Mauricio Deguchi
UID: 405582562
CEE/MAE 20
October 12, 2021

Homework 2

1 Cubic Function Type

1.1 Introduction

The goal of this problem is to determine the behavior of a cubic function as simple, monotone, or neither based on the coefficients input by the user. One of the major challenges will be determining whether the quadratic function (the derivative of the cubic function) has roots.

1.2 Models and Methods

The script first obtains the values of the coefficients using the `input` function. These coefficients are stored and evaluated using two checks to determine the behavior of the function. The first uses the derivative and determines whether the roots exist by looking at the discriminant of the quadratic equation:

$$f'(x) = 3ax^2 + 2bx + c$$
$$B = 2b \;\&\; A = 3a$$
$$discriminant: B^2 - 4Ac$$

If the discriminant is negative, the roots of f'(x) will not exist and thus f is monotone. We can conclude f(x) is monotone because there are no critical points using f'(x), thus f(x) will not have any local max/mins meaning that it is only increasing or only decreasing. If the roots do exist, we calculate them and then check case two. In case two the roots of f'(x) are plugged into f(x) and if their product is less than zero, the equation is simple, otherwise it is neither simple nor monotone. This is because, when we evaluate f at the roots of f', we are bounding two critical points where f(x) can go from decreasing to increasing or vice-versa. If one of these values is positive and the other is negative, the function must cross the x axis creating 3 roots for f(x). Otherwise, f(x) may not cross the x axis or may cross it less than 3 times.

1.3 Calculations and Results

When the program is executed, the coefficient values checked for validity and stored. We first test if the function is monotone. We find the coefficients of the derivative and relabel our variables where B = 2b and A = 3a from the derivative. Next, we calculate the discriminant, if it is not positive, we can declare that f(x) is monotone and return `Monotone` to the user (If the discriminant is 0, our cubic function plateaus once and I was unsure if this plateau meant the function was still solely increasing, so I concluded that if it plateaus, the function is not monotone). Otherwise, we calculate the roots using the quadratic formula and calculate their values in f(x), one of the roots and values is shown below, the other uses the subtraction in front of the square root.

```
r1 = (-B + sqrt(B^2-4*A*c))/(2*A);
fr1 = a*r1^3 + b*r1^2 + c*r1 +d;
```

From here, we can determine if f(x) is simple or neither by multiplying the two values of f(x) at the roots of f'(x). If the value is negative, we return the statement `Simple` to the user, otherwise we return `Neither` to the user.

1.4 Discussion

The completed problem used simple Matlab code, I felt that writing the program wasn't too difficult. Moreso, understanding the values we were working with and having to explain how they correlated, was the challenge. This was a good introductory experience to working with logical statements as I was able to apply what we learned in class into a simple program. This helped ensure that I was understanding the mechanics of the various logic statements.

2 Day from Date

2.1 Introduction

The goal of this problem is to identify the day of the week from any given input date (MMM/DD/YYYY). This problem will implement an algorithm which calculates the day to a value between 0-6 corresponding with a day of the week. Our program will solve this algorithm and return the day of the week. Some major challenges will be to ensure the validity of our input and to account for leap years.

2.2 Models and Methods

The script first obtains three different inputs, the day, the month, and the year. The month is stored as a character array while the day and year are stored as numbers, this satisfies the desire input format; the validity of the month is then checked using a switch statement to assign a corresponding numerical value to the month for computation later in the code. Meanwhile, the validity of the year is checked through the following line:

```
elseif (year > 9999) || (year < 0) || (floor(year)~=year)
```

which ensures that any year of up to 4 digits isn't negative and isn't a decimal. A similar preliminary check is used for the validity of the day. Though, a more extensive check is necessary to ensure the validity of the day. In the months of January, March, May, July, August, October, and December any day 1-31 is valid, for months April, June, September, and November any day 1-30 is valid. And for February, we must check if input year is a leap year (a year is a leap year if it is evenly divisible by 4, except when it is also evenly divisible by 100 but not by 400). If so, any day 1-29 is valid, otherwise, only 1-28 is valid. After all our check have been completed, we can move on to the algorithm.

2.3 Calculations and Results

The calculations are almost solely driven by the formula and conditions given in the problem statement.

$$w = \left(d + \lfloor 2.6m - 0.2 \rfloor + y + \left\lfloor \frac{y}{4} \right\rfloor + \left\lfloor \frac{c}{4} \right\rfloor - 2c\right) \bmod 7,$$

The mod tool in Matlab uses a slightly different syntax than the one on the equation so this is translated into the following line of code:

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

d was converted to its numerical value previously; m is calculated using the proper matlab syntax for 'mod'. Also, there is an if statement to account for the note provided at the bottom of the question. It subtracts 1 from the year when the month is 1 or 2 (Jan or Feb). We can then calculate y using the mod function as shown in the lecture example and c is calculated using the floor function as shown:

```
c = floor(year/100);
```

This eliminates the final two digits by turning them into decimal points and flooring the value to a two-digit integer. Using the formula and constants provided in the problem statement, we can then solve for the value of w, given the conditions, and output our day of the week. Once we have our appropriate value of w, we can use the switch functions to use that value to assign our day of the week. Finally we can print the day of the week and the initial date input.

2.4 Discussion

When the program runs, it takes all the inputs, translates them to these values which can be put through the algorithm formula and lastly, a switch statement interprets the value of w and converts it to the corresponding day of the week. The day is then printed alongside the input date we stored earlier. Using the switch function, I was able to clean up my code at least slightly. Having clean, easily understandable code will be crucial as I progress through this course and the code gets more complicated. This problem was challenging, but it allowed me the opportunity to apply a lot of different functions and logical statements in a more enhanced problem than the previous one. I developed a stronger understanding of new functions and Matlab syntax.

3 Neighbor Identification

3.1 Introduction

The goal of this problem is to develop a program which will ask the user to input an MxN grid and identify a tile 'P' in the grid. The program will then return the neighboring tiles to the user in ascending order. The greatest challenge will be that the program cannot use matrices or arrays to help produce the desired result.

3.2 Models and Methods

For starters, we need to take three input values from the user for M, N, and P. These are all obtained using the `input` function and floored to have the program still run if the user inputs a decimal value. Next, we must check the validity of these inputs. To check the validity of P, P must be between 1 and N*M because the tiles go in ascending order from 1 to N*M, if either of these are not met, an error statement is output. Next, to check the validity of the grid size, M and N must be greater than 3, so if M or N is less than 3 the grid is not well defined, and we output an error statement. The grid must be at least 3x3 in order to have at least 1 center tile which will be how we categorize our neighbors.

After our grid and point have been verified, we can move on to identifying the neighbors. We can do so by considering the various types of tyles we can expect to encounter, there are a total of 9: corner tiles (4, each with their own type of neighbors), top edge tiles, bottom edge tiles, left edge tiles, right edge tiles, and center tiles. Now that we know what our tiles can be categorized as, we must specify numerical conditions.

If P is a center tile, then it must have a top left neighbor located at P – M – 1 (P – M shifts to the left and '-1' moves up one tile) and a bottom right neighbor at P + M + 1. (P + M shifts to the right and '+1' moves down one tile). The logical '>0' and '<=N*M' ensures that these neighbors exist within our grid.

```
if ((P - M - 1) > 0) && ((P + M + 1) <= N*M)
```

3.3 Calculations and Results

Now that we know that P is a center tile, we can begin to identify its neighbors. Its upper left neighbor was detected earlier, and we can calculate the next 2 neighbors + 1 and + 1 again to move down the column. Next, we have a neighbor directly above and below P which are located at P – 1 and P + 1 respectively. Lastly, we have the right sided neighbors of P. The upper right neighbor is located at P + M – 1 and we can move down this row + 1 and +1 again to get the last two neighbors.

This method is repeated for all 9 cases, first we detect which type of tile we have and then we set up the values of the neighboring tiles for such case. The center tiles has every possible type of neighbor which makes it a good sample case. The other categories of tiles will use the same methods described for the center tile, but remove some of their neighbors due to location.

3.4 Discussion

The method I used, directly identifies all the types of tiles we can expect to encounter, set conditions for each type of tile, and then has predetermined formulas for the values of the neighbors in ascending order. Another method would be to use an array to identify if each type of neighbor exists and if they do, add them to the array and print the final array. This problem required me to use the sample matrix provided and generalize about any array and the categories of tiles we could expect to encounter. This problem really challenged my problem-solving methods. The constraint of no arrays stumped me initially and the method I ended up using felt long winded, but the code runs smoothly, nonetheless. This was by far the most I've been challenged in coding, but I was able to come to a solution.