

Name: _____

Introduction to University Mathematics 2016-17
MATLAB WORKSHEET I

Complete the following tasks and hand in before the end of the lab session.

1. Basic calculator. MATLAB can be used as a basic calculator. For example, to evaluate

$$5 \times \left(2.4 - 1.3 + \frac{3}{8} \right)$$

type `5*(2.4 - 1.3 + 3/8)` in the command window and press enter (try it).

You should get 7.3750. Note that spaces are ignored by MATLAB, but they can help make complicated codes more readable. Also note that division takes precedence over $+$ and $-$ in the brackets, so there is no need to type `+(3/8)`. However, use brackets if you're unsure what MATLAB might do first.

Evaluate the following in a single line in MATLAB. Use brackets wisely.

$$\left(9.7 - \frac{2}{15} \right) \times \frac{19}{8} \quad \text{Ans} = \underline{\hspace{2cm}}$$

$$\frac{2 \times (3.142 - 1.75)}{\frac{7}{12} + 2.15} + \frac{5}{12} \quad \text{Ans} = \underline{\hspace{2cm}}$$

$$1 + \frac{2}{3 + \frac{4}{5 + \frac{6}{7}}} \quad \text{Ans} = \underline{\hspace{2cm}}$$

2. Format. By default, MATLAB does all calculations in the so-called *double-precision* system. In this system, every real number is stored as a sequence of 64 binary numbers (zeroes and ones). This is roughly equivalent to 15 or 16 significant digits.

But we don't often need to see so many digits - sometimes a few decimal places will do. It would be nice to be able to control the format of the output we'd like to see.

Try experimenting with more calculations. How many decimal places is displayed in the default format? Ans: _____

Suppose we want to see more decimal places, or work with the so-called *scientific* format (*e.g.* 2.1×10^{50}). How do we change the display format?

There's a search box in the top right-hand box on the screen. Search for "format" and read the first article in the list. (We'll be using this search box a lot).

(Alternatively, you can also type `doc format` in the command window.)

Experiment with different output formats and answer the following questions.

- (a) Evaluate $1/701$, giving your answer in the specified formats:

LONG Ans =

SHORT Ans =

SHORTE Ans =

- (b) What format option will produce the following output? Ans:_____

```
>> 5/16+2/7
```

```
ans =
```

```
67/112
```

Now change the format back to `short` (though you might need to change it later).

At this point, your screen might be a bit cluttered. You might like to start with a clean screen by typing `clc`.

If you'd like to recycle previous commands, press the *up arrow* and choose which previous commands you'd like to reuse. Try it. This will save you a lot of time.

3. Constants. MATLAB has a stock of constants, which include the following.

Command	Meaning	Note
<code>pi</code>	π	Dr Prochno will go over complex numbers, but look them up if you haven't seen them before. Engineers use j instead of i . Try <code>i*j</code> . You should never have to type this Typical size of error when the computer rounds something off Something undefined - something has gone wrong in your code
<code>i</code>	$i = \sqrt{-1}$	
<code>j</code>	$\sqrt{-1}$	
<code>inf</code>	∞	
<code>eps</code>	'machine epsilon'	
<code>NaN</code>	Not a Number	

Note that these commands are *reserved* names: we must try not to create variables that have these names, or there will be chaos! More on this in the coming weeks.

- (a) Find the numerical value of the machine epsilon. Ans: _____
- (b) Write down an operation which produces `Inf`. Ans: _____
- (c) Write down an operation which produces `NaN` Ans: _____
- (d) Evaluate $\frac{1-3i}{4+i}$. Give your answer to 4 decimal places.

Ans:_____

- (e) Use MATLAB to find a rational approximation of π . Explain how you did this.
To how many decimal places is this approximation accurate?

- (f) MATLAB does not store a value for $e = 2.718\dots$. Instead, MATLAB has the function `exp(x)` which calculates e^x .

Check that `exp(0)` and `exp(1)` are what you expect. Many students make the mistake of typing `e^x` instead of `exp(x)`. But MATLAB doesn't understand what this means (try it). The letter `e` in MATLAB is reserved for scientific notation (e.g. 3.5×10^{-12} , see the list below)

Write down the value of $1/e^4$ (to 4 decimal places). Ans: _____

4. Scientific calculator. Here are examples of some maths operations and their commands. Like all MATLAB functions and variables, these *are* case-sensitive. Study each entry below very carefully.

2^{12}	<code>2^12</code>	$3^{-1/3}$	<code>3^(-1/3)</code>
3.5×10^{-12}	<code>3.5e-12</code>	$\sqrt{2}$	<code>sqrt(2)</code>
$\sin x$	<code>sin(x)</code>	$\operatorname{cosec} x$	<code>csc(x)</code>
$\cos x$	<code>cos(x)</code>	$\sec x$	<code>sec(x)</code>
$\tan x$	<code>tan(x)</code>	$\cot x$	<code>cot(x)</code>
e^x	<code>exp(x)</code>	$\ln x$	<code>log(x)</code>
$ x $	<code>abs(x)</code>	$n!$	<code>factorial(n)</code>

Note that all angles in the trigonometric functions are assumed to be in radian.

- (a) Use MATLAB to evaluate the following. Leave answers to 4 dec. pl. where needed.

$$\tan(5.3 \times 10^{50}) =$$

$$\sin i =$$

$$2^4 =$$

(express your answer as $\square \times 10^\square$)

$$(2^4)^3 =$$

(should get a **different** answer)

$$4^{5/2} =$$

$$4^5/2 =$$

$$i^i =$$

(surprising - if you know about complex numbers)

$$\sqrt{2 + \sqrt{2 + \sqrt{2 + \sqrt{2}}}} =$$

$$\sqrt[3]{328509} =$$

$$\ln \cos 10^\circ =$$

(should get a real answer)

$$\log_{10}(-\pi) =$$

(try `help log`)

$$\log_2(\text{machine epsilon}) =$$

Considering the last part, deduce the exact value of machine epsilon. _____

(b) Calculate $\sin 0$ and $\sin \pi$ on MATLAB. Why do you think we get different answers?

(c) The value of the factorial, $n!$, grows notoriously quickly. Read the help document on the factorial.

Beyond a certain integer, n , MATLAB is no longer able to calculate the exact value of $n!$ (and can only give an approximation). This is because the result requires more than 64 binary numbers to store exactly.

What is the largest value of n such that $n!$ is exact in MATLAB?

ANS: $n =$ _____

Beyond another integer, N , MATLAB cannot even give an approximation for $N!$ as it's too large for MATLAB to handle in double precision (and MATLAB reports an **Inf** instead). What is the largest value of N such that $N!$ can be approximated by MATLAB?

ANS: $N =$ _____

(d) Use MATLAB to evaluate the binomial coefficient $\binom{60}{50}$. How did you do it?

5. Help! Explore MATLAB's help documents and explain how the following questions can be solved in MATLAB. In each case, **write down the command used**, and not just the answer. Leave answers to 4 decimal places where necessary. The first part has been done for you.

(a) Calculate $\sinh(1.5 \times 10^{-5})$.

Command: `sinh(1.5e-5)`. Answer = 1.5000×10^{-5}

(b) Calculate $\sin(10^\circ)$ without converting 10° to radian first.

(c) Calculate $\tan^{-1}(0.5)$.

(d) Find the remainder when 5×10^{30} is divided by 299.

(e) Is 1234567 a prime number?

(f) How many prime numbers there are between 1000 and 5000?

Name:

Introduction to University Mathematics 2016-17

MATLAB WORKSHEET II

Complete the following tasks and hand in before the end of the lab session.

1. Variables You can create new variables in MATLAB by simply using the equal sign. If you type

`x=6`

MATLAB creates a new variable called x , which at the moment has been given value 6 (typically you would then put x through a whole load of mathematical operations). You should see that `x` now appears in the *Workspace* window. The name (and value) of every variable you create will appear in this window.

You are now free to manipulate x (e.g. try `x^2` or `3*x+5`). No surprises there.

But try this

`x=x+1`

Of course this is a nonsensical *mathematical* statement, but in programming, it *does* make sense, and actually reads more like

“new x = old x + 1”

And you can now see that x is now assigned the value 7.

In programming, you should always take the sign `=` to mean an *assignment* rather than an equality (I find it useful to think of the statement above as $x \rightarrow x + 1$).

If you understand MATLAB's `=` as the assignment symbol, you can see it's ok to type `x=2`, but not `2=x`, since you can't *assign* 2 any other value (try it and see).

Here are some exercises with variables. Let's start again with a clean slate by typing `clear`. This wipes out all the variables you created. Careful not to get this confused with `clc`, which just clears the command window.

- (a) Write down the MATLAB command needed in each step:

- i) create x , which can be any integer you like, _____
- ii) create y , which is the year you were born, multiplied by i , _____
- iii) create z , which is the sum of x^2 and y^2 , _____
- iv) replace z with the value of $4\sqrt{z}$. _____

Write down the value of $\ln z$ _____ (4 decimal places).

- (b) Search the MATLAB documentation (search box on top-right) for an article called *“variable names”*. Read the article and experiment. Explore even the *“See also”* suggestions. Then answer the following questions.

- i) The maximum number of characters allowed in a variable name is _____.
- ii) Does MATLAB treat `myvar` and `MyVar` as the same variable?

iii) Are `2day` and `_xyz` valid variable names? Explain.

iv) Note that MATLAB allows you create a variable name which clashes with one of MATLAB's stored constants (*e.g.* `i=2`) or function (*e.g.* `sin=3`), although you may suffer some nasty consequences.

If you have accidentally created, say, `sin = 3`, how would you undo this error?

Are the names `cat` and `dog` free from such a clash? How can you tell?

v) There are some highly reserved “*keywords*” that MATLAB forbids you to use as variable names. List 3 such names.

2. Types. Every variable has a *type* or *class* associated with it.

Real numbers with decimal points (also called *floating points* or simply *floats* – keep this word in mind) are probably the most interesting type of numbers for numerical problems. They come in two flavours: *single* or *double* precision, depending on how much memory you are willing to use to store them. By default, MATLAB stores all numbers as *double-precision floats* (“*doubles*” in short).

i) Computer memory is commonly measured in *bit* or *byte*. Explain what they mean.

ii) How much memory is needed to store a variable of type `double`? _____

We will explore many other types of variables as we go along.

3. Logical. Another important type of variable in programming is the *logical* type, with only two members: `True` or `False`.

Sometimes 1 and 0 are used to denote true or false, but which is which?

Ans:

Hint: For example, ask MATLAB if $6 < 7$.

Here are a few MATLAB logical tests that produce a true/false answer, together with their mathematical meaning. Test them out if you like.

<code>==</code>	test for equality	<code>~=</code>	test for \neq
<code><</code>	test for $<$	<code><=</code>	test for \leq
<code>></code>	test for $>$	<code>>=</code>	test for \geq
<code>&</code>	AND	<code> </code>	OR
<code>~</code>	NOT		

In programming, inequality signs are *logical tests* (`6<7` reads “Is 6 less than 7?”). They do not convey facts (mathematically, $6 < 7$ reads “6 *is* less than 7”).

Hence, it also makes sense to type `7<6` (try it and see).

Here AND, OR, NOT are the *logical* (or *Boolean*) operators which work exactly as you think they should. For instance,

`true&false = false` `true|false = true` `~(false) = true`

Try typing the LHS of these statements into MATLAB (try the 0,1 notation also).

- (a) Complete the following truth tables for the AND and OR operators.

<code>&</code>	1	0		<code> </code>	1	0
1				1		
0				0		

- (b) What is the difference between the following commands?

`pi==3` and `pi=3`

- (c) Now create variables `u = 3` and `v = -1` (let’s also `clear pi` just in case).

In each of the following cases, explain what the command means, and give the answer you expect (confirming your answers with MATLAB).

The first one has been done for you.

- i. `u>=v+4`

Ans: This tests whether $u \geq v + 4$, which is TRUE since $3 \geq 3$.

- ii. `~(u==u)`

- iii. `(u<10)&(pi<1)`

- iv. `(v~=1)|(u==1)`

(d) (A conundrum) This is an extract from my MATLAB screen:

```
>>x==x
ans =
    0
```

What value did I assign to `x`? (or has my MATLAB gone mad?)

(Before we move on, try typing `whos`)

4. Arrays.

MATLAB stands for _____. Indeed, Vectors and matrices are at the heart of MATLAB, and much of MATLAB's unique capabilities are due to its vector/matrix based computation. Even problems that don't seem to involve vectors are often more *efficiently* solved in MATLAB when they are rephrased in terms of vectors.

In computing, a vector (*i.e.* a list of numbers) is also called an *array*.

(a) Creating arrays. Let's first look at how to create arrays. Try typing

```
A=[1 2 4 9]
```

This creates a horizontal array (row vector) called `A`. Note that spaces are used to separate entries in the array (you can also use commas). The most important thing to note here is the use of **square brackets**, which tells MATLAB that you're creating an array.

Here are other quick ways to create arrays. Experiment and fill in the blanks. The first row has been done for you. Try to make sense of the syntax in each case.

Commands	Array created
[1:5]	[1 2 3 4 5]
[3:6]	
[1:2:9]	
[1:2:10.3]	
[4:-1:0]	
[3:2]	

(b) Use the colon notation to generate the array `myvec=[-1 -1.5 -2 -2.5 -3]`

Ans: _____

How much memory does it take to store `myvec`? Ans: _____

- (c) Square VS Round. Square brackets are used to create an entire array, whereas round brackets are used to denote specific *elements* of the array. For example, if we create

$$M = [0 \text{ sqrt}(3) \text{ -48 pi}]$$

Then $M(1)=0$ (its first element), and $M(3) = \underline{\hspace{2cm}}$.

Warning: Many programming errors occur when you mix up these two types of brackets.

Round brackets can also be used to modify specific elements of the array. Explain what each of these commands does to M .

$$\text{a) } M(3) = 0 \qquad \text{b) } M(4) = 2*M(4) \qquad \text{c) } M(2) = M(1)$$

- (d) Expand and contract. Continuing with the array M . Explain carefully what happens when you type each of the following commands

$$M(5)=1$$

$$M(16)=9$$

$$M(4:8)$$

Note that the last command does not change M . At this point your array M should contain 16 elements.

What single command can we use to shrink M by discarding all the even entries? (i.e. replace M by a smaller version of itself, keeping only $M(1)$, $M(3)$, $M(5)$ etc.)

- (e) Column vectors. There are two ways to create a column vector.

i) Separate array entries using a semicolon ; (rather than spaces). Try creating

$$B=[1;2;3;4;5;6]$$

ii) Use the apostrophe ' to *transpose* a row vector.

$$\text{What command will create the column vector } B = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} \text{ using method ii)?}$$

5. Matrices. We can think matrices as layers of row vectors, each layer separated by ; as in the previous activity. For example, try

`mat = [3 4 5; 0 9 8]`

We say that `mat` is a matrix of dimension 2×3 . Again note the square brackets. Round brackets can be used to pick out or modify elements as we did with vectors, but now elements are located using 2 numbers, (`row,column`).

- (a) Explain what the command `mat(1,3)` shows you.
- (b) Explain what the command `mat(:,2)` shows you.
- (c) How can you add `[0 3 6]` to `mat` as its third row ? (preferably without typing out `mat` all over again.)
- (d) How can you swap the rows and columns of `mat` (keeping the result as `mat`)?
- (e) What commands can be used to quickly generate these matrices? (use a single command in each case.)

$$X = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad Z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Next week: matrix functions and operations. I will assume you know basic vector and matrix operations.

Name:

Introduction to University Mathematics 2016-17
MATLAB WORKSHEET III

Complete the following tasks and hand in before the end of the lab session.

1. All about arrays. Let's use MATLAB to play with vectors and matrices (all considered *arrays* in MATLAB). In your command window, create the following row vector v and matrix M (see Sheet 2 if you've forgotten how).

$$v = \begin{pmatrix} 1 \\ 0 \\ 3 \\ -1 \end{pmatrix} \quad M = \begin{pmatrix} 1 & 0 & 3 & -1 \\ 0 & 2 & 10 & 5 \\ -4 & 1 & 1 & 5 \end{pmatrix}$$

Experiment with the commands below, *e.g.* by typing `size(v)` and `size(M)`, and describe precisely what each command does to vectors and matrices in general. The first row has been done for you. Feel free to experiment with your own arrays.

Command	effect on a vector	effect on a matrix
<code>size</code>	gives the dimension of the vector in the form [#rows, #columns].	gives the dimension of the matrix in the form [#rows, #columns].
<code>length</code>		
<code>numel</code>		
<code>max</code>		
<code>sum</code>		
<code>prod</code>		

i) Devise a command which picks out the *overall maximum* of M (*i.e.* 10 in this case).

Ans:

ii) Explain precisely what these commands do

<code>v(end)</code>	
<code>P = rot90(M)</code>	
<code>Q = repmat(M,2)</code>	
<code>[r,c] = find(M==10)</code>	

iii) Write down a line of command which will quickly generate a matrix A which consists of 20 rows of the array $(1 \ 3 \ 5 \ 7 \ 9)$.

2. Magic squares. Generate a 3×3 matrix called `mat`, with the command

`mat=magic(3)`

The matrix represents a 3×3 *magic square*, in which the sum of elements in any row, column or diagonal is the same. Let's call this sum the *magic total*. For example, the magic total for `magic(3)` is 15.

Use the commands in the previous questions to answer the following questions.

i) What commands can be used to verify that the rows and columns of `mat` all add up to the magic total?

ii) The *trace* of a matrix is defined as the sum of the entries along its main diagonal (*i.e.* top left to bottom right).

Write down two lines of commands which can be used to verify that the 2 diagonals of `mat` add up to the magic total?

iii) Calculate the magic total for a 51×51 magic square.

Command:

Numerical answer:

iv) In which row/column does the entry 999 appear in the 100×100 magic square? Give the command used (avoid any low-tech method).

3. Element-wise operations. We will often need to perform operations on entire arrays of numbers, element by element.

For example, let's start with the matrix `mat=[4:6 ; 3:-1:1]` (see last week's sheet if this command doesn't make sense).

a) To multiply every element of `mat` by 2, we type _____. Easy enough. Similar for division. Describe what the following commands do.

`mat+10` `sqrt(mat)`

b) Using ideas from last week, or otherwise, write down two ways in which you can quickly create a 15×20 matrix consisting entirely of numbers 9.

c) Now let's suppose we want to square every element in the matrix, so that the desired result is

$$\begin{pmatrix} 16 & 25 & 36 \\ 9 & 4 & 1 \end{pmatrix}$$

The command `mat^2` will produce an error because, as you may know, a 2-by-3 matrix cannot be multiplied to another 2-by-3 matrix.

Instead, to perform an *element-wise* operation, simply place *a dot* in front of the operation. In this case, the command

`mat.^2`

produces the desired result (try it).

i) What command can be used to produce a matrix consisting of the reciprocals of the elements of `mat`?

ii) In fact, we can use the `.*` operation to multiply (or `./` to divide) two matrices element-wise. For instance, you can check that

`mat.*mat`

gives exactly the same result as `mat.^2`. Now fill in the blanks below.

$$\text{mat.} * \begin{pmatrix} 0 & -1 & -2 \\ 1 & 0.5 & 10 \end{pmatrix} = \begin{pmatrix} & & \\ & & \end{pmatrix}$$

$$\text{mat.} * \begin{pmatrix} & & \\ & & \end{pmatrix} = \begin{pmatrix} 7 & 8 & 9 \\ 9 & 8 & 7 \end{pmatrix}$$

Warning: Always use `/` and `./` for division. There is also the backslash `\`, but you will only need it in when solving linear systems. Don't confuse the two slashes.

4. Series calculation. Here is an example of how you can use the element-wise operations in MATLAB to compute series. Let's suppose we want to evaluate the series

$$1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \dots + \frac{1}{100^2} \quad (\star)$$

You could do the following. Write down the command needed in each step.

- Create an array `A=[1 2 3 ... 100]`
(warning: don't type this line in literally).

As the output is going to create a huge mess on your screen, *suppress the output* with a `[a semicolon]`; at the end of the command. Even though nothing seems to happen, you can check the *Workspace* window that `A` has been created.

Command: _____

- Create an array `B` containing the elements of `A` squared; `B=[1 4 9 ... 10000]`
(suppress output).

Command: _____

- Create an array `C` containing the reciprocals; `C=[1 1/4 1/9 ... 1/10000]`
(suppress output).

Command: _____

- Sum all elements in `C`. Obviously don't suppress the output this time.

Command and answer: _____

Write down a *single* line that combines all the previous commands, giving you the final answer for the series (\star) right away without using predefined variables.

Command: _____

Use the array techniques you learnt above (and playing around with arrays), answer the following questions.

- (a) Write down a single line of command that will evaluate the following series.

$$1 + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \dots + \frac{1}{101}$$

Command:

Numerical answer =

- (b) Write down the series that is being evaluated in this line of command. Use the summation sign \sum .

```
sum(exp([1:20]).factorial([1:20]))
```

- (c) Let's try to evaluate the series

$$1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \dots - \frac{1}{1000}.$$

Try following these steps (consult last week's sheet if necessary).

- First create an array `A=[1 1 1 1 ...]` (1000 entries).
- Replace the even entries of `A` by `-1`.
- Divide `A` by the array `[1 2 3 ...1000]` (element-wise).
- Sum the result.

Write down the commands used below (suppress output where appropriate):

Numerical answer =

- (d) Write down another set of commands to calculate the series in part (c), but this time, instead of using the steps above, use this idea:

series = (positive terms) + (negative terms)

5. Error and accuracy. Consider the series (\star) in Question 4 (page 4) again. In second year, you will be able to show (using *Fourier series*) that when we include sufficiently many terms, this series will approach a surprising number. In symbols, we have:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6},$$

where equality holds only if *infinitely* many terms are summed up (similar to the “*sum to infinity*” for geometric series).

Of course, computers cannot add infinitely many terms together. However, using a sufficiently large number of terms (say, 100 or more), the result should be pretty close to $\pi^2/6$.

Suppose we want to know how accurate is the 100-term series in estimating the above infinite sum. One way to quantify the error is to use the formula

$$\text{Percentage error} = \left(\frac{\text{Estimate} - \text{Actual}}{\text{Actual}} \right) \times 100\%$$

where “Estimate” is the 100-term series, and “Actual” is the theoretical value of the infinite series.

The percentage error could be negative – it just means that your estimate is less than the actual answer (an “underestimate”).

What is the percentage error when we use 100 terms to estimate the infinite series? Write down all the commands used.

Ans: Percentage error = _____% (to 4 significant figures)

Does the percentage error become smaller or larger *in magnitude* when more terms are included?

Ans: _____

NEXT WEEK: writing your first proper MATLAB code (M-file).

Introduction to University Mathematics 2016-17

MATLAB WORKSHEET IV

Complete the following tasks. No need to submit anything in hardcopy. Your grade will come from the file *gregory.m* to be submitted on Canvas by **3PM** 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 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 >> 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 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.m` in your G: drive (if you're not using your own laptop). I suggest you create a new folder called MATLAB, and store everything MATLAB-related there (there will be many in the coming weeks). This file is what I'll call an *M file*, or *script* or *code*.

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 the day!

Now go back 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.

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.m` and create a new file called `circle.m`. Always make sure the name of your M-file does not clash with any common MATLAB functions (see Sheet 2).

In `circle.m`, 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 in the command window a few times with different inputs. (Another way to run the code is to click on the big 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.

Go through the following activity checklist, 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.)
- ☐ Type `help circle`.
- 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 big arrays).
- 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.

- ☐ Modify `circle.m` so that it now calculates and displays the *circumference* of a circle of a given radius.
- ☐ Run your scripts 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.m` – 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 directly in the Command Window. 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.

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

<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 your `circle.m` 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!

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 an M-file, **gregory.m**, 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. Make sure that the user's screen is not messed up after running your code.

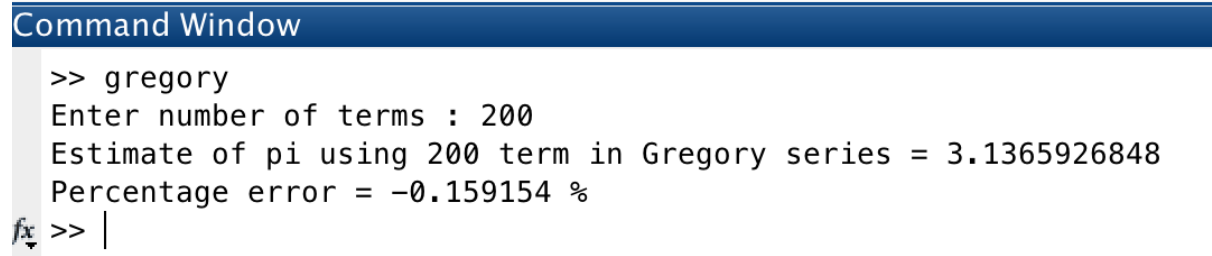
Upload your M-file onto Canvas (go to *32129* homepage and click *Assignments*). The submission box will close at **3PM** today.

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

EXTRA CREDITS (OPTIONAL): Repeat the task with another interesting infinite series or product expression for π . Submit it as a separate M-file.

Before you leave, make sure (please tick):

- ☐ You have saved all your M-files on the network G: drive (or your own laptop).
- ☐ You have checked that your file is definitely, definitely, *definitely* on Canvas.

A screenshot of the MATLAB Command Window. The window has a dark blue title bar with the text "Command Window" in white. The main area is white and contains the following text: ">> gregory", "Enter number of terms : 200", "Estimate of pi using 200 term in Gregory series = 3.1365926848", and "Percentage error = -0.159154 %". On the left side of the main area, there is a vertical grey bar with a small icon of a document with a red 'x' and a downward arrow, followed by the text ">> |".

```
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.m` is run.

Introduction to University Mathematics 2016-17

MATLAB WORKSHEET V

Complete the following tasks. Your grade will come from 2 files submitted on Canvas – a file *emacs.m* due at **1PM**, and a graph, due at **3PM**. They are worth 10 marks each.

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

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

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, *emacs(x)*, such that

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

To do this, open a new *Function M-file* (click *New - Function*). Study the template carefully, and edit it so that you end up with the following¹.

```
1 function y = emacs(x)
2 % emacs(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 **emacs.m** in the MATLAB folder on your G: drive.

Now go back to the Command Window and try calling it, *e.g.* **emacs(1)**. You should see a value close to $e = 2.718\dots$ 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** (duh..). 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.m**

- 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 \wedge before $/$ so there's no need for brackets in this line.

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

- 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 M-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

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

are all the same function.

► **TASK 1:** Modify `emac.m` 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 info².

► **TASK 2:** Modify `emac.m` 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 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

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

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).

²This 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 $A=[0 \ 1 \ 2 \ \dots \ N]$ (N provided by the user).
- (b) create an array $B=[x^0/0! \ x^1/1! \ x^2/2! \ \dots \ x^N/N!]$ (using element-wise operations - see Sheet 3 for help).
- (c) Sum up all the elements of array B .

► **TASK 3:** Modify `emac.m` 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.m` onto Canvas. Click on *IUM2016-17 >> Assignments >> Matlab worksheet V – emac.m*. The submission box closes at 1PM.

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

Let's first try to plot $y = 2x + 1$ over the interval $-2 \leq x \leq 2$. In the Command window, type the commands:

```
1 x=[-2:2];
2 plot(x,2*x+1)
```

You should see a straight line. 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

$$x = \text{linspace}(-2,2);$$

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 $-3 \leq x \leq 3$. Use `linspace`.

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

```
plot(x,x.^2, x,2*x+1)
```

(where \mathbf{x} must have been defined). You can add more pairs of x_n, y_n if needed.

► **TASK 5:** Plot all these 3 functions on the same set of axes (with $-3 \leq x \leq 3$).

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

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

Create a new function M-file and type in the following content.

```
1 function y = eloop(x,N)
2 % eloop(x,N) gives the value of e^x using Maclaurin series up to x^N
3 y =0;
4     for k=0:N
5         y = y + x.^k/factorial(k);
6     end
7 end
```

This code does the same job as the file `emac(x,N)` which you worked on, but it accepts an *array* \mathbf{x} as well (unlike `emac`). Try calling it in the Command Window using, say, `eloop([-3:3],5)`. This should give you the 5th-order Maclaurin series for all integer \mathbf{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 $\mathbf{k}=0$ on line 4, goes through to line 6, then comes back up and repeats with $\mathbf{k}=1$, and so on until $\mathbf{k}=\mathbf{N}$. With each run of the loop, it collects the term $\frac{x^k}{k!}$ and keeps adding it to the total \mathbf{y} . Note from Line 3 that \mathbf{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 $\sum_{k=0}^N \frac{x^k}{k!}$, which is our desired output.

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

- First, use the `plot` command create a basic multi-plot of 5 graphs. Use the function `elooop` to help you do this.
- Now modify this basic plot. In the Figure window, click *Insert* to add the x -axis label, y -axis label, graph title and legend.
- Click *Tools* and *Edit Plot*, then double click anywhere on the plot where you'd like to edit. Use this method to change font size, line thickness *etc.*
- Make sure the curves are plotted in different line styles (solid, dashed, dotted...) or thicknesses. This is to ensure that we can still tell the curves apart even when the graph is printed in black-and-white.

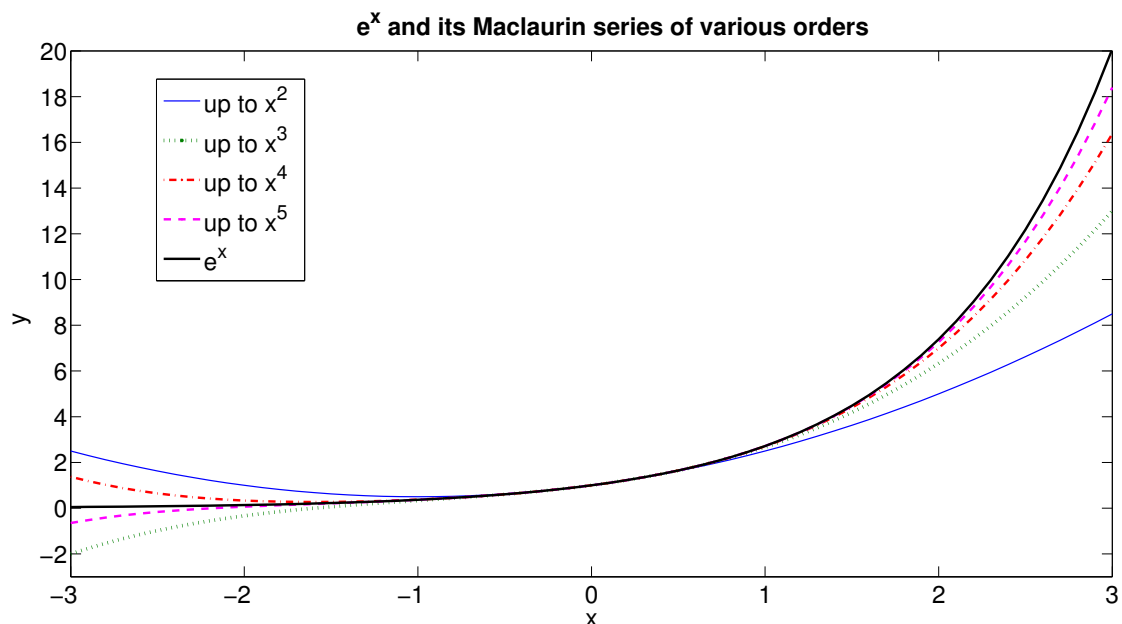
Your figure doesn't have to look exactly like mine, but it should be as informative and easy to read. Feel free to be artistic.

When you are happy with your figure, save it in the jpg format (*Save As*) and upload your jpg file onto Canvas. Click on *IUM2016-17 >> Assignments >> Matlab worksheet V – graph.*

(For future reference, you might like to save your figure as a `.fig` file on your G drive as well. But don't submit the `.fig` file.)

In the text box on Canvas, answer this question:

Give a one-sentence mathematical interpretation of the figure, *i.e.* what are these graphs telling us?



Introduction to University Mathematics 2016-17

MATLAB WORKSHEET VI

Complete the following tasks. Your grade will come the file *harmo.m* to be submitted onto Canvas by **3PM**.

Today we will learn about 3 keywords that will allow you to write powerful codes: *if*, *for* and *while*. Such keywords are very important in programming in any language, not just MATLAB. Read very carefully and don't skip any task.

1 “If”

In a new script file, copy the following bit of code which simulates rolling a single fair die. It congratulates you if you've rolled a 6. Save it as *die.m*

```
1 r = randi(6);
2 if r==6
3     fprintf('Congratulations! You rolled a 6.\n')
4 end
```

Let's try to understand this code.

Line 1 A random integer r , where $1 \leq r \leq 6$, is created. Browse through the help article on `randi` (random integer) if you'd like to know more about it.

Lines 2 & 4 An “*if*” statement begins with “*if*” and must be matched with “*end*”. Think of “*if–end*” as one single structure. No semicolons are needed on these 2 lines

An “*if*” statement is always followed by a logical (*true/false*) test (see Sheet 2 Q3). If the result of that test is *true*, the code within the *if–end* structure is processed. If the test gives *false*, the whole structure is ignored.

Thus, in this case, the user is congratulated only if $r = 6$. If $r \neq 6$, nothing happens.

If you're unsure about how `fprintf` works, consult Sheet 4.

► **TASK 1:** Run the code many times (click the green RUN button). See that you're only congratulated at a reasonable frequency.

► **TASK 2:** Insert another *if* statement so that the code displays “*Oh so close!*” if the die shows 5. Run it many times.

Actually the two *if* statements can be combined into a single structure as shown on the next page.

```

if r==6
    % case 1: congratulate user
elseif r==5
    % case 2: display "oh so close"
end

```

Note that *elseif* is a single word. Again, think of “*if-elseif-end*” as a single structure.

► **TASK 3:** Implement the “*if-elseif-end*” structure in your code (obviously insert the appropriate `fprintf`’s into the above code outline). Make sure your code runs as before.

Suppose we now want to display “*Bad luck. Try again!*” if the die shows 1,2,3 or 4.

We use the keyword “*else*” to take care of all the other possibilities, like so:

```

if r==6
    % case 1: congratulate user
elseif r==5
    % case 2: display "Oh so close!"
else
    % case 3: display "Bad luck. Try again!"
end

```

Note the following points.

- The “*if-elseif-else-end*” statements together form one single structure. It’s also possible to just use “*if-else-end*” (without *elseif*).
- *else* can only appear once in this kind of structure, but *elseif* can appear many times (if it appears at all).

► **TASK 4:** Implement the “*if-elseif-else-end*” structure in your code and make sure that your code runs ok.

► **TASK 5:** Using the “*if-else-end*” structure, modify the code so that the user is congratulated if the die shows 6, but if the die shows anything other than 6, the user is told

“Bad luck. You rolled a ♡.”

where ♡ is the number on the die.

When you are happy with how *if* statements work, you can save and close `die.m`. There is no need to submit it, but you may need to use what you’ve learnt in the upcoming tasks...

2 “for”

Loops are certain bits of codes that are executed over and over again.

The first kind of loop we’ll try is the “*for*” loop. Study the following examples. Try creating a new script file and run each example.

```
% Example 1
for n=1:10
    disp(n)
end
```

```
% Example 2
for k=2:2:16
    fprintf('The square-root of %d is %f \n' , k, sqrt(k))
end
```

Note the following.

- A “*for*” loop begins with “*for*” and must be matched with an “*end*”. Just like previous section, “*for-end*” is a single structure.
- Each “*for*” loop comes with a ‘dummy index’ (**n** in the Example 1; **k** in Example 2).
- The bit of code within the “*for-end*” structure is run with the dummy index taking each and every value in the specified range. The loop stops after the dummy index reaches the last value.

► **TASK 6:** Using the *for* loop, write a 3-line code to produce the following display.

```
2 to the power of 0 equals 1
2 to the power of 5 equals 32
2 to the power of 10 equals 1024
2 to the power of 15 equals 32768
2 to the power of 20 equals 1048576
2 to the power of 25 equals 33554432
2 to the power of 30 equals 1073741824
```

Make sure you understand how the *for* loop works before continuing. Experiment with more examples if you’re unsure.

An important use of the *for* loop is series summation. For example, suppose we want to evaluate

$$\sum_{n=1}^{1000} n = 1 + 2 + 3 + 4 + \dots + 1000$$

We know from previous weeks that we can perform this calculation with an array:

```
sum([1:1000])
```

This is called the *vectorised* approach, which is unique to MATLAB.

But in most other programming languages, this kind of sum is evaluated using a *for* loop. We can try this out in MATLAB. Try running the following code.

```
1 s = 0;
2 for n=1:1000
3     s = s + n;
4 end
5 fprintf('Sum of integers up to 1000 = %f \n', s )
```

So what's going on?

On Line 1, we set **s** to be 0 initially. We will make **s** grow with each cycle of the *for* loop.

In the first cycle of the loop, $n = 1$, and Line 3 reads: **s** = 0 +1 =1.

Remember that in programming, = means *assignment*, not equality. So **s=s+n** means “new **s** = old **s** + n ” (see Sheet 2).

In the second cycle of the loop, $n = 2$, so **s** = 1 +2 =3.

In the third cycle of the loop, $n = 3$, so **s** = 3 +3 =6.

In the fourth cycle of the loop, $n = 4$, so **s** = 6 +4 =10, and so on until **n**=1000.

Once the loop ends, line 5 reports the final value of **s** as 500500.

► **TASK 7:** Open a new script file called **harmo.m**. Write a code using a *for* loop to evaluate the *harmonic series* with 5000 terms

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{5000}.$$

(You will meet this series again in *Numbers, Sequences & Series*.) If you've done this right, typing **harmo** in the command window should produce the same result as the vectorised method, which is simply **sum(1./[1:5000])**

► **TASK 8:** Modify `harmonic.m` so that, when run, it produces the following output

```
The harmonic series with 10 terms = 2.928968
The harmonic series with 20 terms = 3.597740
The harmonic series with 30 terms = 3.994987
The harmonic series with 40 terms = 4.278543
The harmonic series with 50 terms = 4.499205
The harmonic series with 60 terms = 4.679870
The harmonic series with 70 terms = 4.832837
The harmonic series with 80 terms = 4.965479
The harmonic series with 90 terms = 5.082571
The harmonic series with 100 terms = 5.187378
```

This is a more challenging task which requires careful planning and experimenting. Tips:

- a) Use the space below to plan the structure of your code. Don't just launch into it.
- b) If this seems difficult, try to solve an easier version of the problem first. Break the problem down into mini steps. Before discussing with your friends, *think* for yourself first.
- c) Your solution should probably involve at least one “*for*” loop. An “*if*” statement can also help, but the problem can also be solved without “*if*” statements.
- d) Think about the efficiency of your strategy – are there redundant calculations?

There are many ways to solve this problem. Extra marks will be given for originality.

Identical clusters of codes will not all get the full mark.

Copying and pasting the same bit of code 10 times is not an acceptable solution.

Now submit `harmonic.m` onto Canvas. The submission box closes at 3PM.

3 “while” (optional, extra credits)

The “*while*” loop carries on repeating as long as a certain condition remains *true*.

For example, suppose we want to find the smallest integer N such that

$$1 + 2 + 3 + \dots + N > 9999$$

We can solve it easily with the following code.

```
1 s = 0;
2 n = 0;
3 while s<=9999
4     n = n+1;
5     s = s+n;
6 end
7 fprintf('N= %d \n', n)
```

Note that a “*while*” must also be matched with an “*end*”. The “*while–end*” structure runs repeatedly as long as the logical test ($s \leq 9999$) is *true*.

Studying the code, you will see that the summation technique is similar to the *for* loop, except this time we have to increase n manually one at a time, since we don’t know how big n could get.

When we reach $n=141$, $s=10011$, the *while* loop is not executed again since the logical test ($s \leq 9999$) returns *false*. The last line then reports this final value of n .

You can check that this is correct with the commands `sum(1:140)` and `sum(1:141)`.

► **TASK 9:** By modifying the above code, answer as many of the following questions as you can. Include your answers by inserting a “comment” on your Canvas submission. Just the numbers will do (no code needed).

a) How many terms does it take for the harmonic series to exceed 15?

b) Find the smallest odd number N such that

$$1 + \frac{1}{3^2} + \frac{1}{5^2} + \dots + \frac{1}{N^2} > 1.2337$$

c) Find the smallest integer N such that

$$\frac{1}{2} \cdot \frac{3}{4} \cdot \frac{5}{6} \cdots \frac{N}{N+1} < 10^{-3}$$

Name:

Date:

Introduction to University Mathematics 2016-17

MuPAD

Complete the following tasks and hand in the sheet by the end of the lab session.

This week we will be exploring *symbolic* mathematics on the computer. This includes, for instance, solving an equation in x , differentiation, integration *etc.*

We *could* perform these tasks in MATLAB's Command Window as in previous weeks, but it is much easier to use a dedicated symbolic environment called *MuPAD*. It's not entirely clear what the acronym stands for, but "*uPAD*" certainly stems from *University of Paderborn*, Germany, where the software was first developed (and I'm guessing *M* is for Mathematics). MuPAD used to be a software in its own right, but was bought by MATLAB and incorporated into its *Symbolic Toolbox* since 2008. This is why the syntax in MuPAD is slightly different from that in MATLAB, as you will see today.

To start, type `mupad` in the Command Window and you'll be brought into the MuPAD environment. This is where you'll be working today. Note that the MATLAB's Command Window is still available should you wish to work on it.

1. MuPAD basics. Basic MATLAB operations (`+`, `-`, `*`, `/`, `^`) carry over into MuPAD (test it out). Press enter after each calculation.

However, you will find that division seems to produce fractions and not the numerical values (e.g. try typing `5/10` or `7/49`). This is because MuPAD assumes you want to do symbolic mathematics and so it keeps expressions in *exact* forms as far as possible (otherwise you could have just used MATLAB instead). This is also true for other functions like `sqrt`, `log`, `sin`, `cos` *etc.* (try them).

To reiterate this difference, try to evaluate 2^{1000} in both MuPAD and MATLAB. You will see that MATLAB is designed for numerical tasks, and its behaviour is governed by the same rules as other programming languages (essentially just binary arithmetic). On the other hand, MuPAD is designed for symbolic/exact calculations. It is more accurate, but slower and requires a lot more computer memory.

But what if you want to know the numerical value of exact expressions like `sqrt(2)` in MuPAD? You can either:

- use decimal points: `sqrt(2.0)` or
- put the expression into the `float` function: `float(sqrt(2))`

(a) Write down the MuPAD command for calculating $100^{2/7}$. *Ans:*

(b) Write down the last 3 digits of 2^{34} . *Ans:*

You might have noticed by now that the MuPAD environment works a bit like a *Word* document, in the sense that you can go back to previous lines, amend and rerun them. You can delete, copy, paste lines as you would in a *Word* document.

Important: All MuPAD commands should appear in red. Texts in black are not processed. If you have accidentally deleted the MuPAD prompt [, use the button with the little red π symbol on the top panel to “Insert Calculation”.

2. MuPAD vs MATLAB. There are important syntax differences between MATLAB and MuPAD.

Go to MATLAB and look for a Help document on “*Differences Between MATLAB and MuPAD Syntax*”. Read the article and fill the blanks below. For some entries below you might need to explore the Help button in MuPAD, or just experiment! The first row has been done for you.

Task	MATLAB command	MuPAD command
Variable assignment	<code>x=2</code>	<code>x:=2</code>
Suppress output	<code>x=2;</code>	
Infinity	<code>Inf</code>	
e^{1+i}		
$\tan^{-1}(\pi/8)$		
Factorial of 10	<code>factorial(10)</code>	
Clear variable x	<code>clear x</code>	
	<code>clear</code>	
	<code>%</code>	<code>//</code>
		<code>%</code>

Exercise: Define variable a as $7\pi/8$ (see first row of the table). Use MuPAD to find the exact values of $\sin(a)$ and $\cos(a)$.

Ans: $\sin a =$ $\cos a =$

3. Solve. Suppose we want MuPAD to solve the equation $x^2 - 3x - 154 = 0$, type

```
solve(x^2-3*x-154=0, x)
```

(If you get an error, see the table on the previous page.) You should find two solutions: -11 and _____. The `solve(A,B)` function takes two arguments: **A** is the equation you want to solve, **B** is the variable to solve for.

You could also have done the above calculation by defining the equation first.

```
eq:= x^2-3*x-154=0
solve(eq,x)
```

- (a) Write down the solutions to the equation $x^3 - 6x^2 - 19x + 24 = 0$.

Ans:

Now see what happens if we try to solve

$$x^3 - 3x + 1 = 0.$$

You should find that MuPAD can't tell you the *exact* answer. If this happens, we can invoke the *numerical* solver instead of a symbolic one. The command is

```
numeric::solve(eq,x)
```

where **eq** is the equation. You will find $x \approx -1.88, 0.347$, and _____ (to 3 SF).

- (b) Write down the solution of the equation $e^{-u} = u$ (to 4 SF).

Ans:

- (c) Find all solution(s) to the equation $|x| = 2 - \cosh x$ (to 4 SF).

Ans:

The `solve` command can also be found when you click *General Math* on the right-hand panel. Point-and-click options from this panel to save some typing.

4. Algebra. Let's use MuPAD to do some basic algebraic manipulation. Let's first look at `expand` and `factor` commands. Try these and see if you understand the results.

- `factor(x^2-6*x+5)`
- `expand((x-3)^5)`
- `expand(sin(x+y))`

- (a) Use **factor** to simplify the expression $\frac{x^2 - 1}{x^3 + 5x^2 - x - 5}$. Ans: _____
- (b) The coefficient of x^2 in the expansion of $(2x - 3)^5 \left(1 - \frac{2}{x}\right)$ is _____.
- (c) Write down the command which will express $\cos 4x$ in terms of $\cos x$.

Algebraic simplifications can also be done with **simplify**, **Simplify**, **combine**, **collect** amongst many other functions. Read the help documentations and experiment with these commands in your own time, but let's move on for now.

5. Differentiation. Suppose we want to differentiate $\sin(x)$. The command is

```
diff(sin(x),x)
```

The syntax is similar to the **solve** command: in **diff(A,B)**, **A** is the expression to differentiate, **B** is the variable to differentiate with respect to.¹

We can also do multiple differentiations: to differentiate $\frac{1}{x}$ five times, type

```
diff(1/x ,x$5)
```

Sometimes the gradient at a specific point might be needed. Use the function **subs** to substitute in a value into an expression. For instance, here's how to calculate the gradient of the tangent to the curve $y = x^4 - 10$ at $x = 3$:

```
y:=x^4-10
D1:=diff(y,x)
subs(D1,x=3)
```

You should find² the answer = 108. Of course you could have combined everything into a single line, but you'll probably make fewer mistakes if you break it down into small steps.

- (a) If $y = \ln(\ln 5x)$, then $\frac{dy}{dx} =$ _____.
- (b) If $u = \tanh^{-1}(\tan v)$, then $\frac{du}{dv} =$ _____. (try **simplify**)
- (c) If $s(t) = e^{-t^2} \cos t$, then $s''(0) =$ _____.

*Hint: to make **subs** evaluate the result after substitution, try adding the option:*
subs(....., EvalChanges)

¹You can also use the Derivative button on the righthand panel. The *partial derivative* $\frac{\partial}{\partial x}$ reduces to the regular kind of derivative $\frac{d}{dx}$ for a function of one variable.

²For the last steps, you could also write **diff(%,x)** and **subs(%,x=3)**.

(d) (slightly tricky!) Let's define a function $H_n(x)$ by

$$H_n(x) = (-1)^n e^{x^2/2} \frac{d^n}{dx^n} e^{-x^2/2}.$$

This function appears frequently in probability and quantum mechanics. Try evaluating this function with $n = 1$ in MuPAD. You should find $H_1(x) = x$. Now evaluate it with $n = 2, 3, 4, \dots$. They should look like simple *polynomials* with no exponentials around.

$$H_2(x) =$$

$$H_3(x) =$$

$$H_4(x) =$$

$$\text{Calculate } H_{16}(x=0) =$$

[*Hint: to minimise typing, you could first define `n:=1` and define `H` in terms of `n` and `x`, then change `n` and rerun. Use `expand` to tidy up the output.*]

6. Integration. Symbolic integration can be done using the function `int` with the same syntax as `diff`. For instance, to integrate x^2 wrt x , type:

```
int(x^2,x)
```

(note that MuPAD doesn't provide the constant of integration). To evaluate definite integrals such as $\int_0^3 x^2 dx$, we use `..` (two dots).

```
int(x^2,x=0..3)
```

MuPAD can deal with a huge range of nasty integrals, even improper integrals with infinite limits or singularities.

However, if no exact answer can be found by MuPAD, use `numeric::int` to invoke the numerical integrator in the same way we did with `solve`.

Use MuPAD to evaluate the following integrals.

$$\int \sqrt{1-x^2} \, dx =$$

$$\int_0^1 \frac{t^4(1-t)^4}{1+t^2} \, dt =$$

$$\int_0^\infty \frac{\sin x}{x} \, dx =$$

$$\int_{-1}^1 \sec(\theta^2) \, d\theta =$$

These last 2 questions are optional (extra credits).

7. Functions. To define a function in MuPAD, use the \rightarrow construction. For example, to define $f(x) = x^2 + 5$, type

`f:=x->x^2+5`

We read this as : “ f is defined such that $x \rightarrow x^2 + 5$ ” (i.e. x is mapped to $x^2 + 5$).

You can now try using f , for example, by typing $f(2)$.

Here is a little exercise combining functions with integration.

Consider the following function.

$$G(x) = \int_0^\infty t^{x-1} e^{-t} dt.$$

(note that t is just a dummy variable which is integrated away.)

Define this function as $G(x)$ in MuPAD. When done correctly, you should find that $G(1/2) = \sqrt{\pi}$.

Note that it is important to type $G(1/2)$ and NOT $G(0.5)$. As in the first exercise, the decimal point will invoke the numerical engine.

Use your function to calculate the following.

$$\begin{aligned} G(3/2) &= & G(5) &= \\ \int_0^1 \ln G(x) dx &= \end{aligned}$$

8. Plot. Plotting graphs is much easier in MuPAD than in MATLAB for common household functions. For example, to plot $y = \sin(x)$ from 0 to 2π , type

`plot(sin(x),x=0..2*PI)`

Plot the graph of the function $y = G(x)$ where $-4 \leq x \leq 4$. Sketch it below.

