Introduction to University Mathematics 2018
MATLAB Worksheet VI

Complete the following tasks. Your grade will come from the file ***harmo.mlx*** *submitted onto Canvas by* ***7PM****.*

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

1 "If"

In a new Live 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.mlx`

```
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. 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 Run Section 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 $\heartsuit$."

where $\heartsuit$ is the number on the die.

When you are happy with how *if* statements work, you can save and close `die.mlx`. 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 Live 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 the 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

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

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".

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 Live Script file called `harmo.mlx`. Write a code using a *for* loop to evaluate the *harmonic series* with 5000 terms

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

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

▶ **TASK 8**: Modify the `harmo.mlx` 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 kind of 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.
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 the `harmo.mlx` onto Canvas. The submission box closes at 7PM.

3 “while”

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

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

We can solve it with the following code.

```
s = 0;
n = 0;
while s<=9999
        n = n+1;
        s = s+n;
end
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 above 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<=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, find the smallest odd number N such that

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