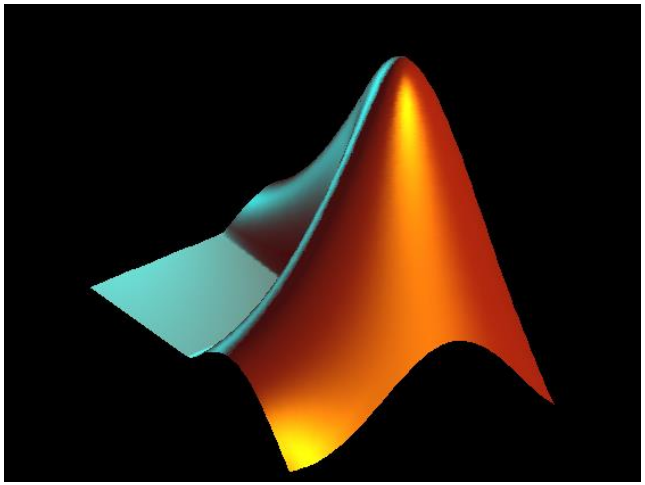
Function m-files



Topic 7

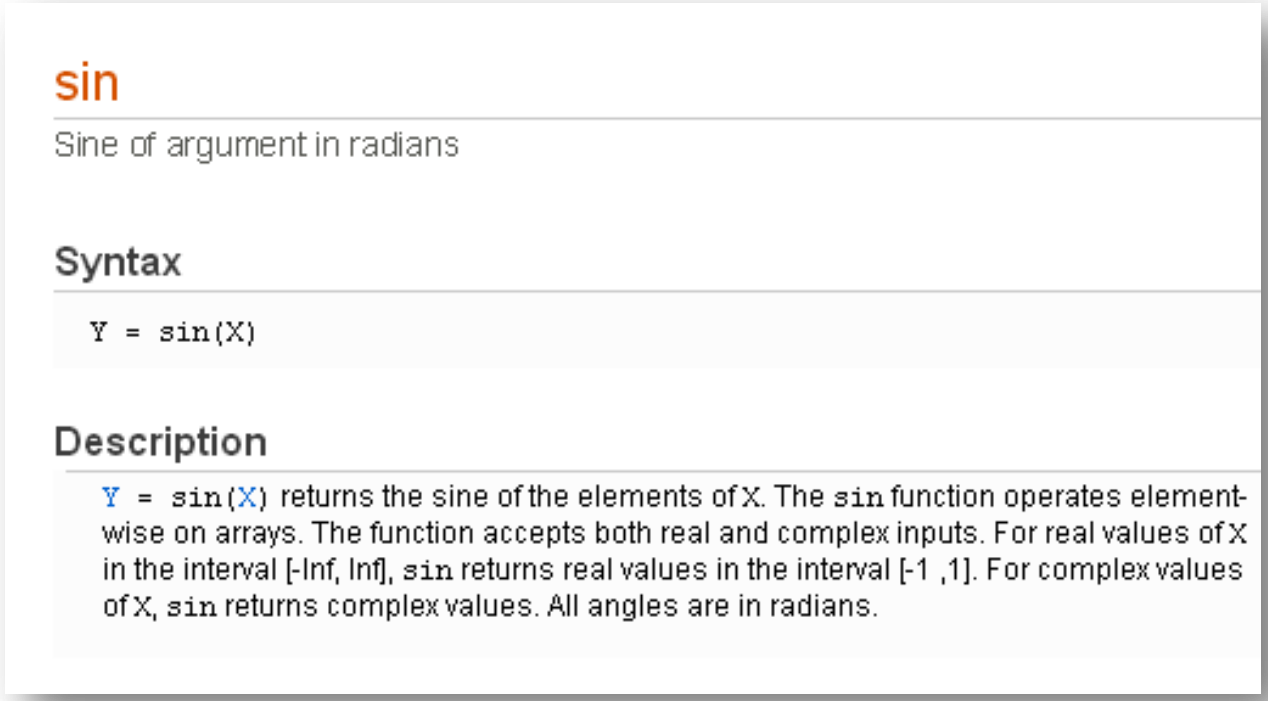
Outline

- MATLAB built-in (pre-defined) functions
- Using functions: the function call, input and output
- Built-ins accepting different type of function calls
- Anonymous functions
- Function m-files
- Global and local variables
- Functions with no input or output variables
- Choosing function output type
- Example Problem: script solution vs function solution

built-in functions

MATLAB has a large number of built-in (pre-defined or pre-programmed) functions. We have already used many of these functions, e.g. sin, plot, diag, size, etc.

Let's look at a few examples, starting with the sin function. The Help page provides the following information



sin

Sine of argument in radians

Syntax

```
Y = sin(X)
```

Description

`Y = sin(X)` returns the sine of the elements of X. The `sin` function operates element-wise on arrays. The function accepts both real and complex inputs. For real values of X in the interval [-Inf, Inf], `sin` returns real values in the interval [-1 ,1]. For complex values of X, `sin` returns complex values. All angles are in radians.

the sin function

Syntax: Y = sin(X)

The **sin** function accepts exactly 1 **input variable (argument)** and returns exactly 1 **output variable.** The **sin** acts elementwise, which means that the input and output are always of the same size.

How is the output of **sin** calculated? MATLAB does not implement built-in functions in the MATLAB language (they are typically written in C/C++ for optimum speed) so we cannot just look up the source code.

It is best to think of a built-in as a black box; we do not need to know how it works (or where the source code is located), only how to use it correctly.

Using the **sin** is simple (as we already know). We type the name **sin** followed by the input in brackets. Examples of **correct function calls**:

```
>> sin(pi/6)    % no output variable is specified
ans = % the output is assigned to default variable ans
    0.5000

>> x=sin(pi/6); % the output is assigned to x
```

calling functions: sin

What would be a wrong way of using the **`sin`** function? See below.

```
>> sin
Error using sin
Not enough input arguments.

>> sin()
Error using sin
Not enough input arguments.

>> sin(pi,0)
Error using sin
Too many input arguments.
```

When using a function for the first time, check the list of acceptable function calls (→Syntax).

calling functions: max

max

Largest elements in array

Syntax

```
M = max(A)
M = max(A,[],dim)
[M,I] = max( __ )

C = max(A,B)

__ = max( __ ,nanflag)
```

(this type of call won't be discussed here)

From the Description:

(1) **M = max(A)** returns the largest elements of A. If A is a matrix, then max(A) is a row vector containing the maximum value of each column

calling functions: max

(2) **M = max(A, [], dim)** returns the largest elements along dimension dim. For example, if A is a matrix, then max(A,[],2) is a column vector containing the maximum value of each row.

Why the [] ? Why not simply M = max(A, dim) ? Let's look at some examples.

```
>> A=randi([-5,5],3,5); disp(A)
    -4     3    -5     2    -1
    -1     5     4     3     2
     5     2     5     3    -4

>> disp(sort(A,2))                    % A is sorted along rows
    -5    -4    -1     2     3
    -1     2     3     4     5
    -4     2     3     5     5

>> disp(max(A,2))                     % maximum of A(i,j) or 2
     2     3     2     2     2  % see also (4)
     2     5     4     3     2
     5     2     5     3     2
```

calling functions: max

(3) **[M,I] = max(__)** finds the indices of the maximum values of A and returns them in output vector I, using any of the input arguments in the previous syntaxes. If the maximum value occurs more than once, then max returns the index corresponding to the first occurrence.

The (___) means that the extra output can be added to any of the previous function calls, e.g. [m,idx]=max(A) or [m,idx]=max(A, [], 2) .

```
>> A=randi([-5,5],3,5); disp(A)
    -4     3    -5     2    -1
    -1     5     4     3     2
     5     2     5     3    -4

>> [m,idx]=max(A)

m =
     5     5     5     3     2  % vector of column maximums

idx =
     3     2     3     2     2  % indices of max values
```

calling functions: max

(4) **C = max(A, B)** returns an array the same size as A and B with the largest elements taken from A or B. Either the dimensions of A and B are the same, or one can be a scalar.

This function call only allows 2 array arguments. This will NOT work: max(A,B,M)

```
>> B=3*ones(3,5); M=A.*B;
>> disp(max(A,B))
     3     3     3     3     3
     3     5     4     3     3
     5     3     5     3     3

>> disp(max(A,M))
    -4     9    -5     6    -1
    -1    15    12     9     6
    15     6    15     9    -4

>> disp(max(A,B,M))
Error using max
MAX with two matrices to compare and a working dimension is
not supported
```

calling functions: max

(4) **C = max(A, B)** returns an array the same size as A and B with the largest elements taken from A or B. Either the dimensions of A and B are the same, or one can be a scalar.

Now consider calls of the type max(A, scalar):

```
% elementwise comparison
% we get the same result using any of the following

>> disp(max(A,2))
     2     3     2     2     2
     2     5     4     3     2
     5     2     5     3     2

>> disp(max(A, 2*ones(3,5)))
     2     3     2     2     2
     2     5     4     3     2
     5     2     5     3     2

% so the 2 in max(A,2) does not mean dimension!
```

multiple output variables

```
% change the test matrix to include several 0s
%
>> A(A<4)=0; disp(A)
     0     0     0     0     0
     0     5     4     0     0
     5     0     5     0     0

% the vector of indices is an optional output of max
% only shown if the function call lists 2 output variables

>> [m,ind]=max(A)
m =
     5     5     5     0     0
ind =
     3     2     3     1     1

>> v=size(A); disp(v) % call with 1 output: 1 x 2 vector
     3     5

>> [n,m]=size(A); disp(n), disp(m) % call with 2 outputs
     3  % output 1, scalar
     5  % output 2, scalar
```

multiple output variables

find

Find indices and values of nonzero elements

(Check the HELP for a detailed description)

```
>> disp(A)
     0     0     0     0     0
     0     5     4     0     0
     5     0     5     0     0

% 1 output: column vector of linear indices
>> k=find(A); disp(k')
     3     5     8     9

% 2 outputs: vectors of row and column numbers
>> [p,q]=find(A); disp([p';q'])
     3     2     2     3
     1     2     3     3
% 3 outputs: row numbers, column numbers and nonzero elements
>> [p,q,v]=find(A); disp([p';q';v'])
     3     2     2     3
     1     2     3     3
     5     5     4     5
```

find vs logical arrays

The **find** function can be used to identify the *locations of elements satisfying certain conditions* – the actual elements can be easily selected using logical arrays.

```
>> disp(A)  % back to original A
    -4     3    -5     2    -1
    -1     5     4     3     2
     5     2     5     3    -4

>> disp(A>=4) % logical array
     0     0     0     0     0
     0     1     1     0     0
     1     0     1     0     0

% vector of elements satisfying the condition
>> disp(A(A>=4)')
     5     5     4     5

% vector of linear indices of selected elements
>> disp(find(A>=4)')
     3     5     8     9
```

Notice that A(find(A>=4)) gives the same answer as A(A>=4)!

anonymous functions

An **anonymous function** is user-defined function which is not stored in a function file. The function definition is typically a single line and it can only include a single executable statement.

functionName = @ (arg1, arg2,..) expression

functionName: usual (variable/script naming) rules apply
arg1, arg2,: input variables (arguments) – must be inside (), comma-separated
expression: single formula/statement

Try the following:

```
>> exist para; disp(ans) % check if the name para can be used
     0

>> para = @(x) (0.5*x.^2-3*x+4); disp(para(5))
    1.5000

>> x=0:0.01:6; y=para(x); plot(x,y), axis square
```

http://uk.mathworks.com/help/matlab/matlab_prog/anonymous-functions.html

function m-files

A (user-defined) function file is a text file with a .m extension.

User-written functions can be used the same way as built-in functions.

function m-file FORMAT

```
function [out1, out2, ..,] = functionName(input1, input2, …)

        function code
        (linking input and output variables)

end (this statement is optional)
```

Use the Editor to save the code below as an m-file - the filename MUST be `prb1`

```
function y=prb1(x)
% quadratic function of array input: y=0.5*x^2-3x+4

a=0.5; b=-3; c=4;
y=a*x.^2+b*x+c;
```

Do NOT try to run this!

function m-files

test your new function

```
% the Help displays the comment after the function definition line
% also called H1 line

>> help prb1
  quadratic function of array input: y=0.5*x^2-3x+4

>> z=sin(prb1(5)) % using the new function in an expression

z =
    0.9975

% create a plot of the sine of prb1
>> x=0:0.01:6; y=sin(prb1(x)); plot(x,y), axis square


% array input/output
>> l=-4:4; disp(l)
    -4    -3    -2    -1     0     1     2     3     4

>> disp(prb1(l))
   24.0000  17.5000  12.0000  7.5000  4.0000  1.5000  0  -0.5000  0
```

basic rules of function m-files

- the function **name must be the same** as the name of the function file
- ***DO NOT WRITE*** *ANYTHING BEFORE THE function LINE!*
- the **function** keyword must be in lowercase (**f**unction, not **F**unction!)
- finish all statements with a semicolon inside the function code
- ***DO NOT USE*** the commands clear, disp(), clc, input() in a function file
- use the **error** function (not the disp) to create an error message
- do not forget to define all **output variables**
- output variables do not have to be of the same size or even of the same type

using user-defined function

Where should the function m-file be stored?

(1) in the current directory, *OR*

(2) anywhere in the MATLAB (Search) Path

MATLAB can access all files in the folders on the search path.

Type path for a list of all folder (and subfolders) on your MATLAB path

To amend your path, go to **HOME → Environment → Set Path**

Why not run a function m-file?

The input variables are declared, but they are not given any values inside the function file. The values of the input variables will be passed on during the function call.

```
>> prb1  % attempt to run prb1 from the Editor
Error using prb1 (line 4)
Not enough input arguments.
```

no input and/or output variable

example function with **no input or output variable**

```
function sayhello()
disp('Hello!') % we really should not put disp here!
```

correct function call

```
>> sayhello
Hello!

>> sayhello()
Hello!
```

incorrect function call (there is no variable to assign to *s*)

```
>> s=sayhello
Error using sayhello
Too many output arguments.
```

Check the workspace – no new variable will appear! **`Ans`** will not be updated either.

no input and/or output variable

example function **without an output** variable

```
function saygoodbye(n)

for k=1:n
disp('Goodbye!') % we really should not put disp here!
end
```

example function call

```
>> saygoodbye(4)
Goodbye!
Goodbye!
Goodbye!
Goodbye!
```

cannot store the output because there no output variable is defined

```
>> s=saygoodbye(7)
Error using saygoodbye
Too many output arguments.
```

All the **Goodbye!**s will be lost as there is no way to store them!
Can you modify this function to make the **Goodbye!**s the actual output?

multiple input variables

amend **prb1** with additional input variables and save it under a new name

```
function y=prb2(a,b,c,x)
% quadratic function of array input: y=a*x^2+bx+c
% a, b, c must be scalars
y=a*x.^2+b*x+c;
```

test this function with a clear workspace

```
% input variables are identified by position
>> prb2(1,2,3,10)
ans =
   123
```

ans should now be the only variable in your Workspace

```
>> a=10;  % define variable a
>> x=prb2(2,2,2,a); % call the output x
>> disp(x)
   222
```

variable **a** in the Workspace does not clash with variable **a** in the function

similarly, Workspace variable **x** cannot interfere with function variable **x**

local variables

What happened to variables `a, b, c, x and y`?

Variables defined inside a function exist only inside the function.

*Each function has its own workspace which is independent of the **base workspace** and the workspace of any other function.*

*We can say that these variables are **local** to the function. The **scope** of local variables is restricted to the **function workspace.***

Variables in the function workspace do not clash with variables in the base workspace in any way.

*Variables declared as **global** can be shared between different functions and/or the base workspace.*

multiple output variables

Problem
write a function which determines the min, median and max of an input list (vector). Display an error message if the input is not in the correct format.

sample solution 1 – using a single vector output

```
function u = stats1(v)
% function to determine min, median, max of array input

if size(v,1) ~= 1 && size(v,2)~=1
    error('Vector input required!')
    else

q0=min(v);
q2=median(v);
q4=max(v);

% define output variable (1 x 3 vector)
u=[q0, q2, q4];
end
```

multiple output variables

sample solution 2 – using 3 scalar output variables

```
function [q0, q2, q4] = stats2(v)
% function to determine min, median, max of array input

if ~isvector(v)
    error('Vector input required!')
    else

% define output variables
q0=min(v);
q2=median(v);
q4=max(v);

end
```

The functions **`stats1`** and **`stats2`** are very similar but they must be called differently!

multiple output variables

Example function calls

```
% define a test variable
>> l=randi(100,1,10); disp(l)
    29    76    76    39    57     8     6    54    78    94

>> v=stats1(l); disp(v)
    6.0000   55.5000   94.0000 % correct call


>> x=stats2(l); disp(x) % incomplete call
       6                % output 2 and 3 are ignored


>> [x, y, z]=stats2(l)

x = 6
y = 55.5000
z = 94

>> [x, y, z]=stats1(l) % incorrect call
Error using stats1
Too many output arguments.
```

multiple output variables

It may seem that **stats1** is easier to use (because it is easier to call). But this type of solution (squeezing all calculated variables into a single output variable) is not always practical.

```
>> v=stats1(l); disp(v)
    6.0000   55.5000   94.0000
```

What if we need the min, median and max values separately?

We can refer to them as v(1), v(2) and v(3) – but this may not be convenient.

Could we assign these 3 values the variable names a, b and c in just 1 step?

The short answer is NO (the long answer involves cell arrays)

→ use multiple output variables whenever appropriate

script solution vs function solution

We will now revisit the conditional array replacement problem in Topic 6. The old script solution can be converted into a function in just a few steps.

Replace each element of a given array (A) with

+1 if the element is even

-1 if the element is odd

0 otherwise

We will have to:

- select a valid function name and use it consistently
- add the function definition line, specifying 1 array input and 1 array output
- remove the input function
- remove any clc, clear type commands
- remove disp
- check that the output is properly defined

conditional replacement problem: script solution

```
clc
A = input('Enter an array > ');
[n,m]=size(A);

B=A;
for i=1:n
   for j=1:m
        if       mod(A(i,j),2)==0,
                      B(i,j)=1;
        elseif   mod(A(i,j),2)==1,
                      B(i,j)=-1;
        else
                      B(i,j)=0;
        end
    end
end
disp(B)
```

conditional replacement problem: function solution 1

```
function B=condr(A)
% to replace array elements with 1 if even, -1 if odd, 0 otherwise

[n,m]=size(A);
B=A;
for i=1:n
   for j=1:m
        if       mod(A(i,j),2)==0,
                      B(i,j)=1;
        elseif   mod(A(i,j),2)==1,
                      B(i,j)=-1;
        else
                      B(i,j)=0;
        end
    end
end
```

conditional replacement problem: function solution 2

An alternative solution using logical indexing:

```
function B=condrl(A)
% to replace array elements with 1 if even, -1 if odd, 0 otherwise

B=A;

s1=mod(A,2)==0;
s2=mod(A,2)==1;

B(s1)=1;
B(s2)=-1;
B(~s1 & ~s2)=0;
```

The logical arrays s1 and s2 were introduced to improve the readability of the code.