Mauricio Deguchi
UID: 405582562
CEE/MAE 20
October 19, 2021

Homework 3

1 Golden Ratio

1.1 Introduction

This problem seeks to approximate the golden ratio using two methods. The first method approximates cos(36°) using a summation. The true summation goes to infinity but in this approximation, we take a user's desired error to end the summation. The second approximation uses the ratio between two continuous numbers in the Fibonacci sequence. We want to find the number of iterations in each sequence required to achieve the same precision in the golden ratio.

1.2 Models and Methods

For the cosine approximation, we first establish all of the constants (converting the degrees to radians) that are given to us and the initial conditions in order to create the while loop. This while loop compared our current approximation's difference from the previous approximation and checked if the difference was greater than the user's input error. The loop continued until our difference was smaller than the input error. The while loop functions by adding each consecutive term in the summation sequence for each round we go through the loop. The loop itself has a counter which is used in the calculation and can be used at the end to show the number of times we had to cycle to obtain our approximation of cosine.

For the Fibonacci sequence, we take another while loop where the exit condition is that the difference between the previous golden ratio found in the cosine approximation and our current ratio must be less than the error provided by the user. We then calculate the golden ratio by using three consecutive digits. The second is divided by the first to find the approximation of the golden ratio for that cycle; in the while loop, we calculate their sum and store that as a temporary third value. The first value is then reassigned to the second value and the second value is reassigned to the third value and the loop starts again if the ratio does not satisfy the exit condition.

1.3 Calculations and Results

The value of the golden ratio in the first approximation is calculated as 2* cos(36°). We can approximate cos(36°) using the summation provided in the problem statement. Once we have a sufficiently precise approximation of cos(36°) given our user's allowed error, we multiply this approximation by 2 to get our approximate golden ratio.

Meanwhile, for the Fibonacci sequence, the while loop only contains the sum of our two consecutive digits and the reassigning of variables to continue the loop. The calculation of the golden ratio occurs at our exit condition in the while loop statement and once we are able to exit the while loop, we calculate our final golden ratio and print it to the number of decimals they specified their error to be. This is done by using the log10 function and absolute value function to obtain the power of the user's input error.

1.4 Discussion

This problem was challenging at first since it was my first time using loops. The way in which a programmer must think is very different from any traditional form of thinking which I have previously had. Now looking at it, the two loops are pretty simple and I'm glad that I am understanding the concept behind them. My only issue is that, on occasion, my estimation of the two golden ratios remains the same even beyond the specified number of decimal places. I am not entirely sure how to correct this, but it is especially noticeable for smaller and smaller error values.

2 Permutation Calculator

2.1 Introduction

Permutations are widely used in statistics to calculate the number of ways to arrange a group 'n' into 'r' sized groups. In this problem, we create a calculator to compute such analysis. This is typically done with a formula which involves factorials; thus, the challenge for this problem is to avoid using the factorial function in Matlab.

2.2 Models and Methods

If we look carefully at the formula provided and attempt some sample situations, we can see that many of the values in the factorial end up cancelling out. By cancelling out these terms, we can simplify our problem substantially. For example, if we have n = 7 and r = 3, we have $\frac{7!}{4!}$ which can reduce to $7 * 6 * 5$. This simplification can be applied to Matlab by using a starting condition in our for loop.

2.3 Calculations and Results

The script first takes in values for 'n' and 'r' and ensures their validity by checking that neither is a decimal or negative value as neither of these inputs works in a factorial situation. The script then begins a for loop starting at (n – r + 1), which is the smallest value we begin multiplying by after our cancellation, and ending at n. The for loop then incrementally multiplies the values in this bounded region until we reach our end condition where it then prints the final value. An if statement is also inserted for scenarios where the user may input an 'n' value which is smaller than the 'r' value. Using our cancellation, we create an error which prints '-0' in these scenarios, so the if statement corrects this to output 0.

2.4 Discussion

My initial thought for this problem was to create two for loops, one for the denominator and another for the numerator and then divide the two values. But, after a few problems on pen and paper, I was able to find a simplification which could be implemented into a single for loop. This simplification has one flaw by printing '-0' but it was later corrected. The cancellation greatly simplified the code and is robust. This problem again was a challenge as I am not entirely used to thinking like a programmer, so the idea of replacing an old value with a current value was difficult to grasp and required a lot of work on paper prior to starting the code.

3 SIR Simulation of the Spread of Disease

3.1 Introduction

This problem uses differential equations to model the spread of disease. These differential equations can then be modeled over time using euler's step method. The three equations are dependent on one another and evolve over time as the population moves from susceptible to infected to recovered. Through these equations, we are trying to plot the number of infected students over time and find at which time we have the largest number of infected students.

3.2 Models and Methods

For starters, we must establish our constants and all the given values in the problem statement. Then, using Euler's step method, we can set up the three equations for each of the populations of students (susceptible, infected, and recovered). Euler's method: $y_{n+1} = y_n + \Delta t * f'(x_n, y_n)$ Our equations are then represented as the following three relationships:

$$S_{k+1} = S_k - \Delta t * \beta * S_k * I_k$$
$$I_{k+1} = I_k + \Delta t * (\beta * S_k * I_k - \gamma * I_k)$$
$$R_{k+1} = R_k + \Delta t * \gamma * I_k$$

We can then use these equations in a for loop with the initial and end states at t = 0 and t = 20 and a step size of 0.1 as given in the problem statement. This loop finds the next value for each population and then resaves the value as the old value for the next cycle in the loop. So lets say at t = 0, I_k = 1 and I_{k+1} = 1.132 so then we store I_k= 1.132 for the next cycle to continue. This is done for each of the populations over the course of the timeline indicated. One caveat is that we are trying to find the maximum number of infected students throughout this time. To do so, we will need at least 3 reference points. These points will be comparing the middle point to see if it is greater than the previous point and greater than the next point. If this happens, it means that the middle point will be the maximum because that point will be where the upwards trend of infected students ends and begins to decline. Within the for loop, we have the 'hold on' function which plots each population of infected students at every value of t onto a single plot which is shown further below.

3.3 Calculations and Results

In order to find the maximum value of infected students in our timeline, first it was useful to plot out the function. Once this was done, we could see that there is a maximum that does not occur at the either of the extremes, so we know that at our maximum, the points one time step behind and one time step ahead would both have to be less than the value at the current time. Because of this, we will have to store three values of the infected students at any given time, and compare the middle value at every step to the other two values. We use an if statement to check for this scenario through the following line:

```
   if (i_k < i_k_1) && (i_k_1 > i_k_2)
       fprintf('The maximum number of infected students is %d at t = 
%.1d\n', floor(i_k_1), t);
   end
```

This line will print the maximum value once it is found and ignore all other points. Then, the following plot is produced once the for loop ends:

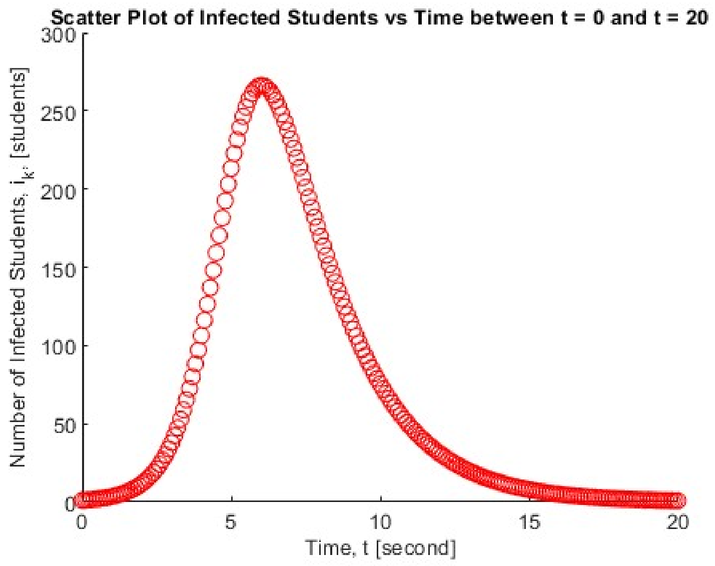

Figure 1. Scatter plot solution for the system of differential equations representing the number of infected students

And we print:

```
The maximum number of infected students is 266 at t = 6
```

From figure 1, we can see that this is an accurate estimate of where our maximum occurs, and on the actual graph from matlab, we can confirm that this is indeed the position and value at our peak. The graph is what we could have expected the function to look like simply from the physical representation of how disease spreads; while at first a lot of students become infected, we reach a maximum and gradually, there are fewer and fewer students who can be infected, lowering our curve just as we see in figure 1.

3.4 Discussion

This problem modeled differential equations using Euler’s step method. The plots of the three relationships were very interesting and were included in my code as a comment. Once we were able to find the appropriate equations, the functions were simple to implement. The greatest challenge was understanding how to use the ‘hold on’ function to plot all our points together. Once I was able to do this, I implemented the marker style of a solid red circle to make my scatter plot uniform. Another challenge was being able to find the maximum since, at first, I was only storing 2 values of the infected students just like the other populations. Once I concluded that three values were necessary, it was slightly difficult to incorporate this without adjusting the previous code. All in all, this problem was very interesting and the final plots we produced were especially intriguing as I was able to see the differential equations in action and how they related to one another.