

Introduction to University Mathematics 2018

MATLAB WORKSHEET IV

*Complete the following tasks. No need to submit anything in hard copy. Your grade will come from the file `gregory.mlx` to be submitted on Canvas by **7PM** today.*

Today you will be writing (possibly) your first proper MATLAB script. A script is simply a file with a bunch of commands which can be run consecutively without the need to type in and run line-by-line as we have been doing in the past weeks.

1. Hello world. The first step is to create a MATLAB script by clicking *New >> Live Script*.

The “*Hello world*” script has been a traditional rite of passage for all beginning programmers since the early 70s, and here it is in MATLAB version. In the Live Editor window (not the Command Window), type

```
disp('Hello world!')
```

The `disp` function simply displays whatever is in the quotation marks as texts, or *strings*. Use only single quotation marks - not double. More on `disp` later.

Now save the file as `hello.mlx` in your G: drive (if you’re using a University computer). This file is what I call a *Live Script file*, or *code*. I suggest you create a new folder called MATLAB, and store everything MATLAB-related there (there will be many in the coming weeks). Remember to set this folder as current folder if not yet done.

Warning: Always save your files on the G: drive since this is regularly backed up and secure. Never save your work on the C: drive of the PC you’re using — everything will be wiped out at the end of this learning session!

Alternatively, you can use script *Editor* to write code in a script file (.m file) which was the one to use before the Live Script was first introduced in MATLAB version 2016a.

If you use a .m file for the “Hello World!” code, you need to go to the Command Window, and type `hello`. MATLAB should respond with *Hello world!*.

This shows you how scripts are run by MATLAB: When you type `♥♥♥` in the Command Window, MATLAB looks for a file called `♥♥♥.m` in its path folders and runs all the commands in it.

Using Live Script (.mlx files) is recommended! MATLAB live scripts are interactive documents that combine MATLAB code with *embedded output, formatted text, equations*, and *images* in a single environment called the **Live Editor**.

Search the Documentation for the page **What Is a Live Script or Function?** to see the differences between *Live Scripts* and *Plain Code Scripts*. In fact, they may be able to get converted to each other.

2. Input-Output. Let's code up something a bit more useful. Let's say we want to write a script which calculates the area of a circle of a given radius.

Close `hello.mlx` and create a new file called `circle.mlx`. Always make sure the name of your Live Script file does not clash with any common MATLAB functions (see Sheet 2).

In `circle.mlx`, first add a headline **Calculating the area of a circle of a given radius** (*INSERT >> Text*); then insert an equation $A = \pi r^2$ (*INSERT >> Equation >> LaTeX equation*); finally, type the four lines in the box below.

```
1 % This script calculates the area of a circle of a given radius
2 r = input('Enter radius : ');
3 area = pi*r^2;
4 disp(area)
```

Now try running the script by pressing **Ctrl-Enter** a few times with different inputs (make sure the `circle.mlx` Live Script is selected each time). Another way to run the code is to click on the big green RUN button.

Here's a line-by-line analysis of the script.

- Line 1: The symbol `%` signifies a comment. MATLAB ignores everything that comes after this sign (until you start a new line). This is useful to remind yourself what the code does (you can put comments on any line in the code). It's also helpful for other people who will be using or marking your code.

Follow the following activity, and try to understand the results.

- ☐ Put some comments at the end of other lines of the code and run it again. (You shouldn't see any comments when the code is run.)

- Line 2: Here we define a new variable `r`, which will be obtained from the user's input (via the MATLAB function `input`). Include your user prompt in single quotation marks.

- ☐ Try omitting the semi-colon at the end of line 2 and run the code again. Can you explain the difference?

Always make sure that your code does not display unnecessary numbers or texts that may clutter the screen, possibly putting the user in a bad mood (especially when working with arrays of a huge size).

- Line 3: A new variable called `area` is assigned the value $\pi \times r^2$.

- Line 4: The numerical value of `area` is displayed to the user.

- ☐ Try `disp('area')` instead of `disp(area)`.

Can you see what the quotation marks do?

Now some exercises.

- ☐ Start a new section in `circle.mlx` by clicking *Section Break*, insert a new headline `Calculating the circumference of a circle of a given radius`, insert a new equation $C = 2\pi r$, and type a few lines to calculate and display the *circumference* of a circle of a given radius.
- ☐ Run your new section of code to find the circumference of a circle with the following radii: a) 0.5 b) $\sqrt{2}$
(Ans: a) 3.1416, b) 8.8858)

3. `fprintf`. It would be nice to have a more informative output from `circle.mlx` – say, something like

The circumference of a circle radius 0.5 is 3.1416

There is a way to make `disp` do this, but it's a rather strange procedure which is not very flexible (e.g. you can't control how many decimal places to display).

A more useful method to create a nice output is to use the `fprintf` function, which has its origin in C programming (the same technique described here applies to C and Fortran programming).

Let's test it out by working in a new section. First, define the variable `p = 24`. To display

The value of p is 24

Type the following command directly in the Command Window:

`fprintf('The value of p is %d \n', p)`

Here `%d` is called a *conversion character*. It is just a place-holder, which is substituted by the required numbers/variables listed at the end of the statement (after a comma).

Now work through the following checklist.

- ☐ Redefine `p` to be any integer you like, and run the same `fprintf` command above again. Do you see your new value of `p` displayed?
- ☐ Define another variable `q = 6` and use `fprintf` to display

The value of `p+q` is ♡

where ♡ is some integer which MATLAB should automatically calculate. Change the value of `q` and check that MATLAB still displays the correct value of `p+q`.

- ☐ Delete `\n` and run the same `fprintf` command again.
Do you see what `\n` does?
Forgetting to use `\n` will leave the user with an annoyingly messy screen, especially when several `fprintf` commands are used.

Conversion characters like `%d` must carry information on the type of number/thing to be displayed. Some examples¹ are given in the table on the next page.

Study the table carefully.

<code>%d</code>	Integer (<code>d</code> specifies that it is a base-10 ('decimal') integer).
<code>%f</code>	Float (double-precision number). Think of a float as a number with many decimal places.
<code>%6f</code>	Float with 6 digits.
<code>%.4f</code>	Float with 4 decimal places.
<code>%6.4f</code>	Float with 6 digits, including 4 decimal places.
<code>%e</code>	Scientific notation (e.g. $2.4\text{e-}20 = 2.4 \times 10^{-20}$).
<code>%g</code>	<code>%f</code> or <code>%e</code> , whichever is more compact.
<code>%s</code>	String (letters and characters).
<code>\n</code>	New line.
<code>%%</code>	Percent character <code>%</code> .

Here is an example involving two types of conversion characters.

Let's define `N=10` and `x=sqrt(N)`. To display

The square root of 10 is 3.162278.

Use the command

`fprintf('The square root of %d is %f. \n', N, x)`

Now consult the Table and modify the above `fprintf` command to produce each of these two lines.

- ☐ The square root of 10 is 3.16.
- ☐ The square root of 10 is 3.16 and by the way, `p=♡`, `q=♡`.
[where the ♡ are your previously defined values of `p` and `q`.]
- ☐ Redefine variables `p`, `q`, `N` to be whatever new integers you like. Define `x=sqrt(N)` again, and rerun the last `fprintf` command.
Did MATLAB display the updated values of all your variables?
- ☐ Go back to the section where your code of circumference is and "comment out" the last line (by putting `%` in front of `disp`). Insert a new display command using `fprintf` instead (with appropriate conversion characters).
If done correctly, your code should be able to display something like:

The circumference of a circle radius 0.5 is 3.1416

Warning 1: Don't delete codes unnecessarily. It's better to comment out unwanted lines using `%` just in case they are needed later.

Warning 2: Using the wrong conversion character can give nonsense/nonexistent output.

You are now ready for today's assignment!

¹For the full list of conversion characters and other special characters, search the help documentation for **formatting strings**.

4. Today's assignment

This task concerns the famous *Gregory series* for π (discovered by the 17th-century Scottish mathematician James Gregory)

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots \right) = 4 \sum_{n=1}^{\infty} \frac{(-1)^{n-1}}{2n-1}.$$

Your task is to write a new Live Script file, `gregory.mlx`, which calculates this Gregory series using a given number of terms (specified by the user). This can be regarded as a way of estimating the value of π .

Study last week's worksheet to help you with the series calculation.

Your code, when run, should do the following:

- ☐ Ask the user how many terms in the series are required.
- ☐ Tell the user the value of the Gregory series, giving at least 10 decimal places.
- ☐ Tell the user how accurate the series is as an estimate for π . You can do this by displaying the *percentage error* given by the formula:

$$\text{Percentage error} = \frac{\text{estimate} - \pi}{\pi} \times 100 \%$$

A screenshot of a successful code is given overleaf. Your code should do something very similar (but not necessarily identical) to the screenshot.

Tips:

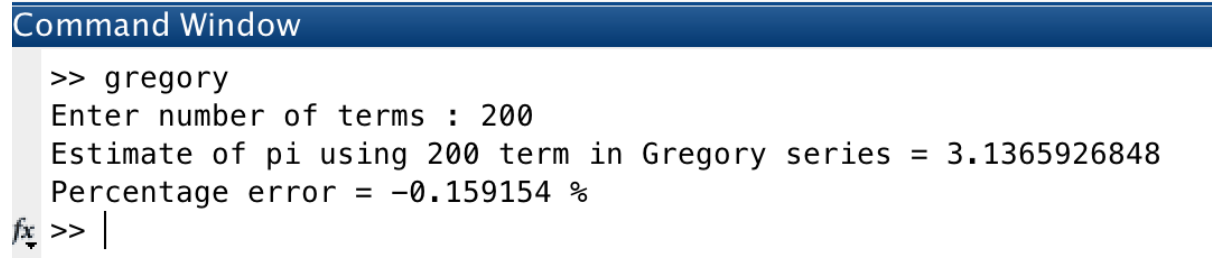
- ☐ Use `fprintf`, not `disp`.
- ☐ Include some comments in the code to make sure anyone (including the marker) can make sense of your code, not just you.
- ☐ Suppress unnecessary results from being displayed after you think they've worked. Make sure that the user's screen is not messed up after running your code. However, during coding, displaying some results of intermediate variables and/or functions could be used as a way to visually debug if necessary.

Upload your `gregory.mlx` file onto Canvas (go to *400296* homepage and click *Assignments*). The submission box will close at **7PM** today.

Your code must not produce an error when run. Codes that produce errors will score at most 40%.

Before you leave, make sure (please tick):

- ☐ You have saved all your Live Script files on the network G: drive (or somewhere else safe).
- ☐ You have checked that your file is definitely, definitely, *definitely* on Canvas.



```
Command Window
>> gregory
Enter number of terms : 200
Estimate of pi using 200 term in Gregory series = 3.1365926848
Percentage error = -0.159154 %
fx >> |
```

Figure 1: A screenshot of what should happen when `gregory.mlx` is run.