| | |
|---|---|
| الكلية | العلوم |
| القسم | الرياضيات |
| المادة باللغة الانجليزية | Programming (MATLAB) |
| المادة باللغة العربية | البرمجة بلغة (ماتلاب) |
| المرحلة الدراسية | الثانية |
| اسم التدريسي | صفوت عبدالقادر حمد |
| عنوان المحاضرة باللغة الانجليزية | Dimensions of Matrix |
| عنوان المحاضرة باللغة العربية | ابعاد المصفوفات |
| رقم المحاضرة | السادسة |
| المصادر والمراجع | MATLAB<br>A Practical Introduction to Programming and Problem Solving |
| | MATLAB<br>The Language of Technical Computing |
| | MATLAB numerical computing |

University Of Anbar

College Of Science

Math Department

# MATRIX - LABORATORY (MATLAB)


Lecture 6

Dimensions of Matrix

By:

Safwat A. Hamad

For the 2nd stage Math Department

# 1. Dimensions of Matrix

The length and size functions in MATLAB are used to find dimensions of vectors and matrices. The **length** function returns the number of elements in a vector. The **size** function returns the number of rows and columns in a vector or matrix. For example, the following vector *vec* has four elements, so its length is 4. It is a row vector, so the size is (1*4).

```
>> vec = -2:1
   vec =
     -2 -1 0 1
>> length(vec)
ans =
     4
>> size(vec)
ans =
     1 4
```

To create the following matrix variable *mat*, iterators are used on the two rows and then the matrix is transposed so that it has three rows and two columns or, in other words, the size is (3*2).

```
>> mat = [1:3; 5:7]'
   mat =
        1 5
        2 6
        3 7
```

The **size** function returns the number of rows and then the number of columns, so to capture these values in separate variables we put a vector of variables (two) on the left of the assignment. The variable (*r)* stores the first value returned, which is the number of rows, and (*c*) stores the number of columns.

```
>> [r, c] = size(mat)
r =
```

```
     3
c =
     2
```

***Note that this example demonstrates very important and unique concepts in MATLAB: the ability to have a function return multiple values and the ability to have a vector of variables on the left side of an assignment in which to store the values.***

If called as just an expression, the size function will return both values in a vector:

```
>> size(mat)
ans =
     3   2
```

For a matrix, the length function will return either the number of rows or the number of columns, **whichever is largest** (in this case the number of rows, 3).

```
>> length(mat)
ans =
     3
```

## 2. Other Functions for dimension of Matrix

MATLAB also has a function **numel**, which returns the total number of elements in any array (vector or matrix):

```
>> vec = 9:-2:1
vec =
     9   7   5   3   1
>> numel(vec)
ans =
     5
>> mat = [3:2:7; 9 33 11]
mat =
     3   5    7
     9   33   11
```

```
>> numel(mat)
ans =
     6
```

For vectors, **numel** is equivalent to the length of the vector. For matrices, it is the product of the number of rows and columns.

It is important to **Note** that in programming applications, it is better to not assume that the dimensions of a vector or matrix are known. Instead, to be general, use either the **length** or **numel** function to determine the number of elements in a vector and use size (and store the result in two variables) for a matrix.

MATLAB also has a built-in expression **(end**) that can be used to refer to the last element in a vector; for example, *v (end)* is equivalent to *v (length(v))*. For matrices, it can refer to the last row or column. So, for example, using end for the row index would refer to the last row. In this case, the element referred to is in the first column of the last row:

```
>> mat = [1:3; 4:6]'
mat =
     1   4
     2   5
     3   6
>> mat(end,1)
ans =
     3
```

Using end for the column index would refer to a value in the last column (e.g., the last column of the second row):

```
>> mat(2,end)
ans =
     5
```

* The expression **end** can only be used as **an index**.

# 3. Changing Dimensions

In addition to the transpose operator, MATLAB has several built-in functions that change the dimensions or configuration of matrices (or in many cases vectors), including **reshape**, **fliplr**, **flipud**, **flip**, and **rot90**. The reshape function changes the dimensions of a matrix. The following matrix variable mat is (3*4) or, in other words, it has 12 elements (each in the range from 1 to 100).

```
>> mat = randi(100, 3, 4)
     14   61   2    94
     21   28   75   47
     20   20   45   42
```

These 12 values could instead be arranged as a (2*6 matrix), 6*2, 4*3, 1*12, or 12*1. The reshape function iterates through the matrix column wise. For example, when reshaping *mat* into a (2*6 matrix), the values from the first column in the original matrix (14, 21, and 20) are used first, then the values from the second column (61, 28, 20), and so forth.

```
>> reshape(mat,2,6)
ans =
     14   20   28   2   45   47
     21   61   20   75  94   42
```

Other example:

```
>> mat = randi(100, 3, 4)
mat =
    82    92    28    97
    91    64    55    16
    13    10    96    98
>> reshape (mat,2 ,6)
ans =
    82    13    64    28    96    16
    91    92    10    55    97    98
```

```
>> reshape (mat,4,3)
ans =
    82    64    96
    91    10    97
    13    28    16
    92    55    98
```

**Note** that in these examples mat is unchanged; instead, the results are stored in the default variable *ans* each time.

There are several functions that flip arrays. The **fliplr** function "flips" the matrix from left to right (in other words, the left-most column, the first column, becomes the last column and so forth), and the flipud function flips up to down.

```
>> mat
mat =
    14   61   2    94
    21   28   75   47
    20   20   45   42
>> fliplr(mat)
ans =
    94   2    61   14
    47   75   28   21
    42   45   20   20
>> mat
mat =
    14   61   2    94
    21   28   75   47
    20   20   45   42
>> flipud(mat)
ans =
    20   20   45   42
    21   28   75   47
    14   61   2    94
```

The **flip** function flips any array; it flips a vector (left to right if it is a row vector or up to down if it is a column vector) or a matrix (up to down by default).

The **rot90** function rotates the matrix counterclockwise 90degrees, so, for example, the value in the top right corner becomes instead the top left corner and the last column becomes the first row.

```
>> mat
mat =
     14   61   2    94
     21   28   75   47
     20   20   45   42
>> rot90(mat)
ans =
     94   47   42
     2    75   45
     61   28   20
     14   21   20
```

The function **repmat** can be used to create a matrix; *repmat (mat,m,n)* creates a larger matrix that consists of an (m*n) matrix of copies of *mat*. For example, here is a (2*2) random matrix:

```
>> intmat = randi(100,2)
intmat =
     50   34
     96   59
```

Replicating this matrix six times as a (3*2) matrix would produce copies of intmat in this form:

```
>> repmat(intmat,3,2)
ans =
     50   34   50   34
     96   59   96   59
     50   34   50   34
```

```
    96   59   96   59
    50   34   50   34
    96   59   96   59
```

Other example:

```
>> mat =
    82    92    28    97
    91    64    55    16
    13    10    96    98
>> repmat (mat, 2,2)
ans =
    82    92    28    97    82    92    28    97
    91    64    55    16    91    64    55    16
    13    10    96    98    13    10    96    98
    82    92    28    97    82    92    28    97
    91    64    55    16    91    64    55    16
    13    10    96    98    13    10    96    98
```

The function repelem, on the other hand, replicates each element from a matrix in the dimensions specified; this function was introduced in **R2015a**.

```
>> repelem(intmat,3,2)
ans =
    50 50 34 34
    50 50 34 34
    50 50 34 34
    96 96 59 59
    96 96 59 59
    96 96 59 59
```

# 4. Empty Vectors

An empty vector (a vector that stores no values) can be created using empty square brackets:

```
>> evec = []
```

```
evec =
     []
>> length(evec)
ans =
     0
```

**Note** that there is a difference between having an empty vector variable and not having the variable at all. Values can then be added to an empty vector by concatenating, or adding, values to the existing vector. The following statement takes what is currently in evec, which is nothing, and adds a 4 to it.

```
>> evec = [evec 4]
evec =
          4
```

The following statement takes what is currently in evec, which is 4, and adds an 11 to it.

```
>> evec = [evec 11]
evec =
     4 11
```

Other example:

```
>> vec = []
vec =
     []
>> vec1 = [vec 5]
vec1 =
     5
```

This can be continued as many times as desired, to build a vector up from nothing. Sometimes this is necessary, although generally it is not a good idea if it can be avoided because it can be quite time-consuming.

Empty vectors can also be used to delete elements from vectors. For example, to remove the third element from a vector, the empty vector is assigned to it:

```
>> vec = 4:8
vec =
     4   5   6   7   8
>> vec(3) = []
vec =
     4   5   7   8
```

The elements in this vector are now numbered 1 through 4. **Note** that the variable *vec* has actually changed.

Subsets of a vector could also be removed. For example:

```
>> vec = 3:10
vec =
     3   4   5   6   7   8   9   10
>> vec(2:4) = []
vec =
     3 7 8 9 10
```

Individual elements cannot be removed from matrices, as matrices always have to have the same number of elements in every row.

```
>> mat = [7 9 8; 4 6 5]
mat =
     7   9   8
     4   6   5
>> mat(1,2) = [];
Subscripted assignment dimension mismatch.
```

However, entire rows or columns could be removed from a matrix. For example, to remove the second column:

```
>> mat(:,2) = []
mat =
     7 8
     4 5
```

**Also**, if linear indexing is used with a matrix to delete an element, the matrix will be reshaped into a row vector.

```
>> mat = [7 9 8; 4 6 5]
mat =
     7   9   8
     4   6   5
>> mat(3) = []
mat =
7 4 6 8 5
```

***** (Again, using linear indexing is not a good idea.)

---

H.M. or lab exercise ?

1. How could you create a matrix of **zeros** with the same size as another matrix?
2. How could you create a matrix of **ones** with the same size as another matrix ?
3. Is there a rot180 function? And how to rotate the matrix as 180 degree, use the variable **mat** as matrix equal **randi** (50, 3, 4)?