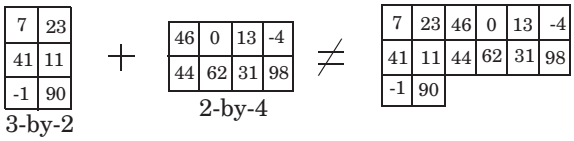
Keeping Matrices Rectangular

You can construct matrices, or even multidimensional arrays, using concatenation as long as the resulting matrix does not have an irregular shape (as in the second illustration shown below). If you are building a matrix horizontally, then each component matrix must have the same number of rows. When building vertically, each component must have the same number of columns.

This diagram shows two matrices of the same height (i.e., same number of rows) being combined horizontally to form a new matrix.



The next diagram illustrates an attempt to horizontally combine two matrices of unequal height. MATLAB does not allow this.



Matrix Concatenation Functions

The following functions combine existing matrices to form a new matrix.

Function	Description
`cat`	Concatenate matrices along the specified dimension
`horzcat`	Horizontally concatenate matrices
`vertcat`	Vertically concatenate matrices
`repmat`	Create a new matrix by replicating and tiling existing matrices
`blkdiag`	Create a block diagonal matrix from existing matrices

Examples

Here are some examples of how you can use these functions.

Concatenating Matrices and Arrays. An alternative to using the `[]` operator for concatenation are the three functions `cat`, `horzcat`, and `vertcat`. With these functions, you can construct matrices (or multidimensional arrays) along a specified dimension. Either of the following commands accomplish the same task as the command `C = [A; B]` used in the section on "Concatenating Matrices" on page 1-7:

```
C = cat(1, A, B);       % Concatenate along the first dimension
C = vertcat(A, B);      % Concatenate vertically
```

Replicating a Matrix. Use the `repmat` function to create a matrix composed of copies of an existing matrix. When you enter

```
repmat(M, v, h)
```

MATLAB replicates input matrix `M` `v` times vertically and `h` times horizontally. For example, to replicate existing matrix `A` into a new matrix `B`, use

```
A = [8 1 6; 3 5 7; 4 9 2]
A =
   8   1   6
   3   5   7
   4   9   2

B = repmat(A, 2, 4)
B =
   8   1   6   8   1   6   8   1   6   8   1   6
   3   5   7   3   5   7   3   5   7   3   5   7
   4   9   2   4   9   2   4   9   2   4   9   2
   8   1   6   8   1   6   8   1   6   8   1   6
   3   5   7   3   5   7   3   5   7   3   5   7
   4   9   2   4   9   2   4   9   2   4   9   2
```

Creating a Block Diagonal Matrix. The `blkdiag` function combines matrices in a diagonal direction, creating what is called a block diagonal matrix. All other elements of the newly created matrix are set to zero:

```
A = magic(3);
B = [-5 -6 -9; -4 -4 -2];
C = eye(2) * 8;

D = blkdiag(A, B, C)
D =
   8   1   6   0   0   0   0   0
   3   5   7   0   0   0   0   0
   4   9   2   0   0   0   0   0
   0   0   0  -5  -6  -9   0   0
   0   0   0  -4  -4  -2   0   0
   0   0   0   0   0   0   8   0
   0   0   0   0   0   0   0   8
```

Generating a Numeric Sequence

Because numeric sequences can often be useful in constructing and indexing into matrices and arrays, MATLAB provides a special operator to assist in creating them.

This section covers

- "The Colon Operator"
- "Using the Colon Operator with a Step Value"

The Colon Operator

The colon operator (`first:last`) generates a 1-by-n matrix (or *vector*) of sequential numbers from the `first` value to the `last`. The default sequence is made up of incremental values, each 1 greater than the previous one:

```
A = 10:15
A =
    10    11    12    13    14    15
```

The numeric sequence does not have to be made up of positive integers. It can include negative numbers and fractional numbers as well:

```
A = -2.5:2.5
A =
   -2.5000   -1.5000   -0.5000    0.5000    1.5000    2.5000
```

By default, MATLAB always increments by exactly 1 when creating the sequence, even if the ending value is not an integral distance from the start:

```
A = 1:6.3
A =
     1     2     3     4     5     6
```

Also, the default series generated by the colon operator always increments rather than decrementing. The operation shown in this example attempts to increment from 9 to 1 and thus MATLAB returns an empty matrix:

```
A = 9:1
A =
   Empty matrix: 1-by-0
```

The next section explains how to generate a nondefault numeric series.

Using the Colon Operator with a Step Value

To generate a series that does not use the default of incrementing by 1, specify an additional value with the colon operator (`first:step:last`). In between the starting and ending value is a `step` value that tells MATLAB how much to increment (or decrement, if `step` is negative) between each number it generates.

To generate a series of numbers from 10 to 50, incrementing by 5, use

```
A = 10:5:50
A =
    10    15    20    25    30    35    40    45    50
```

You can increment by noninteger values. This example increments by `0.2`:

```
A = 3:0.2:3.8
A =
    3.0000    3.2000    3.4000    3.6000    3.8000
```

To create a sequence with a decrementing interval, specify a negative step value:

```
A = 9:-1:1
A =
    9    8    7    6    5    4    3    2    1
```

Combining Unlike Data Types

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike data types when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See “Data Types” on page 2-1 for information on any of the MATLAB data types discussed here.)

Data type conversion is done with respect to a preset precedence of data types. The following table shows the five data types you can concatenate with an unlike type without generating an error.

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a `double` and `single` matrix always yields a matrix of type `single`. MATLAB converts the `double` element to `single` to accomplish this.

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Examples

Here are some examples of data type conversion during matrix construction.

Combining Single and Double Types. Combining `single` values with `double` values yields a `single` matrix. Note that `5.73*10^300` is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value):

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000   -2.8000    3.1416       Inf
```

```
class(x)              % Display the data type of x
ans =
   single
```

Combining Integer and Double Types. Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer):

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21   -22    23     3     7

class(x)
ans =
   int8
```

Combining Character and Double Types. Combining `character` values with `double` values yields a `character` matrix. MATLAB converts the `double` elements in this example to their `character` equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
   ABCDEF

class(x)
ans =
   char
```

Combining Logical and Double Types. Combining `logical` values with `double` values yields a `double` matrix. MATLAB converts the `logical` `true` and `false` elements in this example to `double`:

```
x = [true false false pi sqrt(7)]
x =
    1.0000         0         0    3.1416    2.6458

class(x)
ans =
   double
```

Accessing Elements of a Matrix

This section explains how to use subscripting and indexing to access and assign values to the elements of a MATLAB matrix. It covers the following:

Accessing Single Elements

To reference a particular element in a matrix, specify its row and column number using the following syntax, where `A` is the matrix variable. Always specify the row first and column second:

```
A(row, column)
```

For example, for a 4-by-4 magic square `A`,

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

you would access the element at row 4, column 2 with

```
A(4, 2)
ans =
    14
```

For arrays with more than two dimensions, specify additional indices following the `row` and `column` indices. See the section on "Multidimensional Arrays" on page 1-37.

Linear Indexing

With MATLAB, you can refer to the elements of a matrix with a single subscript, `A(k)`. MATLAB stores matrices and arrays not in the shape that they appear when displayed in the MATLAB Command Window, but as a single column of elements. This single column is composed of all of the columns from the matrix, each appended to the last.

So, matrix `A`

```
A = [2 6 9; 4 2 8; 3 0 1]
A =
     2     6     9
     4     2     8
     3     5     1
```

is actually stored in memory as the sequence

```
2, 4, 3, 6, 2, 5, 9, 8, 1
```

The element at row 3, column 2 of matrix `A` (value = `5`) can also be identified as element 6 in the actual storage sequence. To access this element, you have a choice of using the standard `A(3,2)` syntax, or you can use `A(6)`, which is referred to as *linear indexing*.

If you supply more subscripts, MATLAB calculates an index into the storage column based on the dimensions you assigned to the array. For example, assume a two-dimensional array like `A` has size `[d1 d2]`, where `d1` is the number of rows in the array and `d2` is the number of columns. If you supply two subscripts `(i, j)` representing row-column indices, the offset is

```
(j-1) * d1 + i
```

Given the expression `A(3,2)`, MATLAB calculates the offset into `A`'s storage column as `(2-1) * 3 + 3`, or `6`. Counting down six elements in the column accesses the value `5`.

Functions That Control Indexing Style

If you have row-column subscripts but want to use linear indexing instead, you can convert to the latter using the `sub2ind` function. In the 3-by-3 matrix `A` used in the previous section, `sub2ind` changes a standard row-column index of `(3, 2)` to a linear index of `6`:

```
A = [2 6 9; 4 2 8; 3 0 1];

linearindex = sub2ind(size(A), 3, 2)
linearindex =
     6
```

To get the row-column equivalent of a linear index, use the `ind2sub` function:

```
[row col] = ind2sub(size(A), 6)
row =
     3
col =
     2
```

Accessing Multiple Elements

For the 4-by-4 matrix `A` shown below, it is possible to compute the sum of the elements in the fourth column of `A` by typing

```
A = magic(4);
A(1,4) + A(2,4) + A(3,4) + A(4,4)
```

You can reduce the size of this expression using the colon operator. Subscript expressions involving colons refer to portions of a matrix. The expression

```
A(1:m, n)
```

refers to the elements in rows `1` through `m` of column `n` of matrix `A`. Using this notation, you can compute the sum of the fourth column of `A` more succinctly:

```
sum(A(1:4, 4))
```

Nonconsecutive Elements

To refer to nonconsecutive elements in a matrix, use the colon operator with a step value. The `m:3:n` in this expression means to make the assignment to every third element in the matrix. Note that this example uses linear indexing:

```
B = A;

B(1:3:16) = -10
B =
   -10     2     3   -10
     5    11   -10     8
     9   -10     6    12
   -10    14    15   -10
```

The end Keyword

MATLAB provides a keyword called `end` that designates the last element in the dimension in which it appears. This keyword can be useful in instances where your program doesn't know how many rows or columns there are in a matrix. You can replace the expression in the previous example with

```
B(1:3:end) = -10
```

Note The keyword `end` has two meanings in MATLAB. It can be used as explained above, or to terminate a certain block of code (e.g., `if` and `for` blocks).

Specifying All Elements of a Row or Column

The colon by itself refers to *all* the elements in a row or column of a matrix. Using the following syntax, you can compute the sum of all elements in the second column of a 4-by-4 magic square `A`:

```
sum(A(:, 2))
ans =
    34
```

By using the colon with linear indexing, you can refer to all elements in the entire matrix. This example displays all the elements of matrix A, returning them in a column-wise order:

```
A(:)
ans =
    16
     5
     9
     4
     .
     .
     .
    12
     1
```

Using a Matrix As an Index

You can repeatedly access an array element using the ones function. To create a new 2-by-6 matrix out of the ninth element of a 4-by-4 magic square A,

```
B = A(9 * ones(2, 6))
B =
     3     3     3     3     3     3
     3     3     3     3     3     3
```

Logical Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See “Using Logicals in Array Indexing” on page 2-22 for more information on this subject.

Getting Information About a Matrix

This section explains how to get the following information about an existing matrix:

Dimensions of the Matrix

These functions return information about the shape and size of a matrix.

Function	Description
`length`	Return the length of the longest dimension. (The length of a matrix or array with any zero dimension is zero.)
`ndims`	Return the number of dimensions.
`numel`	Return the number of elements.
`size`	Return the length of each dimension.

The following examples show some simple ways to use these functions. Both use the 3-by-5 matrix `A` shown here:

```
A = rand(5) * 10;
A(4:5, :) = []
A =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
```

Example Using numel

Using the `numel` function, find the average of all values in matrix `A`:

```
sum(A(:))/numel(A)
ans =
    5.8909
```

Example Using ndims, numel, and size

Using `ndims` and `size`, go through the matrix and find those values that are between 5 and 7, inclusive:

```
if ndims(A) ~= 2
   return
end

[rows cols] = size(A);
for m = 1:rows
   for n = 1:cols
      x = A(m, n);
      if x >= 5 && x <= 7
         disp(sprintf('A(%d, %d) = %5.2f', m, n, A(m,n)))
      end
   end
end
```

The code returns the following:

```
A(1, 3) =  6.15
A(3, 1) =  6.07
```

Data Types Used in the Matrix

These functions test elements of a matrix for a specific data type.

Function	Description
`isa`	Detect if input is of a given data type.
`iscell`	Determine if input is a cell array.
`iscellstr`	Determine if input is a cell array of strings.
`ischar`	Determine if input is a character array.
`isfloat`	Determine if input is a floating-point array.
`isinteger`	Determine if input is an integer array.
`islogical`	Determine if input is a logical array.

Function	Description
isnumeric	Determine if input is a numeric array.
isreal	Determine if input is an array of real numbers.
isstruct	Determine if input is a MATLAB structure array.

Example Using isnumeric and isreal

Pick out the real numeric elements from this vector:

```
A = [5+7i 8/7 4.23 39j pi 9-2i];

for m = 1:numel(A)
   if isnumeric(A(m)) && isreal(A(m))
      disp(A(m))
   end
end
```

The values returned are

```
    1.1429
    4.2300
    3.1416
```

Data Structures Used in the Matrix

These functions test elements of a matrix for a specific data structure.

Function	Description
isempty	Determine if input has any dimension with size zero.
isscalar	Determine if input is a 1-by-1 matrix.
issparse	Determine if input is a sparse matrix.
isvector	Determine if input is a 1-by-n or n-by-1 matrix.

Resizing and Reshaping Matrices

You can easily enlarge or shrink the size of a matrix, modify its shape, or rotate it about various axes. This section covers

Expanding the Size of a Matrix

Attempting to access an element outside of the matrix generates an error:

```
A = magic(4);
B = A(4, 7)
Index exceeds matrix dimensions
```

However, if you store a value in an element outside of the matrix, the size of the matrix increases to accommodate the new element.

```
A(4, 7) = 17
A =
    16     2     3    13     0     0     0
     5    11    10     8     0     0     0
     9     7     6    12     0     0     0
     4    14    15     1     0     0    17
```

Similarly, you can expand a matrix by assigning to a series of matrix elements. This example expands a 4-by-4 matrix to new dimensions, 5-by-7:

```
A(2:5, 5:7) = 5
A =
    16     2     3    13     0     0     0
     5    11    10     8     5     5     5
     9     7     6    12     5     5     5
     4    14    15     1     5     5     5
     0     0     0     0     5     5     5
```

Diminishing the Size of a Matrix

You can delete rows and columns from a matrix by assigning the empty array `[]` to those rows or columns. Start with

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

Then, delete the second column of `A` using

```
A(:, 2) = []
```

This changes matrix `A` to

```
A =
    16     3    13
     5    10     8
     9     6    12
     4    15     1
```

If you delete a single element from a matrix, the result isn't a matrix anymore. So expressions like

```
A(1,2) = []
```

result in an error. However, you can use linear indexing to delete a single element, or a sequence of elements. This reshapes the remaining elements into a row vector:

```
A(2:2:10) = []
```

results in

```
A =
    16     9     3     6    13    12     1
```

Reshaping a Matrix

The following functions change the shape of a matrix.

Function	Description
`reshape`	Modify the shape of a matrix.
`rot90`	Rotate the matrix by 90 degrees.
`fliplr`	Flip the matrix about a vertical axis.
`flipud`	Flip the matrix about a horizontal axis.
`flipdim`	Flip the matrix along the specified direction.
`transpose`	Flip a matrix about its main diagonal, turning row vectors into column vectors and vice versa.
`ctranspose`	Transpose a matrix and replace each element with its complex conjugate.

Examples

Here are a few examples to illustrate some of the ways you can reshape matrices.

Reshaping a Matrix. Reshape 3-by-4 matrix `A` to have dimensions 2-by-6:

```
A = [1 4 7 10; 2 5 8 11; 3 6 9 12]
A =
    1    4    7    10
    2    5    8    11
    3    6    9    12

B = reshape(A, 2, 6)
B =
    1    3    5    7    9   11
    2    4    6    8   10   12
```

Transposing a Matrix. Transpose `A` so that the row elements become columns. You can use either the `transpose` function or the transpose operator (`.'`) to do this:

```
B = A.'
B =
     1     2     3
     4     5     6
     7     8     9
    10    11    12
```

There is a separate function called `ctranspose` that performs a complex conjugate transpose of a matrix. The equivalent operator for `ctranpose` on a matrix `A` is `A'`:

```
A = [1+9i 2-8i 3+7i; 4-6i 5+5i 6-4i]
A =
   1.0000 + 9.0000i   2.0000 -8.0000i   3.0000 + 7.0000i
   4.0000 -6.0000i   5.0000 + 5.0000i   6.0000 -4.0000i

B = A'
B =
   1.0000 -9.0000i   4.0000 + 6.0000i
   2.0000 + 8.0000i   5.0000 -5.0000i
   3.0000 -7.0000i   6.0000 + 4.0000i
```

Rotating a Matrix. Rotate the matrix by 90 degrees:

```
B = rot90(A)
B =
    10    11    12
     7     8     9
     4     5     6
     1     2     3
```

Flipping a Matrix. Flip `A` in a left-to-right direction:

```
B = fliplr(A)
B =
    10     7     4     1
    11     8     5     2
    12     9     6     3
```

Shifting and Sorting Matrices

You can sort matrices, multidimensional arrays, and cell arrays of strings along any dimension and in ascending or descending order of the elements. The sort functions also return an optional array of indices showing the order in which elements were rearranged during the sorting operation.

This section covers

Shift and Sort Functions

Use these functions to shift or sort the elements of a matrix.

Function	Description
`circshift`	Circularly shift matrix contents.
`sort`	Sort array elements in ascending or descending order.
`sortrows`	Sort rows in ascending order.
`issorted`	Determine if matrix elements are in sorted order.

Shifting the Location of Matrix Elements

The `circshift` function shifts the elements of a matrix in a circular manner along one or more dimensions. Rows or columns that are shifted out of the matrix circulate back into the opposite end. For example, shifting a 4-by-7 matrix one place to the left moves the elements in columns 2 through 7 to columns 1 through 6, and moves column 1 to column 7.

Create a 5-by-8 matrix named `A` and shift it to the right along the second (horizontal) dimension by three places. (You would use `[0, -3]` to shift to the left by three places):

```
A = [1:8; 11:18; 21:28; 31:38; 41:48]
A =
     1     2     3     4     5     6     7     8
    11    12    13    14    15    16    17    18
    21    22    23    24    25    26    27    28
    31    32    33    34    35    36    37    38
    41    42    43    44    45    46    47    48

B = circshift(A, [0, 3])
B =
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
```

Now take `A` and shift it along both dimensions: three columns to the right and two rows up:

```
A = [1:8; 11:18; 21:28; 31:38; 41:48];

B = circshift(A, [-2, 3])
B =
    26    27    28    21    22    23    24    25
    36    37    38    31    32    33    34    35
    46    47    48    41    42    43    44    45
     6     7     8     1     2     3     4     5
    16    17    18    11    12    13    14    15
```

Since `circshift` circulates shifted rows and columns around to the other end of a matrix, shifting by the exact size of `A` returns all rows and columns to their original location:

```
B = circshift(A, size(A));

all(B(:) == A(:))          % Do all elements of B equal A?
ans =
     1                     % Yes
```

Sorting the Data in Each Column

The `sort` function sorts matrix elements along a specified dimension. The syntax for the function is

```
sort(matrix, dimension)
```

To sort the columns of a matrix, specify 1 as the `dimension` argument. To sort along rows, specify `dimension` as 2.

This example first constructs a 3-by-5 matrix:

```
A = rand(3,5) * 10
A =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
```

Sort each column of `A` in ascending order:

```
c = sort(A, 1)
c =
    2.3114    0.1850    6.1543    4.0571    0.5789
    6.0684    4.5647    7.9194    9.1690    3.5287
    9.5013    7.6210    9.2181    9.3547    8.1317

issorted(c(:, 1))
ans =
     1
```

Sorting the Data in Each Row

Sort each row of `A` in descending order. Note that `issorted` tests for an ascending sequence. You can flip the vector to test for a sorted descending sequence:

```
r = sort(A, 2, 'descend')
r =
    9.5013    7.6210    6.1543    4.0571    0.5789
    9.3547    7.9194    4.5647    3.5287    2.3114
    9.2181    9.1690    8.1317    6.0684    0.1850
```

```
issorted(fliplr(r(1, :)))
ans =
     1
```

When you specify a second output, `sort` returns the indices of the original matrix `A` positioned in the order they appear in the output matrix. In the following example, the second row of `index` contains the sequence `4 3 2 5 1`, which means that the sorted elements in output matrix `r` were taken from `A(2,4)`, `A(2,3)`, `A(2,2)`, `A(2,5)`, and `A(2,1)`:

```
[r index] = sort(A, 2, 'descend');
index
index =
     1     2     3     4     5
     4     3     2     5     1
     3     4     5     1     2
```

Sorting Row Vectors

The `sortrows` function keeps the elements of each row in their original order but sorts the entire row vectors according to the order of the elements in the first column:

```
rowsort = sortrows(A)
rowsort =
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317
    9.5013    7.6210    6.1543    4.0571    0.5789
```

To run the sort based on a different column, include a second input argument that indicates which column to use. This example sorts the row vectors so that the elements in the third column are in ascending order:

```
rowsort = sortrows(A, 3)
rowsort =
    9.5013    7.6210    6.1543    4.0571    0.5789
    2.3114    4.5647    7.9194    9.3547    3.5287
    6.0684    0.1850    9.2181    9.1690    8.1317

issorted(rowsort(:, 3))
ans =
     1
```

Operating on Diagonal Matrices

There are several MATLAB functions that work specifically on diagonal matrices.

Function	Description
`blkdiag`	Construct a block diagonal matrix from input arguments.
`diag`	Return a diagonal matrix or the diagonals of a matrix.
`trace`	Compute the sum of the elements on the main diagonal.
`tril`	Return the lower triangular part of a matrix.
`triu`	Return the upper triangular part of a matrix.

Constructing a Matrix from a Diagonal Vector

The `diag` function has two operations that it can perform. You can use it to generate a diagonal matrix:

```
A = diag([12:4:32])
A =
    12     0     0     0     0     0
     0    16     0     0     0     0
     0     0    20     0     0     0
     0     0     0    24     0     0
     0     0     0     0    28     0
     0     0     0     0     0    32
```

You can also use the `diag` function to scan an existing matrix and return the values found along one of the diagonals:

```
A = magic(5)
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9
```

```
diag(A, 2)       % Return contents of second diagonal of A
ans =
     1
    14
    22
```

Returning a Triangular Portion of a Matrix

The `tril` and `triu` functions return a triangular portion of a matrix, the former returning the piece from the lower left and the latter from the upper right. By default, the main diagonal of the matrix divides these two segments. You can use an alternate diagonal by specifying an offset from the main diagonal as a second input argument:

```
A = magic(6);

B = tril(A, -1)
B =
     0     0     0     0     0     0
     3     0     0     0     0     0
    31     9     0     0     0     0
     8    28    33     0     0     0
    30     5    34    12     0     0
     4    36    29    13    18     0
```

Concatenating Matrices Diagonally

You can diagonally concatenate matrices to form a composite matrix using the `blkdiag` function. See “Creating a Block Diagonal Matrix” on page 1-9 for more information on how this works.

Empty Matrices, Scalars, and Vectors

Although matrices are two dimensional, they do not always appear to have a rectangular shape. A 1-by-8 matrix, for example, has two dimensions yet is linear. These matrices are described in the following sections:

- "The Empty Matrix" on page 1-32

  An *empty matrix* has one of more dimensions that are equal to zero. A two-dimensional matrix with both dimensions equal to zero appears in MATLAB as `[]`. The expression `A = []` assigns a 0-by-0 empty matrix to `A`.

- "Scalars" on page 1-33

  A *scalar* is 1-by-1 and appears in MATLAB as a single real or complex number (e.g., `7`, `583.62`, `-3.51`, `5.46097e-14`, `83+4i`).

- "Vectors" on page 1-34

  A *vector* is 1-by-n or n-by-1, and appears in MATLAB as a row or column of real or complex numbers:

Column Vector	Row Vector

`53.2`
`87.39`
`4-12i`
`43.9`
	`53.2 87.39 4-12i 43.9`

The Empty Matrix

A matrix having at least one dimension equal to zero is called an empty matrix. The simplest empty matrix is 0-by-0 in size. Examples of more complex matrices are those of dimension 0-by-5 or 10-by-0.

To create a 0-by-0 matrix, use the square bracket operators with no value specified:

```
A = [];

whos A
  Name      Size         Bytes  Class

  A         0x0              0  double array
```

You can create empty matrices (and arrays) of other sizes using the `zeros`, `ones`, `rand`, or `eye` functions. To create a 0-by-5 matrix, for example, use

```
A = zeros(0,5)
```

Operating on an Empty Matrix

The basic model for empty matrices is that any operation that is defined for `m`-by-`n` matrices, and that produces a result whose dimension is some function of `m` and `n`, should still be allowed when `m` or `n` is zero. The size of the result of this operation is consistent with the size of the result generated when working with nonempty values, but instead is evaluated at zero.

For example, horizontal concatenation

```
C = [A B]
```

requires that `A` and `B` have the same number of rows. So if `A` is `m`-by-`n` and `B` is `m`-by-`p`, then `C` is `m-by-(n+p)`. This is still true if `m` or `n` or `p` is zero.

As with all matrices in MATLAB, you must follow the rules concerning compatible dimensions. In the following example, an attempt to add a 1-by-3 matrix to a 0-by-3 empty matrix results in an error:

```
[1 2 3] + ones(0,3)
```

```
??? Error using ==> +
Matrix dimensions must agree.
```

Scalars

Any individual real or complex number is represented in MATLAB as a 1-by-1 matrix called a scalar value:

```
A = 5;

ndims(A)        % Check number of dimensions in A
ans =
     2

size(A)         % Check value of row and column dimensions
ans =
     1     1
```

Use the `isscalar` function to tell if a variable holds a scalar value:

```
isscalar(A)
ans =
    1
```

Vectors

Matrices with one dimension equal to one and the other greater than one are called vectors. Here is an example of a numeric vector:

```
A = [5.73 2-4i 9/7 25e3 .046 sqrt(32) 8j];

size(A)         % Check value of row and column dimensions
ans =
     1     7
```

You can construct a vector out of other vectors, as long as the critical dimensions agree. All components of a row vector must be scalars or other row vectors. Similarly, all components of a column vector must be scalars or other column vectors:

```
A = [29 43 77 9 21];
B = [0 46 11];

C = [A 5 ones(1,3) B]
C =
   29   43   77    9   21    5    1    1    1    0   46   11
```

Concatenating an empty matrix to a vector has no effect on the resulting vector. The empty matrix is ignored in this case:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Use the `isvector` function to tell if a variable holds a vector:

```
isvector(A)
ans =
    1
```