SECTION 6: USER-DEFINED FUNCTIONS

ENGR 112 – Introduction to Engineering Computing

User-Defined Functions

- By now you're accustomed to using ***built-in MATLAB functions*** in your m-files
- Consider, for example, `mean.m`
  - Commonly-used function
  - Need not write code each time an average is calculated
  - An m-file – written using other MATLAB functions
- Functions allow ***reuse of commonly-used blocks of code***
  - Executable from any m-file or the command line
- Can create ***user-defined functions*** as well
  - Just like built-in functions – similar syntax, structure, reusability, etc.

User-Defined Functions

- Functions are a specific type of m-file
  - Function m-files start with the word `function`
  - Can ***accept input arguments*** and ***return outputs***
  - Useful for tasks that must be performed repeatedly
- Functions can be called from the command line, from within m-files, or from within other functions
- ***Variables within a function are local in scope***
  - Internal variables – not outputs – are not saved to the workspace after execution
  - Workspace variables not available inside a function, unless passed in as input arguments

Anatomy of a Function

Function m-file must begin with the word ‘`function`’

Output(s)

Function Name

Input Argument(s)

Help comments – displayed when help is requested at the command line:

```
>> help far2cel
  Converts temperature from degrees Farenheit
  to degree Celsius and to Kelvin

  Input:
    Tf: temperature in degrees Farenheit

  Output:
    Tc: temperature in degrees Celsius
    Tk: temperature in Kelvin
```

```
function [Tc,Tk] = far2cel(Tf)
% Converts temperature from degrees Farenheit
% to degree Celsius and to Kelvin
%
% Input:
%   Tf: temperature in degrees Farenheit
%
% Output:
%   Tc: temperature in degrees Celsius
%   Tk: temperature in Kelvin

Tc = (Tf - 32)/1.8; % calculate Celsius temperature
Tk = Tc + 273;      % convert to Kelvin
end
```

MATLAB commands that define the function

Terminate the function with the word ‘end’

Always comment your code

Commenting Functions

- Any function – built-in or user-defined – is accessible by the command-line help system
  - Type: `help` *`functionName`*
- Help text that appears is the first comment block following the function declaration in the function m-file
  - Make this comment block particularly descriptive and detailed
- Comments are particularly important for functions
  - Often reused long after they are written
  - Often used by other users

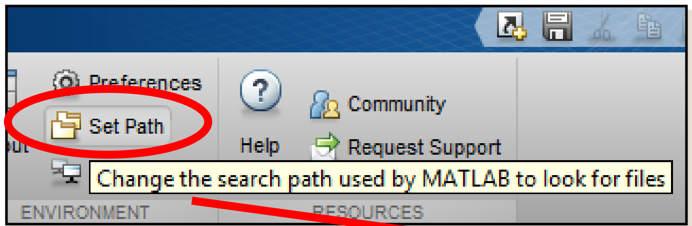
M-Files vs. Functions

- Most code you write in MATLAB can be written as regular (non-function) m-files
- Functions are most useful for ***frequently-repeated operations***

	M-Files	Functions
Scope of variables	Global Facilitates debugging	Local Use debugger to access internal function variables
Inputs/Outputs	No All variables in memory at the time of execution are available. All variables remain in the workspace following execution.	Yes
Reuse	Yes	Yes
Help contents	No	Yes

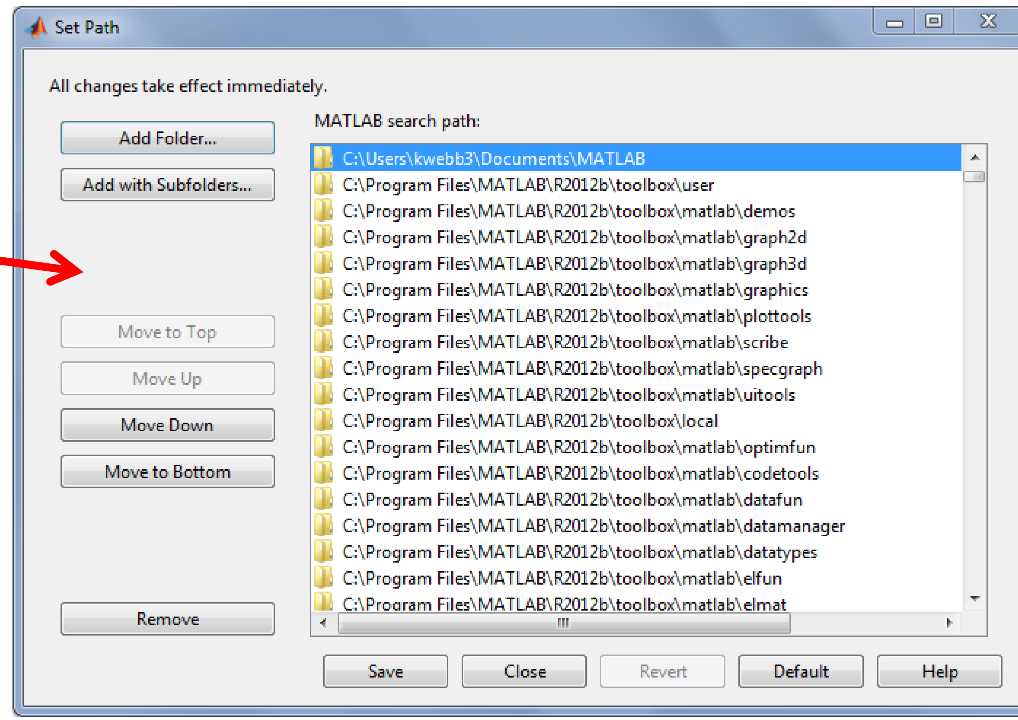
The MATLAB Path

- All functions outside of the PWD – user-defined or built-in – must be in the ***path*** to be accessed



- Add a directory to your path for frequently-used functions, e.g.,

C:\Users\Documents\MATLAB\



Function Inputs and Outputs


```
function y = func(x)
```

- Here, *x* is the ***input*** passed to the function `func`
  - Passed to the function from the calling m-file
  - Not defined within the function
- y is the ***output*** returned from the function
  - Defined within the function
  - Passed out to the calling m-file
    - The only function variable available upon return from the function call

Multiple Inputs and Outputs


```
function [y1,y2] = func(x1,x2,x3)
```

- Functions may have more than one input and/or output
- Here, three inputs: `x1`, `x2`, and `x3`

  and two outputs: `y1` and `y2`
  - Inputs separated by commas
  - Outputs enclosed in square brackets and separated by commas

Function – Example

- Consider a function that converts a distance in kilometers to a distance in both miles and feet
  - One input, two outputs

```
function [mi,ft] = km2mift(km)
% Converts a distance specified in kilometers to both miles and feet
%
% Input:
%       km: distance in kilometers
% Outputs:
%       mi: distance in miles
%       ft: distance in feet

mi = km*0.62137;
ft = mi*5280;

end
```

Command Window

```
>> [miles, feet] = km2mift(42.2)
miles =
   26.2218
feet =
   1.3845e+05
>>
```

Optional Input Arguments

Optional Input Arguments

- Functions often have ***optional input arguments***
  - Variable number of input arguments may be required when calling the function
  - Optional inputs may have ***default values***
  - Function behavior may differ depending on what inputs are specified

- For example, MATLAB's `mean.m` function:

```
y = mean(x)
```

  - Optionally, specify the dimension along which to calculate mean values:

```
y = mean(x,dim)
```

Optional Input Arguments

- `mean.m` allows you to specify the dimension along which the mean is calculated
  - Default is `dim = 1`
    - If `dim` is not specified, it is set to `1` within the function
    - Calculate mean values of columns
  - Setting `dim = 2` calculates mean values of rows

Command Window

```
>> A = [1 2 3;4 5 6;7 8 9]
A =
     1     2     3
     4     5     6
     7     8     9
>> mean(A)
ans =
     4     5     6
>> mean(A,1)
ans =
     4     5     6
>> mean(A,2)
ans =
     2
     5
     8
>>
```

Optional Input Arguments

- Just like built-in functions, user-defined functions can also have optional inputs
- Code executed when function is called depends on the ***number of input arguments***
- `nargin.m` returns the number of input arguments passed to a function
  - Allows for checking ***how many input arguments*** were specified
  - Use ***conditional statements*** to control code branching
  - If an input was not specified, set it to a ***default value***

Optional Inputs – Example 1

- For example, consider a function designed to return a vector of values between `xi` and `xf`
- Third input argument, N, the number of elements in the output vector, is optional
  - Default is N = `10`

```matlab
function x = vecgen(xi,xf,N)
% Generates a vector of N values
% between xi and xf
% Inputs:
%        xi: first value in x
%        xf: last value in x
%         N: number of elements in x
% Outputs:
%         x: the vector returned

if nargin == 2, N = 10, end

dx = (xf - xi)/(N-1);

for i = 1:N
    x(i) = xi + (i-1)*dx;
end

end
```

Optional Input Arguments

- Sometimes we want to allow for optional inputs in the middle, not at the end, of the input list
  - For example, maybe the second of three inputs is optional (or the second *and* third inputs)
  - `nargin.m` alone won't work here
  - Can't differentiate between skipping the second of three inputs or the third of three inputs
    - `nargin` == 2 in both cases
- Instead of skipping the input altogether, pass an ***empty set***, `[ ]`, in its place

Optional Inputs – Example 2

- Revisit the same vector-generating function
- Now both the ***first input***, `xi`, and the ***third input***, N, are optional
  - If `xi` is not specified it defaults to xi = 0
  - Single input, intended to be xf, is assumed to be xi, the first listed input argument
    - Must assign the single input argument to xf

```
function x = vecgen2(xi,xf,N)
% Generates a vector of N values
% between xi and xf
% Inputs:
%       xi: first value in x
%       xf: last value in x
%        N: number of elements in x
% Outputs:
%        x: the vector returned

if nargin == 1      % only xf specified
    xf = xi;        % input assumed to be xi
    xi = 0;
    N = 10;
elseif nargin == 2  % xi and xf specified
    N = 10;
end

if isempty(xi)
    xi = 0;
end

dx = (xf - xi)/(N-1);

for i = 1:N
    x(i) = xi + (i-1)*dx;
end

end
```

Error Checking Using `nargin.m`

- Can use `nargin.m` to provide error checking
  - Ensure that the correct number of inputs were specified
- Use `error.m` to terminate execution and display an error message

```
function x = vecgen3(xi,xf,N)
% Generates a vector of N values
% between xi and xf
% Inputs:
%       xi: first value in x
%       xf: last value in x
%        N: number of elements in x
% Outputs:
%        x: the vector returned

if (nargin ~=2) || (nargin ~= 3)
    error('Incorrect number of inputs specified for vecgen3.m')
end

if nargin == 2, N = 10, end

dx = (xf - xi)/(N-1);

for i = 1:N
    x(i) = xi + (i-1)*dx;
end

end
```

Command Window

```
>> vecgen3(5)
Error using vecgen3 (line 12)
Incorrect number of inputs specified for vecgen3.m

fx >>
```

Sub-Functions

Sub-Functions

- Functions are useful for blocks of code that get called repeatedly
  - We often have such blocks within functions themselves
  - Can define additional functions in separate m-files
  - Or, if the code is only useful within that specific function, define a ***sub-function***
- ***Sub-Functions***
  - A function defined within another function m-file
  - Local scope: only available from within that function
  - Organizes, simplifies overall function code

Sub-Functions – Example

- Here, two sub-functions are defined and called from within the main function

Main function

```
function [Tc,Tk] = far2celk(Tf)
% Converts temperature from degrees Farenheit
% to degree Celsius and to Kelvin
%
% Input:
%   Tf: temperature in degrees Farenheit
%
% Output:
%   Tc: temperature in degrees Celsius
%   Tk: temperature in Kelvin

if Tf < -459.67
    error('Please enter a value greater than absolute zero.');
end

Tc = far2cel(Tf);
Tk = cel2k(Tc);
end
```

Sub-function 1

```
function Tc = far2cel(Tf)
% sub-function to convert from F to C

Tc =(Tf - 32)/1.8;      % calculate Celsius temperature

end
```

Sub-function 2

```
function Tk = cel2k(Tc)
% sub-function to convert from C to K

Tk = Tc + 273;      % calculate temperature in Kelvin

end
```

Anonymous Functions

Anonymous Functions

- It is often desirable to create a function without having to create a separate function file for it
- ***Anonymous functions***:
  - Can be defined within an m-file or at the command line
  - Function data type is `function_handle`
    - A ***pointer to the function***
  - Can accept inputs, return outputs
  - May contain only a ***single MATLAB expression***
    - Only one output
  - Useful for ***passing functions to functions***
    - E.g. using quad.m (a built-in MATLAB function) to integrate a mathematical function (a user-defined function)

Anonymous Functions - Syntax


```
fhandle = @(arglist) expression
```

@ symbol generates a handle for the function

Function definition

- A ***single*** executable MATLAB expression
- E.g. `x.^2+3*y;`

Function name

- A variable of type `function_handle`
- Pointer to the function

A list of input variables

- E.g. `@(x,y);`
- Note that outputs are not explicitly defined

Anonymous Functions – Examples

Command Window

```
>> half = @(x) x/2;
>> b = half(35)

b =

   17.5000

>> resp = @(tau,t) 1-exp(-t/tau);
>> z = resp(2,4)

z =

    0.8647

>> y = resp(2,[0:2:20])

y =

  Columns 1 through 4

         0    0.6321    0.8647    0.9502

  Columns 5 through 8

    0.9817    0.9933    0.9975    0.9991

  Columns 9 through 11

    0.9997    0.9999    1.0000

fx >>
```

- Simple function that returns half of the input value
- May have multiple inputs
  - First-order system response – inputs: time constant, value of time
- Inputs may be vectors
- Outputs may be vectors as well

Passing Functions to Functions

- We often want to perform MATLAB functions on other functions
  - E.g. integration, roots finding, optimization, solution of differential equations – these are ***function functions***
  - This is the real value of anonymous functions

Command Window

```
>> f = @(x) sqrt(x)+log(x);
>> g = quad(f,0,10)

g =

   34.1077

>>
```

- Define an anonymous function
- Pass the associated function handle to the function as an input
- Here, integrate the function, f, from 0 to 10 using MATLAB's quad.m function

Function Function – Example

- Consider a function that calculates the mean of a mathematical function evaluated at a vector of independent variable values
- Inputs:
  - Function handle
  - Vector of x values
- Output:
  - Mean value of $y = f(x)$

```
function favg = fmean(func,x)
% Calculates the mean of func(x)
%
% Inputs:
%       func: function handle
%       x: values at which to evaluate func
% Output:
%       favg: mean value of func(x)

y = func(x);
favg = mean(y);

end
```

```
% funcfuncEx.m

clear all; clc

func = @(x) 0.5*x.^5 - 12*x.^3 + 15*x.^2 - 9;

x = -5:0.01:5;

favg = fmean(func,x)
```

Command Window

```
favg =

  116.2500

>>
```

Recursion

Recursive Functions

- ***Recursion*** is a problem solving approach in which a larger problem is solved by solving many smaller, self-similar problems
- A ***recursive function*** is one that calls itself
  - Each time it calls itself, it, again, calls itself
- Two components to a recursive function:
  - A ***base case***
    - A single case that can be solved without recursion
  - A ***general case***
    - A recursive relationship, ultimately leading to the base case

Recursion Example 1 – Factorial

- We have considered ***iterative*** algorithms for computing $y = n!$
  - for loop, while loop
- Factorial can also be computed using recursion
  - It can be defined with a base case and a general case:

$$n! = \begin{cases} 1 & n = 1 \\ n * (n-1)! & n > 1 \end{cases}$$

  - The general case leads back to the base case
    - $n!$ defined in terms of $(n-1)!$, which is, in turn, defined in terms of $(n-2)!$, and so on
    - Ultimately, the base case, for $n = 1$, is reached

Recursion Example 1 – Factorial

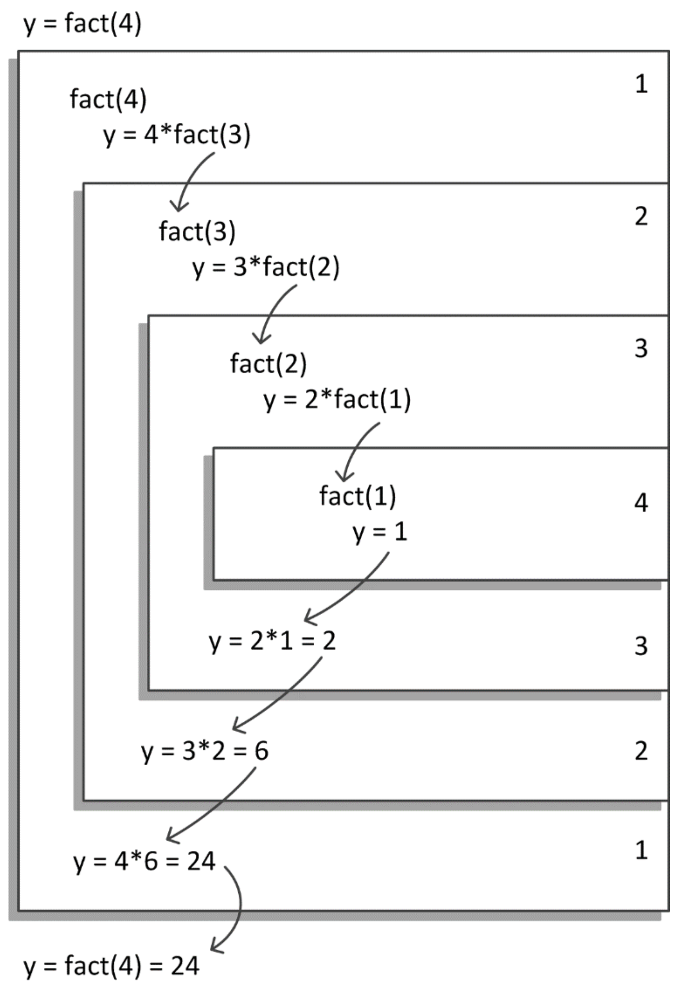
$$n! = \begin{cases} 1 & x = 1 \\ x * (x-1)! & x > 1 \end{cases}$$

- The general case is a recursive relationship, because it defines the factorial function using the factorial function
  - The function calls itself
- In MATLAB:

```
function y = fact(n)
% Compute the factorial of a positive interger, n,
% using a recursive algorithm.

if n == 1
    y = 1;
else
    y = n*fact(n-1);
end

```

Recursion Example 1 – Factorial


```
function y = fact(n)
% Compute the factorial of a positive interger, n,
% using a recursive algorithm.

if n == 1
    y = 1;
else
    y = n*fact(n-1);
end

```



- Consider, for example: $y = 4!$
- `fact.m` recursively called four times
- Fourth function call terminates first, once the base case is reached
- Function calls terminate in reverse order
  - Function call doesn't terminate until all successive calls have terminated

Recursion Example 2 – Binary Search

- A common search algorithm is the ***binary search***
  - Similar to searching for a name in a phone book or a word in a dictionary
  - ***Look at the middle value*** to determine if the search item is in the ***upper or lower half***
  - Look at the middle value of the half that contains the search item to determine if it is in that half's upper or lower half, …
- The ***search function gets called recursively***, each time on half of the previous set
  - Search range shrinks by half on each function call
  - Recursion continues until the middle value is the search item – this is the required ***base case***

Recursion Example 2 – Binary Search

- ***Recursive binary search*** – the basic algorithm:
  - Find the index, i, of x in the sorted list, A, in the range of $A(i_{low}\!:i_{high})$

1) Calculate the middle index of $A(i_{low}\!:i_{high})$:

$$i_{mid} = \text{floor}\left(\frac{i_{low} + i_{high}}{2}\right)$$

2) If $A(i_{mid}) == x$, then $i = i_{mid}$, and we're done
3) If $A(i_{mid}) > x$, repeat the algorithm for $A(i_{low}\!:i_{mid} - 1)$
4) If $A(i_{mid}) < x$, repeat the algorithm for $A(i_{mid} + 1\!:i_{high})$

Recursion Example 2 – Binary Search

- Find the index of the $x = 9$ in:

$$A = [0, 1, 3, 5, 6, 7, 9, 12, 16, 20]$$

- $A(i_{mid}) = A(5) = 6$
  - $A(i_{mid}) < x$
  - Start over for $A(6:10)$

$$A = [0, 1, 3, 5, 6, 7, 9, 12, 16, 20]$$

- $A(i_{mid}) = A(8) = 12$
  - $A(i_{mid}) > x$
  - Start over for $A(6:7)$

$$A = [0, 1, 3, 5, 6, 7, 9, 12, 16, 20]$$

- $A(i_{mid}) = A(6) = 7$
  - $A(i_{mid}) < x$
  - Start over for $A(7)$

$$A = [0, 1, 3, 5, 6, 7, 9, 12, 16, 20]$$

- $A(i_{mid}) = A(7) = 9$
  - $A(i_{mid}) == x$
  - $i = i_{mid} = 7$

Recursion Example 2 – Binary Search

- Recursive binary search algorithm in MATLAB
- Base case for `A(imid) == x`
- Function is called recursively on successively halved ranges until base case is reached

```
function ind = binsearch(A,x,ilow,ihigh)
% Recursive algorithm for locating the index of
% a search item within an ordered list. Search value
% must be in the list.
% Inputs:
%        A: ordered list
%        x: value whose index is to be found
%        ilow: lower bound index on search region
%        ihigh: upper bound index on search region
% Output:
%        ind: index x in A, i.e., A(ind) == x

imid = floor((ilow + ihigh)/2);

if A(imid) == x
    ind = imid;
elseif A(imid) > x
    ind = binsearch(A,x,ilow,imid-1);
else
    ind = binsearch(A,x,imid+1,ihigh);
end

end
```

Recursion Example 2 – Binary Search

- A=[0,1,3,5,6,7,9,12,16,20]
- x=9
- ind = binsearch(A,x,1,10)
  - ind = 7

```
function ind = binsearch(A,x,ilow,ihigh)
% Recursive algorithm for locating the index of
% a search item within an ordered list. Search value
% must be in the list.
% Inputs:
%       A: ordered list
%       x: value whose index is to be found
%       ilow: lower bound index on search region
%       ihigh: upper bound index on search region
% Output:
%       ind: index x in A, i.e., A(ind) == x

imid = floor((ilow + ihigh)/2);

if A(imid) == x
    ind = imid;
elseif A(imid) > x
    ind = binsearch(A,x,ilow,imid-1);
else
    ind = binsearch(A,x,imid+1,ihigh);
end

end
```