Adrian Loekman
UID: 105785820
CEE/MAE M20
October 9, 2021

HOMEWORK 2

1. Cubic Function Type

1.1. Introduction

The goal in this problem is to determine if a cubic function, f, with coefficients a, b, c, and d is simple, monotone, or neither. The key questions are finding the derivative of the cubic function, then finding the roots of the derivative function. Find out the characteristics of the roots to determine the type of function that is corresponding to the user inputs. The results will be printed on the command window according to the characteristics of the cubic function.

1.2. Model and Methods

The script first asks the user to input 4 coefficients: a, b, c, and d that corresponds to the cubic function $f(x) = ax^3 + bx^2 + cx + d$. This will be done using the input function. Next, the value a must not be 0 so an if statement is used to validate the value of a so that it returns an error whenever 0 is inputted as the value of a.

We can do the calculations to find the derivative of the function f(x) by hand, which gives us $f'(x) = 3ax^2 + 2bx + c$. Then we can use a root finding equation to find the roots of the quadratic equation, $x_{1,2} = \frac{-b\pm\sqrt{b^2-4ac}}{2a}$. In this equation, we want to switch the coefficients with the coefficients we got on the quadratic equation.

After obtaining the value of the two roots, use if else statements to determine the characteristic of the cubic function. We use the MATLAB function `isreal` to check if the roots of the quadratic are real numbers. Also, check if the roots are distinct or not using an expression where `(root1 ~= root2)`. If both these conditions are true, we can proceed by calculating the values of f at root1 and root2. Finally, check if f(r1) * f(r2) < 0. If the condition is true, then there should be an output `fprintf('Simple\n')`. If the condition is false then the output should be output `fprintf('Neither\n')`.

If the real number check condition is false, then the output should be `fprintf('Monotone\n')` because the roots of the derivative function f'(x) are not real and distinct.

1.3. Calculations and Results

There are three possible outputs that could happen based on the user's inputs.

```
Enter a value for a (a ~= 0) : 0
Enter a value for b : 3
Enter a value for c : 4
Enter a value for d : 5
Error using hw2_105785820_p1 (line 35)
a must not be zero!
```

When a user input 0 for a, it will print out an error message. Now, let's try another value for each constants.

```
Enter a value for a (a ~= 0) : 1
Enter a value for b : 2
Enter a value for c : 3
Enter a value for d : 4
Monotone
```

When we have these values for the coefficients, it will print out Monotone. This is because the roots of the derivative function are -0.666666666666667 + 0.745355992499930i and -0.666666666666667 - 0.745355992499930i which are not real numbers. This doesn't qualify for the if conditions so then MATLAB will print the else condition, which is 'Monotone'.

```
Enter a value for a (a ~= 0) : 1
Enter a value for b : 0
Enter a value for c : -7
Enter a value for d : -6
Simple
```

This equation will print simple because the roots of the quadratic function are -1.527525231651947 and 1.527525231651947. They are similar but distinct because 1 is a positive root and the other is negative. Thus, f(r1) = -13.128451081042417 and f(r2) = 1.128451081042418. If we multiply f(r1) with f(r2) it would be a negative number. Since a simple cubic function is when f(r1) * f(r2) < 0, the cubic function with these coefficients would be a simple cubic function.

```
Enter a value for a (a ~= 0) : 1
Enter a value for b : -3
Enter a value for c : 2
Enter a value for d : 4
Neither
```

This equation will print neither because the value of f(r1) and f(r2) are both positive. Thus, if f(r1) is multiplied with f(r2), the output would be larger than 0. This fails the if condition to make the function simple. So, the output of the function would print 'Neither'.

1.4. Discussion

The if else statements on this code is very useful to determine the characteristics of a cubic function because it can classify the functions based on how the function and derivative would be relative to the conditions. Notice that the printed output is determined solely by the coefficients that is being put into the equation. In other words, the coefficients a, b, c, and d play a big role in determining the characteristic of the cubic function.

If the graph of the functions is plotted, we can find out why we can say that f is monotone and f(r1) * f(r2) < 0 is simple. In the third example, it is printed as ‘simple’, which means that the function has three real and distinct roots. The check is by multiplying f(r1) * f(r2). This must be less than 0 because if it’s simple it means that there will be a local minimum and maximum at r1 and r2. The value of r1 and r2 has to be the opposite so that it passes the x axis to get the three roots.

The last example on 1.3 shows that the function is neither simple nor monotone. The graph of the function only has 1 root because the local maximum and minimum are both above the x axis. This is because f(r1) * f(r2) > 0, which means f(r1) and f(r2) are either both negative or both positive. This would mean for the function to only have 1 root. If either f(r1) or f(r2) is 0, then the cubic function would have 2 roots where one of them is 0, which is just touching the x axis.

Monotone means that the cubic function is always increasing or decreasing, unlike the more common cubic function where there’s a ‘peak’ and a ‘valley’. This is because there are no real roots from the quadratic equation. The roots of the quadratic equation explains the local max and the local min of the cubic function. Since these roots are imaginary numbers, there are no local max and min. Thus, the cubic function would only have 1 root with the function trend always increasing or decreasing.

This exercise utilizes if else statements along with nested if statements. It is most important to debug the code into little steps so it is possible to track each code and which if else conditions are true for each inputs. So, the equation can only be simple if there are three roots within the cubic function, that is when f(r1) * f(r2) < 0. If this is not true than it’s possible that it would be neither. A monotone function is only when the roots of the quadratic are imaginary. Thus, making the function always increasing or decreasing.

2. Write a MATLAB script that takes a date as input and returns the day of the week.

2.1. Introduction

The main goal in this problem is develop a set of codes to print out the day of the week from a given date, month, and year in a precise format `DD MM YYYY`. Some key questions include validating the inputs, which are inputting the correct date month and year according to the convention of Gregorian Calendars and using an algorithm to determine the day of the week.

2.2. Model and Methods

The script starts by soliciting user inputs for month, day and year. This must be in a precise format where month must be in MMM which are the first three letters of the month in capital letters, day must be in DD, and year must be in the format YYYY which is a 4-digit number from year 1 to 9999. Thus, when wanting to input year 450, users must input 0450. In this case, users that input 1, 2, or 3 digits would still be acceptable since later in the code, there will be a conditional statement that when the number of digits inputted is less than the maximum digits a string of zero(s) will appear in front of the input. This will result in a string as the output from the class conversion from double to string.

The next step is to validate the inputs so that unreasonable inputs will output an error. The year must be from 0001 to 9999 so then if a year is out of that range, it will output an error. Each month has different dates so an if statement can be used to validate the day input and also the month format.

```
if strcmp(month,'JAN') || strcmp(month,'MAR') || strcmp(month,'MAY') ||
...
 strcmp(month,'JUL') || strcmp(month,'AUG') || strcmp(month,'OCT')...
        || strcmp(month,'DEC')
     if day < 1 || day > 31 %in these months, there are only 31 days
             error('Please input a valid day');
     end
```

From the code above, we see that `strcmp(month,'JAN')` is used for validating the month. And the nested if is used to validate the day. The code below this is just another elseif statements for months that end in 30 days.
February has a special case because it only has 28 days and will add one day if the year is a leap year. In this case, we will need to add a special code to take into account in case the year inputted is a leap year.

```
    startLeapFlag = 0;

if  (mod(year,4) == 0 && mod(year,100) ~= 0) || ( mod(year,400) == 0)
      startLeapFlag = 1;
    end

    if day < 1 || day > (28 + startLeapFlag)
        error('Please input a valid day')
    end
```

In the code above, there is a new variable `startLeapFlag` set equal to 0. The if condition is the condition for a leap year, and if the condition is true, it will change the variable `startLeapFlag` to 1. Then, it is possible to validate the input for the month of February with the correct number of days. If letters other than the first three letters in the month is inputted, the code will print out an error message.

Next, the algorithm will be set up. Since the algorithm is already shown in the problem statement along with the details, we would only need to set up the variables and do the correct math operations. One of the most important variables is the month. Since the input of month is a char, we would need to change it to a number corresponding to each month (i.e. January = 1, February = 2, etc.). So we need a code such like

```
If month = January
    Month = 1
Elseif month = February
    Month = 2
…
```

This pseudocode is applied until the month of December = 12. Note that there is a special brackets that is used in the algorithm that will round the result to the integer. Thus, the algorithm w will be looking like this:

```
w = mod((d + floor(2.6*m - 0.2) + y + floor(y/4) + floor(c/4) -
2*c),7);
```

The other variables will be quite easy because it tells us what each variable is expressing. For variable m, we will need to use mod function according to the input month. Then, y and c will be calculated using mod and floor functions where y is the last two digits of the year and c is the first two digits. We can get this by using mod and floor functions. There is a special case when the input month is either January or February, the year must be subtracted by 1. Thus, another if statement is needed before calculating the variables y and c. Then, we can add the month later after finishing calculating the algorithm.

Since the output from the algorithm is a number from 0 (Sunday) to 6 (Saturday), the numbers must be converted to a string (i.e. Sunday, Monday, Tuesday, etc.). In order to do this, another if else statements must be created. The code would look similar to this:

```
if w == 0
    dayOfWeek = 'Sunday';
elseif w == 1
    dayOfWeek = 'Monday';
```

This is done until w == 6 where the day of the week is ‘Saturday’.

Lastly, the outputs will be printed. However, in order to get the correct format where MMM DD YYYY is a ‘dayOfWeek’, we need another if statement so that if a digit less than the maximum digits is put on the input, it will still output a 0 in front. The if statements will also change the double class of both year and day into string. This is because if we keep it as a double the zero in front would not be printed. The code for this conversion would be like this:

```
if day>= 1 && day < 10
    day = sprintf('0%d', day);
else
    day = sprintf('%d', day);
end
```

The function sprint is not used to convert the double into a string. For the year, we can adjust the number of digits according to the input.

The fprintf function can be used to print the output from the inputs

```
fprintf('%s %2s %4s is a %s\n', month, day, year, dayOfWeek);
```

2.3. Calculations and Results

There are some variations of outputs when the code is inputted. Let's start with a false date

```
Please enter the month as MMM (e.g. JAN): HHU
Please enter the day as DD (e.g. 01): 21
Please enter the year as YYYY (e.g. 2000): 209
Error using hw2_105785820_p2 (line 69)
Please input a valid month
```

We get an error because the input for month is unidentified as a month. Next, we can input a date that is less than 10.

```
Please enter the month as MMM (e.g. JAN): APR
Please enter the day as DD (e.g. 01): 08
Please enter the year as YYYY (e.g. 2000): 2018
APR 08 2018 is a Sunday
```

We can see that the output has a 0 in front of the date 8. This is because the double is converted to a string to have the correct format DD. Now, we can also try inputting a year less than 4 digits.

```
Please enter the month as MMM (e.g. JAN): SEP
Please enter the day as DD (e.g. 01): 20
Please enter the year as YYYY (e.g. 2000): 97
SEP 20 0097 is a Friday
```

We also see that there are 00 that fills in the empty slot on the format YYYY. This can be done using sprint and adding a string '0' to print at the output.

2.4. Discussion

From the code above, we get a straightforward input. However, the complexity of this problem arises when the validity of the month, day, and year needs to be addressed. This is quite difficult since there are many variations in dates, along with printing errors when the date input is unreasonable. In addition to that, there are many conditions that must be fulfilled for the date to be correct like in real life.

The takeaway lesson from this exercise is mainly on the if else statements. The logic must also be used in this problem because we must know how many days there are in a month and consider leap years. This is hard because there is no way to calculate every time we code. Thus, if else statements are very helpful in this case. I just also learned about the algorithm. It is unique because it never prints out the wrong date as long as the correct conditions are fulfilled. Other things to note as well are basic knowledge about prior lectures and logic that can be useful to fulfill if else conditions.

3. Neighbor Identification

3.1. Introduction

The main goal of this problem is to print out the 'neighbors'(i.e. the numbers right next to the cell) of a cell on a grid rectangular array m * n. The number starts from the upper left and goes down until m rows, and continues on the next column, and so on. The inputs will be solicited from the user. Validity will be check since the value m and n must be greater than tree. And the cell, p, must be between 1 and m * n. The 'neighbors' must be printed in ascending order and give an output when the cell p is a corner node.

3.2. Model and Method

First of all, we will need to use an input function that are all numbers for variables m, n, and p. m is the number of rows, n is the number of columns, and p is the identified cell. Next, a validation is needed to make sure that m & n must be a value greater than 3. This is done because otherwise, the algorithm to find the neighbors would be incorrect. So, we can use the condition m < 3 || n < 3 would print an error. Variable p has condition where it must be from 1 to the maximum value m*n.

The problem-solving method for this question involves a large if else statement with nested if else statements as well. I split up the task into 6 big sections which are: corner cells, top cells, bottom cells, left cells, right cells, and all the other cells which I will refer to as 'middle cells' from this point on. Each cell has unique characteristics to one another. Thus, the only way is to create multiple if else statements since matrix and arrays are not allowed.

In the case for corner nodes, the condition would be cell p has to be either 1, m * n, m, and (m*n) – (m-1). Each corner will have 3 neighbors. Then from this point, each corner would have a unique algorithm to find its neighbors. So the pseudocode would be something like :

```
If p = (all the corner nodes connected by ||)
        If p = corner 1
                n1 = right of corner 1
                n2 = diagonal bottom right of corner 1
                n3 = cell below corner 1
        elseif p = corner 2
        …
```

This is done uniquely for each corner because each one has unique algorithms. There is no possible way to get an ascending order of the correct neighbors only by using 1 formula. Then, print the cell along with its neighbors and also print out 'Corner node'. Since all corner nodes have 3 neighbors, the output would look like this :

```
fprintf('The neighbors for Node %d are %d %d %d\n', p, n1, n2,…
        n3);
fprintf('Corner Node\n');
```

There are 4 %d that corresponds to the cell and its 3 neighbors.

Left cells are cells that are numbered 2 through (m-1). Each of the left cells will have 5 neighbors. The pseudocode will be :

Continuing from the pseudocode above…

```
elseif p = 2<=p<= (m-1)
    n1 = the cell above
    n2 = the cell below
    n3 = the cell diagonal right up
    n4 = the cell right
    n5 = the cell diagonal right down ….
```

This applies to all leftmost cells for whatever the value m is. Then we can get the output

```
        fprintf('The neighbors for Node %d are %d %d %d %d %d\n',
        p, n1, n2, n3, n4, n5);
```

There are 6 %d now because the any leftmost cell will have 5 neighbors.

The rightmost cell also uses a similar algorithm to the leftmost cells. However, the specific neighbor identification will be slightly different. The pseudocode would be:

```
Continuing from the codes above…
Elseif p = the rightmost cells except the corner nodes.
    n1 = cell diagonal left up
    n2 = cell to the left
    n3 = cell diagonal left down
    n4 = cell above
    n5 = cell below
…
```

The output of the rightmost cells should be exactly the same as the leftmost cells because each rightmost cells should have 5 neighbors. Thus, 6 %d including the cell, p.

For the bottom cells, each cell that will be inputted (i.e. cell p) will only have 5 neighbors. The code would be:

```
Continuing from the codes above…
Elseif p = the bottom cells except the corner nodes.
        n1 = cell diagonal left up
        n2 = cell to the left
        n3 = cell above
        n4 = cell diagonal right up
        n5 = cell to the right
    …
```

Also, continuing from the bottom cells the top cells, excluding the corner nodes, would have a similar code setting:

```
Continuing from the codes above…
Elseif p = the top cells except the corner nodes.
        n1 = cell to the left
        n2 = cell diagonal left bottom
        n3 = cell below
        n4 = cell to the right
        n5 = cell diagonal right up
    …
```

In this case, the output function fprintf will be exactly the same with the rightmost and leftmost nodes because of the same amount of neighbors, 5.

The last part is the 'middle cells', which are cells not located at the sides of the array. In this case, cell p will have 8 neighbors because of the setting of the array. The code should identify first that it's not any of the conditions above, then right the neighbors. Note that the neighbors also depend on the value of m and n.
pseudocode:

```
else (all other cells that are not cells on the side)
                n1 = cell diagonal left up
                n2 = cell to the left
                n3 = cell diagonal left down
                n4 = cell above
                n5 = cell below
                n6 = cell diagonal right up
                n7 = cell to the right
                n8 = cell diagonal right down
        …
```

Thus, the print statement should have 9 %d including the cell identified.

```
fprintf('The neighbors for Node %d are %d %d %d %d %d %d %d
%d\n', p, n1, n2, n3, n4, n5, n6, n7, n8);
```

3.3. Calculations and Results

The output for this code when executed are variative. First, let's look at a false input

```
Enter the number of rows (M): 1
Enter the number of columns (N): 3
Enter the cell number : 9
Error using hw2_105785820_p3 (line 39)
M and N must be greater than 3

Enter the number of rows (M): 4
Enter the number of columns (N): 4
Enter the cell number : 29
Error using hw2_105785820_p3 (line 43)
P must be between 1 and M*N
```

Notice that there are errors when the conditions are not fulfilled. In the first input, m and n has values less than 4, thus will printout an error. Similarly, the second input has p larger than m*n, thus another error is printed. The next lines will show if the input is a corner node.

```
Enter the number of rows (M): 4
Enter the number of columns (N): 6
Enter the cell number : 4
The neighbors for Node 4 are 3 7 8
```

```
Corner Node
```

This output above is when the input is a corner node. This cell has 3 neighbors and there's an additional output stating the input cell, p, is a corner node. The next output will show the output for cells with 5 neighbors (left, right, top, bottom cells excluding the corners)

```
Enter the number of rows (M): 4
Enter the number of columns (N): 6
Enter the cell number : 3
The neighbors for Node 3 are 2 4 6 7 8

Enter the number of rows (M): 5
Enter the number of columns (N): 5
Enter the cell number : 16
The neighbors for Node 16 are 11 12 17 21 22
```

And, when one of the 'middle cells' are used as an input:

```
Enter the number of rows (M): 4
Enter the number of columns (N): 6
Enter the cell number : 18
The neighbors for Node 18 are 13 14 15 17 19 21 22 23
```

These are the different outputs for different inputs throughout the rectangular array. Each outputs are correct since these cells are located correctly.

3.4. Discussion

The most important lesson from this problem is the rigorous logic that needs to be addressed even before initiating to code on MATLAB. Each algorithm must be sought thoroughly for every location of cell p in the rectangular array. Another important takeaway from this problem is to pay attention to the conditions for each location of cell p. The condition play a big role in progressing through the right condition and getting the correct neighbors.

There is one interesting aspect that I obtain from this problem. I see that as I go through finding each neighbor for the cell, p, I found out that each neighbor is an extension from the previous condition. For example, the neighbors for cells that have 8 neighbors have identical neighbors' algorithm with 5 and 3 neighbors but added with the neighbors that the cells with 5 and 3 neighbors do not have. Surprisingly as well, I found out that the neighbors' algorithm for cells that have 5 neighbors are very similar. I wonder if there's a way to only use 1 if else statement to solve for the cells with 5 neighbors, but I believe it would be impossible since every neighbor set is unique.

The outputs fprintf statements also seem redundant to me and I also offer an alternative to use another if else statement which would check the conditions of calculating the amount of neighbor the cell has, then it can automatically print 3, 5, or 8 neighbors.

To summarize, this problem was challenging because work outside MATLAB is needed. However, the problem becomes easier since the algorithm is repetitive and simple. It’s just a little tedious and tiring to code a lot of similar commands.