Comments on use of MATLAB

Why MATLAB?

- enables user to focus more on the research problem than coding issues
- can go rapidly from idea, to deployable code
- stable, professionally supported platform, well tested functionality
- works well across different operating systems
- easy-to-use debugger, development environment, documentation
- good support for complex numbers
- large base of existing code
- fairly easy to use by colleagues who are domain experts in other fields (i.e. not computer scientists)

Notes on some points

This section is explains a few points that might not be obvious to a new MATLAB user. Many stem from MATLAB evolving from a 'Matrix Laboratory".

Variables, even single number scalars, have a size described as the number of rows x number of columns. So a scalar will be 1x1. A list of numbers (a vector) can be specified as a row or a column. For example 10 numbers could be [10x1] or [1x10], depending on how it is set-up. To reduce confusion, it is usually best to kep vectors as a column. Using the colon `:` will always return a column vector.

```
rowv = [ 1 2 3 4 5 6 7]  % a 1x7 row vector (1 row, 7 columns)
```

```
rowv = 1×7
     1     2     3     4     5     6     7
```

```
colv = [10; 20; 30; 40]  % a 4x1 column vector (4 rows, one column)
```

```
colv = 4×1
    10
    20
    30
    40
```

```
rowv(:)  % colon will give a column output, whatever the input
```

```
ans = 7×1
     1
     2
     3
     4
     5
     6
     7
```

```
colv(:)
```

```
ans = 4×1
```

```
10
20
30
40
```

MATLAB will perform **implicit expansion** if you combine a row vector and column vector. Although useful in some scenarios, it can be confusing.

```
rowv + colv  % result of adding 1x7 vector to 4x1 vector is 4x7 matrix!
```

```
ans = 4×7
    11    12    13    14    15    16    17
    21    22    23    24    25    26    27
    31    32    33    34    35    36    37
    41    42    43    44    45    46    47
```

Be aware that sometimes vectors read in from external data will "appear" as row vectors so you may want to convert to column if you aren't certain. You can use the colon (`:`) to return a column vector. You can also go back and forth between row and column vectors using `transpose`. The shorthand for `transpose` is `.'` but note that this can be hard to see, and sometimes people just use `'` which is actually the transpose and complex conjugate.

```
datav = [ 1 2 3] ;

data_col_vec = datav(:)  % returns column if datav is row or column
```

```
data_col_vec = 3×1
     1
     2
     3
```

```
transpose(datav) % Both these transpose the vector
```

```
ans = 3×1
     1
     2
     3
```

```
datav.'
```

```
ans = 3×1
     1
     2
     3
```

By default, numeric variables are of type `double`, meaning floating point numbers are stored with high precision. For most applications, this accuracy is more than enough and you don't have to worry about numerial

precision. Normally you should not compare two doubles to see if they are equal because small rounding errors can artificially make them appear to be not equal. However you do not need to specify integer types for applications such as loop counters when the values are clearly always integers.

```
trackLength = 399.98 ;

for ilap = 1:10
    disp("Lap number " + ilap)

    if ilap == 7 % This test is OK because ilap can only take integer values
        disp(" Reached lap 7")
    end

    distance = ilap * trackLength ;
    % example of testing equality within a certain tolerance
    if abs(distance - 800) < 0.1  % using an absolute tolerance here of 0.1
        disp(" Passing the 800m mark")
    end

end
```

```
Lap number 1
Lap number 2
 Passing the 800m mark
Lap number 3
Lap number 4
Lap number 5
Lap number 6
Lap number 7
 Reached lap 7
Lap number 8
Lap number 9
Lap number 10
```

You can specify the use of integers (often to save memory for large arrays) and sometimes values read from files "arrive" as integers when you may not realise. Care needs to be taken because MATLAB will return the integer type when you 'combine' them and so **you can loose precision**.

```
pixelValue = int16(42)  % Explicitly of type int16
```

```
pixelValue = int16

   42
```

```
offset = 0.4  % by default, is of type double
```

```
offset =
         0.4
```

```
newValue =  pixelValue + offset  %! newValue is int16 and the answer is
rounded
```

```
newValue = int16

   42
```

```
% You can prevent this by using double to convert from int16
updatedValue = double(pixelValue) + offset
```

```
updatedValue =
         42.4
```

Note in the example before, the lap counter was called `ilap.` The `i` in the variable name implied it should take integer values, but it was always of the default type `double`. This is sometimes called a *flint* a floating point integer.

For functions where the user specifies the size of the output, entering a single number can result in a 2D matrix For example, `zeros(4)` outputs a 4x4 matrix with all entries zero, `randn(3)` outputs a 3x3 matrix of random numbers picked from a normal (Gassian) distribution. I recommend avoiding this syntax and always specify the number of rows and columns e.g. `zeros([4 4])` for a 4x4 matrix.

The indexing of arrays follows the mathematical convention of using row-column order. For example, `g(4,6)` is the entry of `g` on its 4th row and 6th column.

Indexing in MATLAB starts at 1. The top row (first row) of a matrix is represented by the index 1. For example `toprow = mat(1,:).`

In summary

- Sizes are `nrows x ncols.` Indexing starts at 1.
- The default type is `double`. This works fine in most situations.
- Be aware that reading in data from an external source can return a row vector and/or an integer type and future calculations might then have **implicit expansion**, or **return a rounded integer**.
- In 25+ years of using MATLAB, only these two issues have given me numerical problems.

Code Style and Decisions

See also *A concise guide to reproducible MATLAB projects* https://rse.shef.ac.uk/blog/2022-05-05-concise-guide-to-reproducible-matlab/

Here are some of the aspects that came to mind when writing the demo notebooks.

- Prioritise clarity over speed, memory use and compactness.
- Make use of graphs, plots and visualisations.
- Avoid programming statements that do too many things on one line - it should be readable without having to unpack too much in your head.
- Try to give variables names that are meaningful, reflect the quantitiy they represent and avoid confusion with other possible meanings. Prefer shorter names for readability. Avoid double negatives. Use the

shift-enter functionality to change variable names duing editing if a change will help clarity (then check with a search for the old name as it is not changed in comments).