Introduction to University Mathematics 2018

MATLAB Worksheet V

*Complete the following tasks. Your grade will come from 2 files submitted on Canvas – emac.mlx and graph.mlx, due at **7PM**. They are worth 10 marks each.*

1. Function. Recall the Maclaurin series for e^x,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots = \sum_{k=0}^{\infty} \frac{x^k}{k!}$$

Suppose that for a given x, we want to evaluate the Maclaurin series for e^x up to the term in x^3. Let's create a new MATLAB function, $emac(x)$, such that

$$emac(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!}.$$

To do this, open a new *Function file* (click *New - Live Function*). Study the template carefully, and edit it so that you end up with the following[1]. By the way, MATLAB supports live functions in versions R2018a and above. Previously, there were all plain code functions.

```
1  function  y = emac(x)
2  % emac(x) gives the value of e^x using Maclaurin series up to x^3
3  y=1+x+x^2/2+x^3/6;
4  end
```

Save this file as `emac.mlx` in the MATLAB folder on your G: drive.

Now go back to the Command Window and try calling it, *e.g.* `emac(1)`. You should see a value close to $e = 2.718\ldots$. Try other values.

Here's a line-by-line analysis of the code above.

- Line 1: If you're creating a function, begin with the word `function`. The structure of the first line is

  `function output = name(input)`

  - `output` = the variable(s) that the function produces.
  - `name` = the name of your function (watch out for clashes).
  - `input` = the variable(s) that the user must provide (enclosed in brackets).

  No need for ; at the end of this line.

  Important: The function name must also be the file name. If your function is called `XXX`, then it should be saved as `XXX.mlx`

[1]You could also just start with a blank script file.

- Line 2: Insert comments to help people understand what your function does.
- Lines 3: Calculation of the output, `y`, for a given input `x`. Note that MATLAB does `^` before `/` so there's no need for brackets in this line.
- Line 4: This marks the end of the function.

You might ask: why work with function? why can't we do these calculations in a "script" file like last week? Well, the advantages are:

- Functions help you break up a big task into little tasks, which are easier to organise.

  In real applications, one would write a script file that calls lots of functions in various script/function files. The functions can even call one another.
- Recall that variables in script files are *global*: meaning they can be seen in the *Workspace* window, and so you must keep track of *all* of them to prevent clashes.

  Variables in a function file, however, are *local* to that function, meaning that the variable definitions are only understood within the function.

  For example, see that `x` and `y` do not appear in the *Workspace* window, yet they are defined inside the function. This reduces the risk of variable clashes.

In fact, if you replace all occurrences of `x` in the function by any other valid names, the function would still work in exactly the same way (try it!). You can think of the input and output variables as *dummy* variables – use whatever names you like. This is analogous to saying that

$$f(x) = e^x \qquad f(t) = e^t \qquad f(\heartsuit) = e^{\heartsuit}$$

are all the same function.

▶ **TASK 1**: Modify `emac.mlx` so that the command `emac(x)` returns an estimate of e^x using the Maclaurin series with terms up to x^5 (I'll call this the *5th-order* series). Test it to make sure that it is more accurate than the cubic series.

2. Multi-output. Suppose that you want `emac(x)` to tell you more than just one output. Let's say, apart from the 5th-order Maclaurin approximation, you also want to know how accurate that approximation is.

   You can ask MATLAB to also calculate the error, and stitch both the estimate and the error together as a *paired* output. Here's a rough work scheme.

```
function [y, err]=emac(x)
% emac(x) gives a pair of numbers [y, err] as output
% y = Maclaurin's series for e^x, err = error
y=   .....
err =  .....
end
```

(Obviously you'll need to replace `.....` with your own code.) For the error, you could quote the percentage error from last week, or the *fractional error* given by

$$\text{Fractional error} = \frac{\text{Estimate} - \text{Actual}}{\text{Actual}},$$

which is just the percentage error without the factor of 100.

To call the function `emac` in the Command window, use this kind of command:

```
[est,error] = emac(3)
```

which produces the Maclaurin estimate (which we've named `est`) for e^3 and the error (named `error`). Again these are dummy variable names so use whatever names you like. For example, typing

```
[unicorn, rainbow] = emac(3)
```

would give us exactly the same info[2].

► **TASK 2**: Modify `emac.mlx` so that it gives an output of the form `[y,err]`, where `y` = 5th-order Maclaurin estimate for e^x, and `err` = the fractional error.

If you have performed this task correctly, you should be able to produce the following in your Command window:

```
>> [jack, jill]= emac(3)
jack=
   18.4000
jill=
   -0.0839
```

The negative sign just means that the 5th-order series for e^3 *underestimates* the actual value (by 8.39%).

Note that if you type `emac(1)` now, you will only see the first output, *i.e.* the estimate (this is handy when you don't want to see too much information).

3. Multi-input. Let's now make our function even more sophisticated by making it accept more than one *input* argument. Let's now give the user the choice of how many terms in the series to keep, so that the command

```
emac(x,N)
```

would produce the Maclaurin series up to the term x^N. In other words

$$emac(x,N) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^N}{N!} = \sum_{k=0}^{N} \frac{x^k}{k!}$$

Let's plan step-by-step how to work out this sum (this is similar to what we've done in the past 2 weeks). Suppose for now that `x` is an ordinary number (not an array).

[2]This should remind you of the command `[r,c]=find(M==10)` for finding the (row,column) of element 10 in a matrix from Sheet 3.

(a) create an array `A=[0 1 2 ... N]` (`N` provided by the user).

(b) create an array `B=[x^0/0! x^1/1! x^2/2! ... x^N/N!]` (using element-wise operations - see Sheet 3 for help).

(c) Sum up all the elements of array `B`.

▶ **TASK 3**: Modify `emac.mlx` so that it takes 2 inputs `(x,N)` (where `x` is a number, and `N` is an integer) and produces two outputs `[y,err]`, as described above.

If you have done this correctly, you should be able to produce the following in your Command window:

```
>> [y,err]=emac(3,10)
y =
   20.0797
err =
   -2.9234e-04
```

Now submit `emac.mlx` onto Canvas. Click on the Canvas module site >> Assignments >> Matlab worksheet V – emac.mlx. The submission box closes at 7PM.

---

4. Plot. Let's plot some graphs with MATLAB.

Let's first try to plot $y = 2x + 1$ over the interval $-2 \le x \le 2$. Start a new Live Script and type the commands:

```
x=[-2:2];
y=2*x+1;
plot(x,y)
xlabel('x axis') % Add a label to the x-axis
ylabel('y axis') % Add a label to the y-axis
title('A line segment') % Add a title to the plot
```

You should see a straight line together with the x label, the y label and the plot title. Note that the x-axis ranges from -2 to 2.

☐ Try to plot $y = x^2$ over the same domain.

You should find that `plot(x,x^2)` doesn't quite work. This is because MATLAB needs 2 *arrays* to make a plot – one for x and one for y. The syntax `x^2` doesn't make sense if `x` is an array. Instead, you need the *element-wise* squaring...

You should now see a graph, but it is quite jagged and rough. This is because we only have data points at integer values of `x`. To make the graph smoother, use a finer resolution in the `x` array. For example, try `x = [-2:0.1:2]`.

There is another easy way to get lots of equally spaced points from -2 to 2 using

x = `linspace(-2,2);`

This is a really useful command. Read more about `linspace` in Help. Note that `linspace` produces 100 values by default (it's possible to produce more), so use ; to prevent an explosion of 100 numbers on your screen.

In summary, think of plotting a graph as putting two arrays together. Start with the `x` array. then use element-wise operations to create the `y` array.

▶ **TASK 4**: Plot the graph of the function $y = e^x$, *smoothly* over the interval $-3 \le x \le 3$. Use `linspace`. Show x label, y label and plot title.

5. Multi-plot. Sometimes you might want to plot more than one graph on the same set of axes. For instance, to plot $y = x^2$ and $y = 2x + 1$ over the same range of x, try

```
x=[-2:2];
y1=2*x+1;
y2=x.^2;
plot(x,y1, x,y2)
xlabel('x axis')
ylabel('y axis')
title('A multi-plot')
% Add a legend to distinguish one from the other
legend('A line segment','A quadratic curve', 'Location', 'north')
```

You can add more pairs of x_n, y_n if needed. Type *doc plot* or *help plot* to see how various line types, plot symbols and colors can be obtained with plot(x,y,s) where s is a character string made from some pre-defined characters.

▶ **TASK 5**: Plot all these 3 functions on the same set of axes (with $-3 \le x \le 3$) using different line types, plot symbols and colors showing x label, y label, title and legend.

- $y = 1 + x$,
- $y = 1 + x + \frac{1}{2}x^2$
- $y = e^x$

6. A pretty graph. Finally, let's now try to plot a graph similar to the one shown on the last page.

Create a new function file and type in the following content. Save it as *eloop.mlx* - a file function which includes function code only.

```
function y = eloop(x,N)
% eloop(x,N) gives the value of e^x using Maclaurin series up to x^N
y=0;
    for k=0:N
        y = y + x.^k/factorial(k);
    end
end
```

This code does the same job as the file `emac(x,N)` which you worked on, but it accepts an *array* `x` as well (unlike `emac`). Try calling it in a Live Script or in the Command Window using, say, `eloop([-3:3],5)`. This should give you the 5th-order Maclaurin series for all integer `x` from -3 to 3.

At the heart of `eloop` lies a *"for"* loop (lines 4-6). We will come back to it next week when we look at *control statements*, so you can skip the explanation in the next paragraph if you're not too bothered and simply trust that it works.

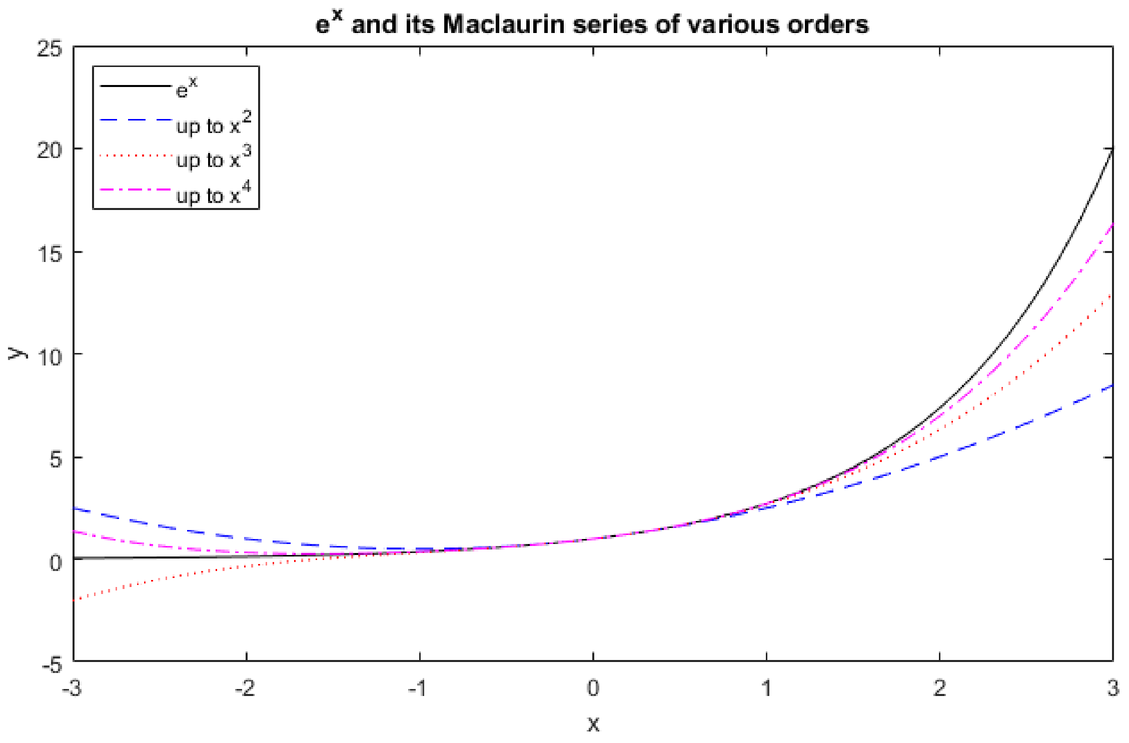
The *for* loop starts with `k=0` on line 4, goes through to line 6, then comes back up and repeats with `k=1`, and so on until `k=N`. With each run of the loop, it collects the term $\frac{x^k}{k!}$ and keeps adding it to the total `y`. Note from Line 3 that `y` is initially assigned to be 0, and it then grows and grows with each cycle of the loop. After $N+1$ cycles of the loop, the total value collected is therefore $\displaystyle\sum_{k=0}^{N} \frac{x^k}{k!}$, which is our desired output.

▶ **TASK 6**: Create a plot similar to the one below. Here's the recipe.

- Firstly, create an x vector using a number of equally spaced points from -3 to 3.
- Secondly, use the function `eloop` to help you get different y-axis vectors.
- Thirdly, use the `plot` command to create a basic multi-plot of 4 graphs. Make sure the curves are plotted in different line types (solid, dashed, dotted, or dashdot...). This is to ensure that we can still tell the curves apart even when the graph is printed in black-and-white.
- Lastly, add the x-axis label, the y-axis label, the graph title and the legend.

Your figure doesn't have to look exactly like mine, but it should be as informative and easy to read. Feel free to be artistic.

> **When you are happy with your graphs, save your *graph.mlx* file and upload it onto Canvas. Click on the *Canvas module site* >> *Assignments* >> *Matlab worksheet V – graph.mlx.* The submission deadline is 7PM today.**


e^x and its Maclaurin series of various orders
e^x
up to x^2
up to x^3
up to x^4
y
x
25
20
15
10
5
0
-5
-3
-2
-1
0
1
2
3