Practical 2: Submit solutions in Matlab Grader before Fri 23rd August at 5pm.

Bring your lecture notes with you for reference.

The aim of this practical is to gain an understanding of modularisation of programs, through using MATLAB `functions`. Included should be an understanding of input and output *arguments* and *scope*. Read the following carefully before coming to the practical.

When writing large programs the code may become extremely difficult to follow. A simple mechanism for improving readability and programming efficiency is to divide the code into self-contained units called functions in MATLAB (in other languages they might be called procedures or subroutines). These ideas are part of a broader concept of *modularity* associated with packages and object-oriented programming, which improves maintainability of code, and facilitates *reuse*. The latter is an important issue: if we are writing code there is an adage: DRY – don't repeat yourself. Functions provide a way to group some commands that you would otherwise have to type out multiple times.

The goals in defining functions are

- *information hiding*: a user should be able to use the function, without knowing how it is implemented, and in fact that might change in the future.
- *clean interfaces:* limit inter-dependencies or *coupling* between components that make programs complex to understand.
- *cohesion:* related processes are grouped.

We specify a function in MATLAB by putting the following in the `function_name.m` file

```
function [out1, out2,...  ,outN] = function_name(inp1, inp2,...  ,inpM)
% comments to be displayed by help
statements
out1 = expression1;
out2 = expression2;
...
outN = expression2;
```

where '`...`' is used here to indicate that there may be an indefinite number of input and output variables (you would not type this). The function would typically be called by typing

```
[o1, o2, ... , oN] = function_name(i1, i2, ... ,iM);
```

The rules that we *must* follow when defining a function are as follows:

1. The function name and the name of the file containing it must be identical except for the `.m` filename extension. Remember MATLAB is case sensitive. It is suggested that you use lower-case for function names as this will increase portability of your code to other operating systems.
2. The function name must follow MATLAB's rules for variable names, as must the input and output variables.

3. If there is only one output variable square brackets are not necessary (even if it is a vector). If there is more than one output variable, the output variables must be separated by commas and enclosed in square brackets (as with a vector). This is true both in the function definition, and when we call a function.

4. If an output is unassigned, this may cause an error, so we need to have at least one statement assigning values to each output variable. Often these statements are at the end of a function file. They don't have to be, but it can improve program readability.

The input variables (`inp1`, `inp2`, ...) and output variables (`out1`, `out2`, ...) are the function's means of communicating with the workspace. The input variables allow you to pass in values to the function, and the output variables allow values to be passed out. There can be some other interactions, for instance if the function reads a file, or presents at plot, but typically these do not pass values in or out of the workspace.

Note that if the function changes the value of any of its input variables the change does not affect the corresponding variable used in the call to the function.

Variables defined inside a MATLAB function have *local scope*. This means that they cannot be seen outside of this function. We cannot obtain their values, or interact with these variables in any way. For instance, they will not appear in the workspace. Even the implicitly created input and output variables cannot be accessed outside the function except at the start (for input variables) and the finish (for output variables).

There are various reasons for this. Firstly, it is highly desirable to minimise the side-effects of functions. This makes program behaviour more predictable. Side-effects can have unanticipated consequences!

Secondly, giving function variables local scope improves the modularity of code. It simplifies the interface between the function and other programs. It makes the interface as simple as calling the function, *e.g.,*

```
[ave,st_dev] = stats(r)
```

The result is that MATLAB functions are easy to use without necessarily examining all of the MATLAB code in the function. We just need to look at the first line of the function to see how to call it. In essence, we can treat a MATLAB function as a black-box that performs an action, and we don't need to understand how it does it.

The third important result of local scope is that we avoid *name collisions* between variables inside and outside the function. Say I want to use a complex function defined by third party. It may define many internal variables. I don't want to have to make sure all of my variables have different names. For instance, they may use the common name `x` inside their code, and I may want to use `x` for something different outside of the code. I don't want to have to check that will be OK. This problem may sound trivial, but imagine I am calling a hundred different functions each with its own variable names. If there was overlap in the namespaces, then not only might my names collide with the functions' variable names, but also the functions might have collisions between each others variables. Local scope of internal variables guarantees that variables can co-exist peacefully within their own functions.

It is possible to create "global" variables that are accessible elsewhere in the MATLAB program. However, this mechanism should very rarely be used for the reasons described above, and so we will not describe how to create such variables here.

Tasks

1. Enter and trial the following example which calculates the mean and the standard deviation of the values in the vector `x`. Open a file called `stats.m` and enter into it
```
function [average, standard_deviation] = stats(x)
% Calculates the mean and standard deviation of
% the data in the vector x.
average = mean(x);
standard_deviation = std(x);
```
We can now test it with some random numbers:
```
r = rand(100,1);
[ave,st_dev] = stats(r)
```
The function will calculate the mean and standard deviation of the input (random) numbers, and will pass these to the output variables `ave` and `st_dev`.

2. Create your own function to calculate $n!$ (*i.e., n* factorial). Make sure that your function checks the inputs are valid (see the notes below on throwing errors). MATLAB has its own function called `factorial` – you can check your code is working by comparing answers. **Finally, login to Matlab Grader and check your function.**

   When you get to Grader, **don't delete the template function.** You you need to use this (and the function name defined there) for testing. Just fill in the missing parts.

3. Try the following: define a function *testing.m* as follows
```
function result = testing(input)
input = 10;
result = input;
```
Now use it
```
x = 1;
y = testing(x);
```
What are the values of x and y at the end, and why?

4. We can call one function from within another. In fact this is common, particularly with respect to MATLAB's built in functions. Define the following in a *.m* file, and test:
```
function result = normal_cdf(input)
% compute the CDF of the normal distribution using MATLAB's
% built in erf function
result = (1 + erf( input / sqrt(2) ))/2;
```

5. Consider Gauss-Jordan elimination: write a function `Mout = pivot(Min,i,j)`, which takes 3 input arguments: a G-J Tableau (which is just an array) M_{in}, and two natural numbers i, and j, and returns the array M_{out} after pivoting M_{in} at the point (i, j).
   - It should check that the right number of inputs are passed to the function (see the notes below on throwing errors).
   - It should check that (i, j) is a valid pivot, and return an error if it is not. For an $n \times m$ array M_{in}, you must have $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, m$, and $m_{ij} \neq 0$.
   - Remember that with floating-point numbers we can't expect a naive check $m_{ij} \neq 0$ to do anything useful. Write your check to see if m_{ij} is close to zero.

   I have put a protected MATLAB function (a `.p` file) on the web page. You can compare your results to this function. Once you have a working function **check your function on Matlab Grader.**

Throwing errors

What do we do when something goes wrong? Suppose you wrote a function to calculate n factorial, but someone inputs a negative number into the function. What would we do then?

The answer is that we *throw an error*, or sometimes we say we throw an exception. The idea is that, instead of returning a set out output argument values, the function tells us that there was an error. We could do this manually, by printing out a string saying error and stopping, but MATLAB has a function to help called, strangely enough `error`. When something goes wrong, we call this function (usually) with a message to describe the error. Often, we use this approach to check that the input arguments are valid at the start of a function. We can also use the implicitly created variable `nargin`, and likewise for outputs with `nargout`.

We can see the result in the function `realsqrt()` written below, which will output the square root of a non-negative number, but will throw an error if you pass it a negative number, or if the wrong number of inputs are given.

```
function [y] = realsqrt(x)
  if x < 0
      error('input x must be >= 0, or realsqrt would produce complex result.');
  end
  y = sqrt(x);
```

You can also use the `assert` function to test for a condition, and throw an error if it is false, all in one command. For instance, we can rewrite the above code more concisely as

```
function [y] = realsqrt(x)
  assert(x >= 0, 'input x must be >= 0, or realsqrt would produce complex result.');
  y = sqrt(x);
```

Note that often, when something is wrong, for instance an array index is outside of the matrix it addresses, then MATLAB will throw an error of its own. It is much better for us, in many cases, to control the errors ourselves, so that we can more easily debug code. So in much of your code you will be expected to throw an error yourself, rather than waiting for MATLAB to do it.

Programming style

Apart from the usual programming style guidelines, there are some additional style considerations when we write MATLAB functions.

Comment lines up to the first non-comment line in a function file will be displayed if `help` is requested for the function name. This allows us to put interesting information about the function in a place that is easily accessible without reading the file itself. It is common to use these lines of comments to define (i) what the function does, (ii) what its inputs and outputs are, and (iii) to provide meta-information such as the author of the function, its version, and the dates it was created and revised.

The first line of a MATLAB function's comment, and its the function name are also used in `lookfor` command. Hence we should (i) choose a meaningful function name, and (ii) carefully select the first line of comments to make this function easy to find.

It is good coding practice to always use semi-colons at the end of lines in MATLAB functions so that the function has no unintended "side-effects" such as printing out intermediate values.

It is also good practice to avoid redefining common functions. This will avoid simple bugs in code where a function gives an unexpected result.

In any programming language, care should be taken to check input arguments carefully! This makes good sense, as a user of a function may not have read the documentation carefully and may make a mistake in calling the function. Also, failing to check validity of input arguments is a major source of security holes in various pieces of computer code. We generally omit such checks in our examples, to keep them short and simple, but they should not be omitted in practice.

Finally, we should (almost) never use global variables.