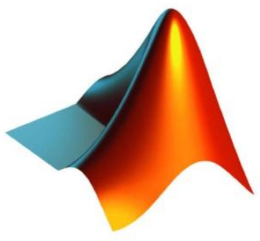
الكلية	العلوم
القسم	الرياضيات
المادة باللغة الانجليزية	Programming (MATLAB)
المادة باللغة العربية	البرمجة بلغة (ماتلاب)
المرحلة الدراسية	الثانية
اسم التدريسي	صفوت عبدالقادر حمد
عنوان المحاضرة باللغة الانجليزية	Function Arguments
عنوان المحاضرة باللغة العربية	دوال المصفوفات
رقم المحاضرة	السابعة
المصادر والمراجع	MATLAB A Practical Introduction to Programming and Problem Solving
	MATLAB The Language of Technical Computing
	MATLAB numerical computing

University Of Anbar

College Of Science

Math Department

MATRIX - LABORATORY (MATLAB)



Lecture 7

Function Arguments

By:

Safwat A. Hamad

For the 2nd stage Math Department

1. Function Arguments

In MATLAB, an entire vector or matrix can be passed as an argument to a function; the function will be evaluated on every element. This means that the result will be the same size as the input argument.

For example, let us find the absolute value of every element of a vector *vec*. The **abs** function will automatically return the absolute value of each individual element and the result will be a vector with the same length as the input vector.

```
>> vec = -2:1
vec =
     -2  -1  0  1
>> absvec = abs(vec)
absvec =
     2  1  0  1
```

For a matrix, the resulting matrix will have the same size as the input argument matrix. For example, the **sign** function will find the sign of each element in a matrix:

```
>> mat = [0 4 -3; -1 0 2]
mat =
     0   4  -3
     -1  0   2
>> sign(mat)
ans =
     0   1  -1
     -1  0   1
```

Functions such as **abs** and **sign** can have either scalars or arrays (vectors or matrices) passed to them. There are a number of functions that are written specifically to operate on vectors or on columns of matrices; these include the functions **min**, **max**, **sum**, and **prod**. These functions will be demonstrated first with vectors, and then with matrices.

For example, assume that we have the following vector variables:

```
>> vec1 = 1:5;
>> vec2 = [3 5 8 2];
```

The function **min** will return the minimum value from a vector, and the function **max** will return the maximum value.

```
>> min(vec1)
ans =
        1
>> max(vec2)
ans =
        8
```

The function **sum** will sum all of the elements in a vector. For example, for *vec1* it will return 1+2+3+4+5 or 15:

```
>> sum(vec1)
ans =
        15
```

The function **prod** will return the product of all of the elements in a vector; for example, for *vec2* it will return 3*5*8*2 or 240:

```
>> prod(vec2)
ans =
        240
```

There are also functions that return cumulative results; the functions **cumsum** and **cumprod** return the cumulative sum or cumulative product, respectively. A cumulative, or running sum, stores the sum so far at each step as it adds the elements from the vector. For example, for *vec1*, it would store the first element, 1, then 3 (1+2), then 6 (1+2+3), then 10 (1+2+3+4), then, finally, 15 (1+2+3+4+5). The result is a vector that has as many elements as the input argument vector that is passed to it:

```
>> cumsum(vec1)
ans =
     1   3   6   10   15
>> cumsum(vec2)
```

```
ans =
     3   8   16   18
```

The **cumprod** function stores the cumulative products as it multiplies the elements in the vector together; again, the resulting vector will have the same length as the input vector:

```
>> cumprod(vec1)
ans =
     1   2   6   24   120
```

Similarly, there are cummin and cummax functions, which were introduced in R2014b. Also, in R2014b, a 'reverse' option was introduced for all of the cumulative functions. For example,

```
>> cumsum(vec1,'reverse')
ans =
     15   14   12   9   5
```

For matrices, all of these functions operate on every individual column. If a matrix has dimensions (r*c), the result for the **min**, **max**, **sum**, and **prod** functions will be a (1 * c) row vector, as they return the minimum, maximum, sum, or product, respectively, for every column. For example, assume the following matrix:

```
>> mat = randi([1 20], 3, 5)
mat =
     3    16    1    14    8
     9    20    17   16   14
     19   14    19   15   4
```

The following are the results for the **max** and **sum** functions:

```
>> max(mat)
ans =
     19   20   19   16   14
>> sum(mat)
ans =
     31   50   37   45   26
```

To find a function for every row, instead of every column, one method would be to transpose the matrix.

```
>> max(mat')
ans =
     16   20   19
>> sum(mat')
ans =
     42   76   71
```

Exercise 1: How would we determine the overall maximum in the matrix ?

For the **cumsum** and **cumprod** functions, again they return the cumulative sum or product of every column. The resulting matrix will have the same dimensions as the input matrix:

```
>> mat
mat =
     3   16   1   14   8
     9   20   17 16   14
     19 14   19 15   4
>> cumsum(mat)
ans =
     3    16   1    14   8
     12   36   18   30   22
     31   50   37   45   26
```

Note that the first row in the resulting matrix is the same as the first row in the input matrix. After that, the values in the rows accumulate. Similarly, the **cumin** and **cummax** functions find the cumulative minima and maxima for every column.

```
>> cummin(mat)
ans =
     3   16   1   14   8
     3   16   1   14   8
```

```
        3   14   1   14   4
>> cummax(mat,'reverse')
ans =
        19   20   19   16   14
        19   20   19   16   14
        19   14   19   15   4
```

Another useful function that can be used with vectors and matrices is **diff**. The function **diff** returns the differences between consecutive elements in a vector. For example,

```
>> diff([4 7 15 32])
ans =
        3   8   17
>> diff([4 7 2 32])
ans =
        3 -5 30
```

For a vector *v* with a length of n, the length of diff (v) will be (n – 1). For a matrix, the **diff** function will operate on each column.

```
>> mat = randi(20, 2,3)
mat =
        17   3   13
        19   19   2
>> diff(mat)
ans =
        2 16 -11
```

2. SCALAR AND ARRAY OPERATIONS ON VECTORS AND MATRICES

Numerical operations can be done on entire vectors or matrices. For example, let’s say that we want to multiply every element of a vector *v* by 3.

In MATLAB, we can simply multiply *v* by 3 and store the result back in *v* in an assignment statement:

```
>> v = [3 7 2 1];
>> v = v*3
v =
     9 21 6 3
```

As another example, we can divide every element by 2:

```
>> v= [3 7 2 1];
>> v/2
ans =
1.5000 3.5000 1.0000 0.5000
```

To multiply every element in a matrix by 2:

```
>> mat = [4:6; 3:-1:1]
mat =
     4   5   6
     3   2   1
>> mat * 2
ans =
     8   10  12
     6   4    2
```

This operation is referred to as scalar multiplication. We are multiplying every element in a vector or matrix by a scalar (or dividing every element in a vector or a matrix by a scalar).

Exercise 2: There is no tens function to create a matrix of all tens, so how could we accomplish that?

Array operations are operations that are performed on vectors or matrices term by term, or element by element. This means that the two arrays (vectors or matrices) must be of the same size to begin with. The following examples demonstrate the array addition and subtraction operators.

```
>> v1 = 2:5
v1 =
    2   3   4   5
>> v2 = [33 11 5 1]
v2 =
    33   11   5   1
>> v1 + v2
ans =
    35   14   9   6
>> mata = [5:8; 9:-2:3]
mata =
    5   6   7   8
    9   7   5   3
>> matb = reshape(1:8,2,4)
matb =
    1   3   5   7
    2   4   6   8
>> mata - matb
ans =
    4   3   2   1
    7   3  -1  -5
```

However, for any operation that is based on multiplication (which means multiplication, division, and exponentiation), a dot must be placed in front of the operator for array operations. For example, for the exponentiation operator, (.^) must be used when working with vectors and matrices, rather than just the (^) operator. Squaring a vector, for example, means multiplying each element by itself so the (.^) operator must be used.

```
>> v = [3 7 2 1];
>> v ^ 2
Error using ^
```

```
Incorrect dimensions for raising a matrix to a power. Check that the
matrix is square and the power is a scalar. To perform elementwise matrix
powers, use '.^'.
>> v .^2
ans =
     9   49   4   1
```

Similarly, the operator (**.***) must be used for array multiplication and (**./**) or (**.\**) for array division. The following examples demonstrate array multiplication and array division.

```
>> v1 = 2:5
v1 =
     2   3   4   5
>> v2 = [33 11 5 1]
v2 =
     33   11   5   1
>> v1 .* v2
ans =
     66   33   20   5
>> mata = [5:8; 9:-2:3]
mata =
     5   6   7   8
     9   7   5   3
>> matb = reshape(1:8, 2,4)
matb =
     1   3   5   7
     2   4   6   8
>> mata ./ matb
ans =
     5.0000    2.0000    1.4000    1.1429
     4.5000    1.7500    0.8333    0.3750
```

The operators (**.**^), (**.***), (**./**, and **.**\) are called array operators and are used when multiplying or dividing vectors or matrices of the same size term by term. **Note** that matrix multiplication is a very different operation and will be covered in next lecture.

<u>Exercise 3: Create a vector variable and subtract 3 from every element in it.</u>

<u>Exercise 4: Create a matrix variable and divide every element by 3.</u>

<u>Exercise 5: Create a matrix variable and square every element.</u>
