ENME202 Matlab

1-D ARRAYS

Like scalar numbers, an array of values can be assigned to a variable. Here we define a 4 element array:

```
x = [1 3 5 6]
```

```
x = 1×4
     1     3     5     6
```

All of the numbers in the square brackets are now associated with the variable "x", which now holds a list (array) of 4 numbers, instead of a single number.

Can use real or complex numbers, mathematical expressions, or existing variables to specify the elements of an array. Matlab will work out the numerical values for any expressions in the brackets, then associate these numbers with the given variable name:

```
a = -1;
y = [1+2j  sqrt(-2)  a  exp(a)]
```

```
y = 1×4 complex
   1.0000 + 2.0000i   0.0000 + 1.4142i  -1.0000 + 0.0000i   0.3679 + 0.0000i
```

Default "orientation" of an array is horizontal (row), but we can convert a horizontal (row) array to a vertical (column) array using the **transpose operator: '** (apostrophe)

```
x = [1 3 5 6]'
```

```
x = 4×1
     1
     3
     5
     6
```

Alternately, **semicolons** can be used to create individual rows when first defining the array. The following will yield a column array:

```
z = [1;3;5;6]
```

```
z = 4×1
     1
     3
     5
     6
```

Transpose is a "toggle" which turns rows into columns and vice-versa. The following statements are all equivalent:

```
z = [1 3 5 6]
```

```
z = 1×4
     1     3     5     6
```

```
z = [1;3;5;6]'
```

```
z = 1×4
     1     3     5     6
```

```
z = [1 3 5 6]''    % double transpose
```

```
z = 1×4
     1     3     5     6
```

IMPORTANT: for complex-valued arrays, the ' operator also yields the complex conjugate!

```
x = [1 2+3i]
```

```
x = 1×2 complex
   1.0000 + 0.0000i   2.0000 + 3.0000i
```

```
x'           % yields [1; 2-3i]
```

```
ans = 2×1 complex
   1.0000 + 0.0000i
   2.0000 - 3.0000i
```

To prevent Matlab from taking the conjugate, add a dot in front of the ' operator (**.'**)

```
x.'          % yields [1; 2+3i]
```

```
ans = 2×1 complex
   1.0000 + 0.0000i
   2.0000 + 3.0000i
```

ARRAY INDEXING

Access a specific value stored in an array by "indexing", i.e. indicating which position in the array you want to look at. The index is in parentheses after the variable name, and indicates the numbered position in the list for the value you want to work with.

In Matlab, we start counting array elements at 1 (unlike C++ and many other languages where counting starts at 0 instead):

```
z = [-10  5  0.54]
```

```
z = 1×3
  -10.0000    5.0000    0.5400
```

```

z(1)  % 1st element of the array --> -10
```

```
ans = -10
```

```
z(2)  % --> 5
```

```
ans = 5
```

```
z(3)  % --> 0.54
```

```
ans = 0.5400
```

```

% z(4)  % Error: Index exceeds the number of array elements
% z(0)  % Error: Array indices must be positive integers or logical values
% z(1.5)% Error: Array indices must be positive integers or logical values
```

We can change values stored in a specific position in an array:

```
z(2) = pi;
z
```

```
z = 1×3
  -10.0000    3.1416    0.5400
```

As an aside, we can use the ***disp()*** *function to define how the variable z (or any other value) should be displayed:*

```
disp("z = " + z)
```

```
     "z = -10"    "z = 3.1416"    "z = 0.54"
```

Here we seem to be "adding" a string and a complex value here, but Matlab is smart enough to recognize that the value held by z should be converted into a string before combining the result with our string for display.

Any individual element of an array can be used just like an ordinary (non-array) variable

```
asin(z(3))   % same as asin(0.54), since z(3) has value 0.54
```

```
ans = 0.5704
```

Length function tells how many values stored in an array

```
length(z)   % returns 3
```

```
ans = 3
```

"end" is a special index that means last element of the array

```
z(end)      % same as z(3) or z(length(z))
```

```
ans = 0.5400
```

Can do offsets from end to move "backwards" through the array

```
z(end-1)    % same as z(2)
```

```
ans = 3.1416
```

Variables can be used as indexes into an array, if they contain positive integers

```
k = 2;
z(k)
```

```
ans = 3.1416
```

```
k = k+1;
z(k)
```

```
ans = 0.5400
```

Matlab will "grow" arrays if you assign values to elements that do not yet exist:

```
x = [1;2;3;4]
```

```
x = 4×1
     1
     2
     3
     4
```

```
x(5) = pi;
x
```

```
x = 5×1
    1.0000
    2.0000
    3.0000
    4.0000
    3.1416
```

x has now "grown" to be a 5 element column array.

```
x(8) = exp(1)
```

```
x = 8×1
    1.0000
    2.0000
    3.0000
    4.0000
    3.1416
         0
         0
    2.7183
```

Here Matlab has expanded x to be 8 elements, and set the 8th element to exp(1) as specified. The values of the 6th and 7th elements have not been specified, and default to 0.

An empty array (null array):

```
x = []
```

```
x =

     []
```

```
length(x)     % returns 0
```

```
ans = 0
```

Colon notation can be used to automatically create arrays whose values are generated according to a specified linear sequence:

```
y = 2:5       % yields y = [2 3 4 5]
```

```
y = 1×4
     2     3     4     5
```

The default increment is 1, but this can be changed by specifying a different value between the limits. For example, this generates an array from -3 to 1 in steps of 0.5:

```
z = -3:0.5:1
```

```
z = 1×9
   -3.0000   -2.5000   -2.0000   -1.5000   -1.0000   -0.5000         0    0.5000    1.0000
```

All elementary functions operate on arrays by applying the function to every element of the array seperately, and retuning a new array containing each result in successive array elements:

```
q = 1:10
```

```
q = 1×10
     1     2     3     4     5     6     7     8     9    10
```

```
exp(-q)
```

```
ans = 1×10
    0.3679    0.1353    0.0498    0.0183    0.0067    0.0025    0.0009    0.0003    0.0001
```

```
sin(q)
```

```
ans = 1×10
    0.8415    0.9093    0.1411   -0.7568   -0.9589   -0.2794    0.6570    0.9894    0.4121
```

```
sqrt(q)
```

```
ans = 1×10
    1.0000    1.4142    1.7321    2.0000    2.2361    2.4495    2.6458    2.8284    3.0000
```

Scalar arithmetic works the same way (element-by-element):

```
2*q       % multiply every element in q by 2
```

```
ans = 1×10
     2     4     6     8    10    12    14    16    18    20
```

```
q/2       % divide every element by 2
```

```
ans = 1×10
```

```
    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000    4.5000
```

```
q+5        % add 5 to every element of q
```

```
ans = 1×10
     6     7     8     9    10    11    12    13    14    15
```

```
q-3        % subtract 3 from every element of q
```

```
ans = 1×10
    -2    -1     0     1     2     3     4     5     6     7
```

More complex arithmatic operations are also possible:

```
2*q + q/5 - 8*sin(q) + 1
```

```
ans = 1×10
   -3.5318   -1.8744    6.4710   15.8544   19.6714   16.4353   11.1441   10.6851   17.5031
```

Multi-Array Operations

```
x = 1:4;                % row vector
y = [-2; -3; 0; 1];     % column vector
```

You can add every element of one array to the corresponding elements of another array, BUT the two arrays must have the **same dimensions** (both orientation and length). The result will be an array of the sums.

```
x + y'      % x and y' are both row vectors -- ok!
```

```
ans = 1×4
    -1    -1     3     5
```

```

x' + y      % x' and y are both column vectors -- ok!
```

```
ans = 4×1
    -1
    -1
     3
     5
```

```

x + y       % This will create a 4x4 matrix! Try it and see what happens
```

```
ans = 4×4
    -1     0     1     2
    -2    -1     0     1
     1     2     3     4
     2     3     4     5
```

Must be careful if we want to multiply or divide every element of one array by those in another array. You CANNOT use ordinary * for this, even if lengths and orientations match:

```
% x * y'      % Error: Inner matrix dimensions must agree.
```

Matlab thinks we want to do matrix multiplication (not element-by-element multiplication), which requires that the inner dimensions of the arrays be the same (we will discuss this topic in linear algebra review soon).

For element-by-element multiplication, we must use the special dot-star (**.***) operator. The dot (.) tells Matlab that we want the given operation (*) to be performed individually for each index across both arrays:

```
x .* y'     % ans = [1*-2 2*-3 3*0 4*1] = [-2 -6 0 4]
```

```
ans = 1×4
    -2    -6     0     4
```

Similarly for division

```
x ./ y'     % ans = [1/-2 2/-3 3/0 4/1] = [-0.5 -0.6667 Inf 4]
```

```
ans = 1×4
   -0.5000   -0.6667       Inf    4.0000
```

```

y ./ x'       % ans = [-2/1 -3/2 0/3 1/4] = [-2 -1.5 0 0.25]
```

```
ans = 4×1
   -2.0000
   -1.5000
         0
    0.2500
```

Exponentiation is just iterated multiplication. Use the dot-caret (**.^**) operator to raise every number in an array to a given power:

```
x.^2        % ans = [1 4 9 16]
```

```
ans = 1×4
     1     4     9    16
```

Example – Using arrays to evaluate a function over a range of values:

Generate the values of a function f(x) over a range of x values, given: f(x) = 2x^3 + 8x^2 + 12x + 8

(A) The wrong way

Let's start by doing the work "manually". First, create the range of x values:

```
x = -3:.01:1;
```

How many values were created?

```
length(x)
```

```
ans = 401
```

Find f(x) for the first value in the array x, and store this in the first position of an array:

```
f(1) = 2*x(1)^3 + 8*x(1)^2 + 12*x(1) + 8      % note: x(1) = -3
```

```
f = -10
```

Note that "f(1)" is *not* f(x) when x = 1. Instead, it is the value of f(x(1)), where x(1) = -3.

Repeat for 2nd value of x:

```
f(2) = 2*x(2)^3 + 8*x(2)^2 + 12*x(2) + 8      % note: x(2) = -2.99
```

```
f = 1×2
  -10.0000   -9.8210
```

Then the 3rd value, then the 4th...

This would be very tedious to repeat for all of the remaining 399 values of x!

<u>(B) The right way</u>

Now let's do it the proper way, using Matlab's array capabilities to generate all 401 values for f(x) in a single step:

```
f = 2*x.^3 + 8*x.^2 + 12*x + 8;

length(f)       % same as length(x)
```

```
ans = 401
```

Results agree with what we calculated manually for the first two x values:

```
f(1)   % ans = 10
```

```
ans = -10
```

```
f(2)   % ans = -9.8210
```

```
ans = -9.8210
```

Other useful array functions

The following functions return a single scalar value for a given array.

```
sum(x)    % Add all elements in array together
```

```
ans = -401
```

```
prod(x)   % Multiply all elements in array together:
```

```
ans = 0
```

```
mean(x)   % Average (mean) of all array elements, same as sum(x)/length(x)
```

```
ans = -1
```

```
std(x)    % Standard deviation of array elements
```

```
ans = 1.1590
```

Min & Max

```
min(x)    % Find the minimum of all elements in an array
```

```
ans = -3
```

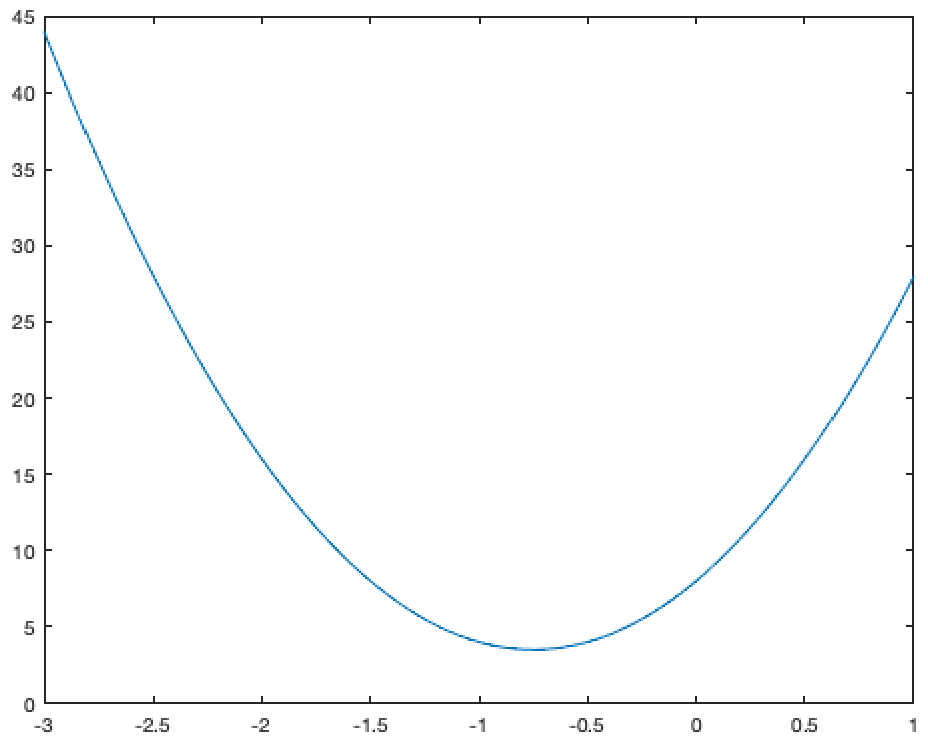
```
max(x)    % Find the maximum of all elements in an array
```

```
ans = 1
```

Both min() and max() can also return the index at which the given value occurs. To extract the index, assign the result of the function to a 2-element matrix: the first element will be the value, and the second element will be the index.

A bit more advanced: how to get both the minimal value of a function, AND the value of the independent variable at that minimum? Let's first make a plot to visually see where the minimum occurs:

```
x = -3:.01:1;
f = 8*x.^2 + 12*x + 8;
plot(x,f)
```



Use min to get both the minimal value AND the position in the array where this minimum occurs:

```
[fmin,imin] = min(f)    % yields fmin = 3.5, imin = 226
```

```
fmin = 3.5000
imin = 226
```

So the minimal value of the f array is 3.5, and this happens at the 226th index in the array. What is the value of x at which f(x) = 3.5? Simply look at the 226th element of the x array to find the corresponding value:

```
x(imin)                    % ans = -0.7500
```

```
ans = -0.7500
```

So the minimum occurs at x = -0.75.

Searching Arrays

A common challenge is to find the location of given values within an array. We can use the Matlab find() function to do this. For a 1-D array, find() will return *all* of the indices where a given *Boolean statement* is true:

```
x = [3 8 8 6 3];

find (x == 3)
```

```
ans = 1×2
     1     5
```

```
find (x < 6)
```

```
ans = 1×2
     1     5
```

```
find (x >= 6)
```

```
ans = 1×3
     2     3     4
```

```
find (x < 8 & x > 3)   % "&" means "logical and" -- more about this soon!
```

```
ans = 4
```

Array Slicing

We can "slice" arrays using an "array of indices":

```
y = [2 -9 3 0 8];
y_slice = y([1 2])        % ans = [2 -9]
```

```
y_slice = 1×2
     2    -9
```

Pull out elements 4,2,5 (in that order):

```
y([4 2 5])      % ans = [0 -9 8]
```

```
ans = 1×3
     0    -9     8
```

Pulling out contiguous slices using the colon notation:

```
y(1:3)
```

```
ans = 1×3
     2    -9     3
```

```
y(2:end)
```

```
ans = 1×4
    -9     3     0     8
```

```
y(end:-1:1)   % reverse the array!
```

```
ans = 1×5
     8     0     3    -9     2
```

Changing multiple array elements:

```
y([1 3 7]) = [pi; 100; -10]
```

```
y = 1×7
    3.1416   -9.0000  100.0000         0    8.0000         0  -10.0000
```