Tutorial 5 – Week 6

Aims:

Upon successfully completing these tutorial exercises, students should be able to:

- Understand how to implement conditions in MATLAB
- Understand how to perform loops in MATLAB

Tutorial 5: Conditions and loops are central topics in most programming languages – they are used everywhere because they are so useful! Make sure you understand them well enough to implement with a problem not seen before.

1.1 For you to do:

Program your script from Tutorial sheet 4.

1. Converts degrees Fahrenheit to degrees Celsius. The script should request degrees Fahrenheit and print the solution to the user. The equation is:

$$CC = \frac{5}{9}(FF - 32)$$

2 Understand how to implement conditions in MATLAB

It is possible to have nested **if** statements, meaning **if** statements inside other **if** statements. It is important to realise that the inner **if** statement will only be tested if the outer **if** statement is true (otherwise the code will be skipped. Can you predict the output of the following code?

```
a=10;
b=7;
if a < 8
    if b == 7
        disp('Monkey');
    else
        disp('Banana');
    end

    disp('Rabbit');
end
```

Note the indentation used with this code (it is very deliberate). Every time a coder writes an **if**, **ifelse**, **else**, **for** or **while** (or similar), the following line is indented. The indentation is then withdrawn whenever an **end**, **else** or **elseif** is written. Matlab will automatically indent as you press enter. This indentation is important to understand how the conditional and loop statements correspond to the **end**, **else** and **elseif**. In the previous example, the first **if** aligns

with the final **end**. Hence, if the condition if false, all code between the **if** and corresponding **end** is skipped. Any inner **if** statements are therefore not tested.

So far, we have only looked at comparing numbers, but it may be useful to compare two strings to see if they are the same. For example, I can request a string from a user with the **input** function.

```
answer=input('Choose a door: (left or right)','s');
```

Then I can compare the string with expected answers and reply differently based on the answer using **if** statements. The function **strcmp** has two inputs (both strings) and returns logical 1 if the strings are identical, otherwise it returns logical 0.

```
answer=input('Choose a door: (left or right) ','s');

if strcmp(answer,'left')
    disp('You find some treasure!');
elseif strcmp(answer,'right')
    disp('Oh no! A velociraptor');
    answer2=input('Did you bring your motorbike? (yes or no) ','s');
    if strcmp(answer2,'yes')
        disp('You use your time machine to escape!');
    elseif strcmp(answer2,'no')
        disp('You are eaten by a velociraptor');
    else
        disp('You confound the velociraptor with stupidity and escape');
    end
else
    disp('Incorrect entry');
end
```

2.1 For you to do:

Create a story line with at least 8 possible outcomes based on the answer to different questions. Unless an incorrect entry is given, a user should be asked at least 3 questions before their story ends. Be creative!

3 Understand how to perform loops in MATLAB

Programs are often required to handle repetitive tasks. A clever programmer will make use of loops to minimise the amount of code required to handle a problem. Let's begin with a simple task. Say we wish to print the string 'Again and Again' 100 times. We could type out disp('Again and Again'); and copy and paste 100 times, but a **for** loop can be used to the same effect with much less effort:

```
for i=1:100
    disp('Again and Again');
end
```

Recall that the colon operator : will create an array of numbers. In this case, an array [1,2,3,4,5…,100] is created with 100 elements. The **for** loop will run as many times as there are elements in the array. Here, **i** will be equal to 1 in the first loop, 2 in the second loop and so on until **i** is 100 in the final loop. Hence, what do you think would be the output of the following code:

```
for i=1:10
    disp(['The value of i is now: ' num2str(i)]);
end
```

Loops are very useful for calculating and recording quantities over time where the same equations apply across time. For example, consider car, which is originally at rest, drive forward at a constant velocity of 10 m/s. If we want to record the displacement of the car over time in an array, we can use the equation:

$$ss = ss_0 + vvv$$

Hence, we can use the previous value of s to calculate the next value of s. We can create a loop that calculates the displacement in time steps of 0.1 s from 0 to 5 seconds by:

```
t=0; % initialise time
s=0; % initialise displacement
v=10; % define the constant velocity
i=0; % initialise indexing variable
Ts=0.1; % define timestep
for t=Ts:Ts:5 % define range of times to be calculated
```


```
        i=i+1; % increment i
        s(i+1)=s(i)+v*Ts; % calculate s(i+1) using s(i)
    end
    plot(0:Ts:5,s); % plot solution
```

Notice how the variable **i** is used to keep track of the position with the array **s**. The first time the loop runs, the second value of **s** will be calculated using the first value of **s**. The second time the loop runs, the third value of **s** will be calculated using the second value of **s** and so on…