Lecture 3: Functions

Math 98, Fall 2024

Agenda

- Functions
- Exercises
- Anonymous Functions
- Functions vs. Scripts
- Local Functions
- `nargin`/`return`
- Nested Functions

Functions: Motivation

We have functions in addition to scripts because functions are

1. reusable
   - A function replaces a repeated block of code.
2. simplifying
   - A function organizes groups of code, and can be written in a separate file. Makes the code easier to read.
3. changeable
   - Easier to change a procedure if it's packaged by a single function.
4. modular
   - Reduce presence of intermediate variables

Functions: Structure

Any function we write should have the following format:

```
%%%Name.m%%%
function [output vars] = Name(input)

    % code here

end
```

The name of the function should match the name of the M-file. Built-in Matlab functions use all lowercase letters, so use at least one uppercase letter to avoid conflict.

Functions: Example

Sample function:

```
function [n] = myfun(m)
    n = m + 1;
end
```

Using the function:

```
>> myfun(5)
ans =
    6
>> n = myfun(9)
n =
    10
>> blah = myfun(pi)
blah =
    4.1416
```

Functions: Forgetting to assign output

Sample function:

```
function [n] = myfun(m)
    m + 1;
end
```

Using the function:

```
>> myfun(10)
>>
```

Nothing happens!! No output was assigned.

Functions: Intermediate Variables

Sample function:

```
function [n] = myfun(m)
    a = m + 1;
    b = 2*a;
    n = b - 2;
end
```

Using the function:

```
>> n = myfun(4)
n =
    8
```

The 'outside world' knows nothing about the a and b that were created. What happens in the function stays in the function.....

Exercise: `myfun.m`

Write a function of the form

```
function [sum, diff, prod] = myfun(a, b)
```

that takes in two numbers a, b and returns their sum, difference, and product. Run each of the following lines and understand the result.

```
>> myfun(3, 4)
>> sum = myfun(3, 4)
>> prod = myfun(3, 4)
>> sum = myfun(3)
>> price = 5; units = 4; [~, ~, rev] = myfun(price, units)
```

Exercise: `sumrowcols.m`

Write a function of the form

```
function [colsum, rowsum] = sumrowcols(A)
```

that takes in a matrix $m \times n$ matrix A and returns vectors `colsum` and `rowsum` of the column sums and row sums of A, respectively.

Exercise: `checkerboard.m`

Write a function of the form

```
function A = checkerboard(n, m)
```

that takes two positive integers `n` and `m` as inputs and returns a matrix `A` such that every element of the $n \times m$ output matrix for which the sum of its indices is even is 1. All other entries are zero.

Here is a sample output.

```
>> checkerboard(4, 5)
ans =
     1  0  1  0  1
     0  1  0  1  0
     1  0  1  0  1
     0  1  0  1  0
```

Function Handles

A function handle is a Matlab variable that allows us to reference functions indirectly. Use them to include functions as inputs to or outputs from other functions.

```
>> integral(cos,0,1)
Error using cos
Not enough input arguments.
>> integral(@cos,0,1)
ans =
    0.8415
```

Anonymous Functions

A way to define functions in the middle of a Matlab script or in the command line. Takes the form `functionName = @(inputs)(output)`, and returns the function handle `functionName`.

```
>> f = @(x,y)(x^2-y);
>> f(10, 3)
ans =
    97
>> fzero(@(x)(x^2-2), 1.5)
ans =
    1.4142
```

Useful when defining functions with simple expressions.

Anonymous Functions: Examples

Here are some more functions:

```
>> b = 3; c = 5;
>> f1 = @(x)(x^3 + b*x + c);
>> fzero(f1,0)
ans =
    -1.1542
>> b = 2; c = -1;
>> f2 = @(x)(x^3 + b*x + c);
>> fzero(f2, 0)
ans =
    0.4534
```

Question: does changing the values of `b` and `c` change the function `f1`, or will `f1` and `f2` be different functions?

Functions vs. Scripts

Scripts:

- No inputs or outputs - Matlab just executes all commands
  - (Unless you use input)
- Operates on existing data in the workspace
- Variables created remain in the workspace

Functions:

- Accept inputs and return outputs
- Create their own separate workspace
- Only requrested output variables get saved

Functions vs. Scripts: Accesing Variables in Workspace

Functions do not access variables stored in the main Workspace.

```
%%%exampleFunction.m%%%

function w = exampleFunction(x,y)
    w = x + y + z;
end
```

```
>> z = 5; a = exampleFunction(2,3);
Undefined function or variable 'z'.
```

Functions vs. Scripts: Saving Variables in Workspace

Functions do not save variables back to the main Workspace unless they are requested as outputs.

```
%%%exampleFunction.m%%%

function a = exampleFunction(x,y)
    a = x + y; b = 101;
end
```

```
>> a = exampleFunction(2,3); disp(a);
    5
>> disp(b)
Undefined function or variable 'b'.
```

Functions vs. Scripts: Conflicting Variables

Because functions use their own workspace, variables named inside a function cannot conflict with variables of the same name outside the function.

```
%%%exampleFunction.m%%%

function a = exampleFunction(x,y)
    b = 100; a = x + y + b;
end
```

```
>> b = -300; a = exampleFunction(40,5); disp(a);
    145
>> disp(b);
    -300
```

Local Functions

We can define more than one function in a single file.

```
%%%myStats.m%%%
function avg = myStats(x)
% takes a vector and returns the average
    n = length(x);
    avg = myMean(x,n);
end

function m = myMean(v,n)
% it takes a vector and its length, returns the mean
    m = sum(v)/n;
end
```

Only the first function (the **main** function) can be called form other programs or the command line.

Local Functions: In Scripts

We can also define local functions in scripts:

```
v = 1.5;
L =  myLength(v);
fprintf('the length of v is %f \n', L);

function len = myLength(x)
    len = sqrt(sum(x.^2));
end
```

Any function definitions must come at the end of the script.

Commenting

As with built-in Matlab functions, we can use comments and help to inform how each function is properly used.

```
>> help myStats
 takes a vector and returns the average
>> help myStats>myMean
   it takes a vector and its length, returns the mean
```

Any function definitions must come at the end of the script.

nargin/return

When used in the code for a function, nargin is the number of inputs specified by the user. Handy when setting default values for inputs.

```
%%%addMe.m%%%
%Input: one or two floating point numbers
%Output: addMe(x,y) returns x + y; addMe(x) returns 2*x
function s = addMe(x,y)
    if (nargin == 1)
        s = x + x;
    elseif (nargin == 2)
        s = x + y;
    else
        fprintf('Read the comments! \n');
        return
    end
end
```

return automatically halts the function

Exercise: `myCosine.m`

Write a function `myCosine(theta,units)` that returns the cosine of an angle. If the second parameter is 'deg', convert the angle to radians with a local function `DegToRadians(x)` before using Matlab's `cos`. In all other cases (including no second parameter), assume the angle is in radians.

```
>> myCosine(180, 'deg')
ans =
    -1
>> myCosine(pi, 'rad')
ans =
    -1
>> myCosine(pi)
ans =
    -1
```

Problem

We would like to find the roots of the polynomial

$$p(x) = x^3 + bx + c$$

for various numbers $b, c \in \mathbb{R}$.

- How can we produce this family of functions?
- What tools does Matlab have to solve this problem?

Nested Functions

Nested functions are functions defined within other functions.

```
function f = makeCubic(b,c)
    function y = myCubic(x)
        y = x.^3 + b*x + c;
    end

   f = @myCubic;
end
```

They can access variables in the workspace of the parent function, and don't need to be defined at the end of the code in the parent function.