Week 2: aim to cover

- Further MATLAB (Lecture 3)
- Intro to NumPy/SciPy (Lab 2)
- Plotting with MATLAB/Matplotlib (Lecture 4)

Datatypes

Originally, MATLAB had only 2D arrays of double-precision floating point variables, i.e. that needed for numerical linear algebra. This is still the default datatype(class). The vectors we covered before are really 2D arrays with one dimension equal to 1.

Now there are more datatypes(classes):

- char (to form strings and store text)
- logical(Boolean), with values `true` and `false`
- single-precision floating point
- 8 kinds of integer types
- cell arrays
- structures
- tables

What we need ...

Cell arrays, tables and structures are useful for storing heterogeneous data (variables of different types) whereas arrays must have all elements with the same type. Structures are used to specify options for some functions we'll use.

Arrays can be n-dimensional (require n indices) but we need only 2D arrays. Double or logical arrays can be full or sparse — we only deal with full (dense) arrays.


```
class(1.2)
class(1)
class(2>1)
class('t')
class({1,'t'})
```

```
double    double    logical     char     cell
```

Finally there is the scalar (1×1) class of Function Handle (see below).

Coding conventions

I have put on the website a document with suggested coding conventions in MATLAB to help you write clearer code.

They cover things like:

- naming conventions
- which program statements are clearer

- layout, comments and documentation
- files and organization

Don't be like the student in 2017 who only used single-letter variable names!!

To use sparingly

To break out of a loop, use `break`.
To stop the current iteration and go straight to the next iteration, use `continue`
These can often be avoided by better logical tests.

switch

A selection between more than 3 or 4 options should use a `switch` rather than a complicated chain of `if. ... elseif..... elseif .`

```
method = 'Bilinear';

switch lower(method)
      case {'linear','bilinear'}
            disp('Method is linear')
      case 'cubic'
            disp('Method is cubic')
      case 'nearest'
            disp('Method is nearest')
      otherwise
            disp('Unknown method.')
      end
```

```
Method is linear
```

Function handles

In numerical computing, it is common to write functions that solve some problem where 1 or more functions are inputs. These are called function functions. Examples include root-finders (such as Newton's method), numerical quadrature (definite integrals) and solving differential equations numerically.

Where functions take an unknown function as an argument, (the actual function is only specified when the function function is called) we specify a function argument using a special data type: the function handle.

An example: quadrature

An example is MATLAB's numerical quadrature function `integral`

```
help integral
```



```
integral  Numerically evaluate integral.
    Q = integral(FUN,A,B) approximates the integral of function FUN
    to B using global adaptive quadrature and default error toleranc
```



```
    FUN must be a function handle. A and B can be -Inf or Inf. If bo
    finite, they can be complex. If at least one is complex, integra
    approximates the path integral from A to B over a straight line

    For scalar-valued problems the function Y = FUN(X) must accept a
    argument X and return a vector result Y, the integrand function
    evaluated at each element of X.
```

An example: quadrature

It takes a function handle as input and the interval endpoints a, b. To get the function handle of a built-in function , use `@funcname`


```
Q = integral(@sin,0,pi)
```

```
Q =

    2.0000
```

To get the function handle for one of your primary functions, again use @

```
type Sinc  % upper case to avoid shadowing sinc.m
Q = integral(@Sinc,0,pi)
```

```
function y = Sinc( x )
%SINC Sinc function
if x==0
    y = 1;
else
    y = sin(x)./x;
end
```

```
Q =

    1.8519
```


We will use function handles to tell MATLAB which differential equation to solve.

Anonymous functions

If the function is very simple (one MATLAB expression) it is possible to easily define a function handle in 1 line using anonymous functions.


```
Q = integral(@(x) x.^3 -3*x +2.0,pi)
```



```
Q =

   15.8311
```

Notice that we didn't have to give the function a name, hence anonymous.

But we can give it a name, storing it as a function handle

```
f1 = @(x) x.^3-3*x+2;   % note array operators
```

which we can then use like any other function

```
y=f1(3)
Q = integral(f1,0,pi)  % note no @ - it's already a handle
```



```
y =

    20


Q =

   15.8311
```

I use anonymous functions in my demo programs.

Passing information between functions

The safest way is to use input/output arguments
e.g. form a structure containing all the variables (and their names) you want to send to another function.
There are ways to share variables between function explicitly

`global` `persistent`

but these are regarded as unsafe and should be used sparingly. Variables declared as `global` are all stored in the base workspace!
There are other ways to share variables between functions in a more limited way, for functions that are stored in the same M-file. This allows more structuring of how functions can be written and variables shared.
You can only call a function from the command window or a script or another function M-file where that primary function `myfun` is stored in a file named `myfun.m`.

Local functions aka subfunctions

If you need to break up your function into smaller tasks, you can write them as local functions (subfunctions) in the same file. Each one is kept below the primary function, and every function should finish with an `end` statement.


```
function [root5,count] = NewtonFuncSubfunc(x0,tol)
if validInput(x0,tol)
    <snip>
else
    disp('Error: not valid input');
    root5 = nan;    % give return value
    count = 0;
end; end

function result = validInput(x,tol)
% a subfunction of NewtonFuncSubfunc
result = true;  % default
if x == 0 || tol <= 0 result = false;  end
end
```

```
[x,number] = NewtonFuncSubfunc(2,1e-10);
disp([x number]);
[x,number] = NewtonFuncSubfunc(0,1e-10);
disp([x number]);
[x,number] = NewtonFuncSubfunc(2,1e-10);
disp([x number]);
```



```
    2.2361    4.0000

Error: not valid input
   NaN     0

Error: not valid input
   NaN     0
```

Scope in subfunctions

The subfunction `validInput` can only be called by `NewtonFuncSubfunc` not by other functions or scripts. The variables inside a subfunction are local to that function and cannot be seen by the primary function or other subfunctions in the same file. Hence why they are also called local functions. You can call other subfunctions from a subfunction in the same file.


```
subFuncTest
```

```
The value of x in primary is 100
The value of x in subfunc1 is  11
The value of x in primary is 100
The value of x in subfunc3 is  33
The value of x in subfunc1 is  11
The value of x in primary is  33
```

Nested functions

The chief virtue in using subfunctions is that it allows all your code to be split up into small tasks and saved in the same file, with the name of the primary function. It is very useful when solving IVP problems.
By contrast , there is another way to embed functions inside a primary functions — as nested functions. To tell MATLAB a function is nested, it and the primary function must finish with `end` statements ; also the nested function must finish before the primary function i.e. must be enclosed by the primary function.
The main use of nested functions is to allow some sharing of variables across sets of functions without using global variables. The variable scoping rules are quite complex (see MATLAB documentation for details). They are useful for passing parameter values to other functions that contain parameters.

```
nestedTest
```



```
The value of x in primary is 100
The value of x in subfunc1 is  11
The value of x in primary is  11
The value of x in subfunc3 is  33
The value of x in subfunc1 is  11
The value of x in primary is  33
```

You probably won't have any need for nested functions.

Default values for input arguments

MATLAB doesn't have a clean method for providing default values for input arguments so has to resort to counting the number of input arguments, using `nargin`, which provides to the function the number of arguments it was called with. This allows you to do different things depending on how many arguments are provided e.g. use default parameter values if none are given.


```
[root,number] = NewtonFuncDefault(0.2,1e-3);
 fprintf('root = %10.7e after %3d iterations\n',root,number);
 [root,number] = NewtonFuncDefault(0.2,1e-10);
 fprintf('root = %10.7e after %3d iterations\n',root,number);
 [root,number] = NewtonFuncDefault(0.2);
 fprintf('root = %10.7e after %3d iterations\n',root,number);
```

```
root = 2.2361142e+00 after   6 iterations
root = 2.2360680e+00 after   8 iterations
root = 2.2360680e+00 after   8 iterations
```

Many MATLAB functions use this method to provide default values for input arguments.

Variable input argument lists

Some MATLAB functions you meet might use the special cell array `varargin` to cater for input argument lists of unknown length

```
help varargin
```



```
varargin Variable length input argument list.
    Allows any number of arguments to a function.  The variable
    varargin is a cell array containing the optional arguments to th
    function.  varargin must be declared as the last input argument
    and collects all the inputs from that point onwards. In the
    declaration, varargin must be lowercase (i.e., varargin).
        For example, the function,

        function myplot(x,varargin)
        plot(x,varargin{:})
    collects all the inputs starting with the second input into the
    variable "varargin".  MYPLOT uses the comma-separated list synta
    varargin{:} to pass the optional parameters to plot.
```

The call,

```
myplot(sin(0:.1:1),'color',[.5 .7 .3],'linestyle',':')
```

results in varargin being a 1-by-4 cell array containing the values 'color', [.5 .7 .3], 'linestyle', and ':'.

I have never used `varargin` but some MATLAB functions expect you to use it to pass in optional parameters.

Input error checking

You can either check the validity of input arguments by using `if` statements, or by using `assert`


```
function [x,y] = myfun(a,b,c)
assert(isnumeric(a), 'First input must be a number.')
assert(numel(a)==1, 'First input must be a scalar.')
assert(?any(isinf(b)), 'Second input must be finite.')
assert(?any(isnan(b)), 'No NaNs allowed in second input.')
assert(ischar(c), 'Third input must be a string.')
```

If the first expression given to `assert` is not true, an error is thrown with the message and is given as the second argument.

Catching errors

Sometimes you would like the ability to recover from an error in a function and continue with a contingency plan. This can be done using the try/catch construct. For example, the following will continue asking for a statement until you give it one that executes successfully:


```
done = false;
while ~done
  state = input('Enter a valid statement: ','s');
  try
    eval(state);
    done = true;
  catch me
    disp(That was not a valid statement! Look:')
    disp(me.message)
  end
end
```


Within the catch block you can find the most recent error message by inspecting the exception object `me`, as explained in the help pages for `MException`.