Adrian Loekman
UID: 105785820
CEE/MAE M20
October 16, 2021

HOMEWORK 3

1. Golden Ratio

1.1. Introduction

The goal in this problem is to take an uncertainty error as an input, then use the Taylor Series of 2 * cos (36^o) to calculate the golden ratio to an error smaller than the uncertainty error. Then, calculate the golden ratio using the Fibonacci sequence until the ratio has the same precision as the Taylor Series (i.e. the error between the two ratios is less than the uncertainty error). Print the ratios and the number of Fibonacci terms required to achieve the result.

1.2. Model and Methods

First, we need to solicit an input from the user. This input is the uncertainty error as stated in the problem. For example, the input could be 1e-10. Then, we can write placeholders for variables that will come up in the while loop such as the value of the new term, cos(x) term, iteration n, and a value for error.

Next, the method of golden ratio using 2 * cos (36^o) is calculated. Note that the angle in the Taylor Series formula is in radians, thus we can use the built-in function `deg2rad` or multiply the angle by pi/180.

This next step is the most substantial because we will calculate the summation. To do that, a loop must be used. But since we do not know the range of the summation, we will need a while loop with conditions where the error of the consecutive terms is less than the uncertainty error. Inside the loop, we can redefine the new term as the old term because we will calculate the new term, which is just calculating the equation in the summation with n = 0. Then, add the iteration n+1. We can calculate the value of cosx which is the previous cosx values added with the new term. This is how a summation would look like in loops. Finally, calculate the error of the new term with the old term. This loop will go on until the error of the new term and old term is less than the uncertainty error. We can get the golden ratio by multiplying the final cosx term with 2. This is the first golden ratio.

The next part of this code is the Fibonacci Sequence. We can set up the first two Fibonacci terms 0 and 1 along with the number of terms, m, starting with two since we wrote out the first two terms of the sequence. Also, put a value for variable `golden2` because this will be used in the while loop to find the golden ratio from Fibonacci Sequence.

The while loop condition is that the ratio from the previous part with this part is larger than the error, then the loop proceeds. This is done to get the most precise numbers from both ratios. The next term in the Fibonacci sequence is gotten from adding the previous two. Then, calculate the ratio which is term(n) / term(n-1) and add the iteration by 1. Set f2 as f1 and fib as f2 so that the term continues in the Sequence. Lastly, calculate the eror

from the ratio in part b with the most recent golden ratio from the Fibonacci sequence. This while loop will stop as soon as the error gets smaller than the uncertainty error. In other words, the ratio from both approaches get to a very small difference.

The output is basically printing the ratios from each part and print the number of Fibonacci terms needed to get to the value close to the previous part.

```
fprintf('The ratio of 2*cosd(36) : %.17f\n', golden1);
fprintf('The ratio of the Fibonacci Sequence : %.17f\n', golden2);
fprintf('The number of Fibonacci Terms Needed : %2d\n', m);
```

The ratios are floating point numbers decimals), thus we need to use %f. I use 17 decimals just to see how the decimals of both ratios are different after the uncertainty error. The terms in the Fibonacci sequence is an integer, thus I use %d.

1.3. Calculations and Results

When the program is executed and 1e-10 is set as an input, we can get an output

```
Error (e.g. 1e-10) : 1e-10
The ratio of 2*cosd(36) : 1.61803398874989490
The ratio of the Fibonacci Sequence : 1.61803398867044312
The number of Fibonacci Terms Needed : 27
```

We can see that the ratio is in fact the golden ratio and the value of the decimals change after the 10^{th} digit. This is in accordance with the error set to the negative tenth power. Let's try a smaller uncertainty error

```
Error (e.g. 1e-10) : 1e-4
The ratio of 2*cosd(36) : 1.61803399401937331
The ratio of the Fibonacci Sequence : 1.61797752808988760
The number of Fibonacci Terms Needed : 13
```

Since the uncertainty error is a larger number, the ratios now become less precise and the number of terms needed to converge to the ratio is now less than previous attempt. As suspected, a smaller uncertainty error is going to give an output of a more precise output with more Fibonacci terms. However, when the uncertainty error is 1e-16 and smaller, the terms from the Fibonacci sequence does not increase and gets stuck at 42.

```
Error (e.g. 1e-10) : 1e-36
The ratio of 2*cosd(36) : 1.61803398874989490
The ratio of the Fibonacci Sequence : 1.61803398874989490
The number of Fibonacci Terms Needed : 42
```

The results from both ratios now are very similar up to the 17^{th} decimal, but they never become the exact same number.

1.4. Discussion

The last sentence from section 1.3 is very interesting to me. I believe that the number of decimals from both ratios will never be the same because the method of calculation is different, but they will be very similar. However, MATLAB can only store so much number until it is no longer able to. I believe this is called the double precision of MATLAB where it has a limit to how many digits MATLAB can store until it does not recognize the difference anymore because the difference occurs at a very small decimal number.

The results from both ratios are similar, in fact they are the same digits until the uncertainty error. The error is gotten from the different approach taken to get the golden ratio. In general, both methods can be said as a valid method to obtain the golden ratio. But, we must be very careful to set the error so that we know what to expect when the digits are not as similar to other approaches.

2. Write a Script for Permutation

2.1. Introduction

This goal in this problem is to calculate the mathematical expression of Permutation P(n,r), which is n! divided by (n-r)!, given inputs of integers n and r. The key question in this question is to create a code that proceeds like a factorial without using the built-in MATLAB function, factorial.

2.2. Model and Methods

The first part of this code is having n and r as inputs.

```
n = input('input n: ');
r = input('input r: ');
```

The important note from this is that since we will calculate the factorial from n and n-r, we will need n and r to be an integer. In order to do that, we need a while loop so that as long as the inputs are invalid (i.e. not an integer) then MATLAB will prompt the user to input an integer in order to proceed. In addition to that, factorial could not be negative, thus n and r must be a positive integer and n must be larger than r otherwise n-r would return a negative value.

```
while n < 0 || (mod(n,1) ~= 0)
while r < 0 || (mod(r,1) ~= 0)
while n < r
```

the mod used in this while loop shows that if the input n and/or r is divided by 1, it should return no remainders. Thus, in this case, if the input divided by 1 returns a remainder, this means that the input must be a decimal/fraction. Decimals and fraction could not be set as inputs for factorials.

From the problem statement, P(n,r) = n!/(n-r!). So I set up a method where I calculated the factorial of the numerator and denominator separately, then I can divide them to get P(n,r). This uses for loops

```
num = 1;
for i = n:-1:1
    num = num * i;
end
denom = 1;%set the last integer for any factorial, 1
for j = (n-r):-1:1 % from value n-r to 1
    denom = denom * j; % multiply the factorial by i
end
```

The values of num and denom is equal to 1 because for each factorial the last multiplier is 1. The for loop starts at n for the numerator and n-r for the denominator to 1 decreasing by 1. The loop would output a new value for num/denom* each I which works as the factorial calculator.

On the output, we can print the final result through dividing num / denum.

```
fprintf('Value of Permutation is P(%d,%d) = %d\n', n, r, pnr);
```

Since all the values here are integers, we will use the notation %d for all variables.

2.3. Calculations and Results

When the code is executed, the output on the command window shows P(n,r) = the result from the calculation of factorials. First, we can take a look when the outputs are false then reinput the values.

```
input n: -2
input r: 3.3
Invalid n, please reinput
input n: 2
Invalid r, please reinput
input r: 4
n cannot be less than r
input n: 7
Value of Permutation is P(7,4) = 840
```

When the input is not an integer, the code will prompt the user to reinput the values until it gets a value that satisfies the conditions. When the inputs are satisfied, we can get the value from the permutation. Let's try a large number like 1000 for n and 0 for r.

```
input n: 1000
input r: 0
Value of Permutation is P(1000,0) = NaN
```

This is indeed a very odd output since we know that the result should be 1000!/1000! That is equal to 1.

2.4. Discussion

The output of the code is basically the factorial of n divided by the factorial of n-r. The for loops in the code are the method to do factorial multiplication without using the built in function `factorial`.

The discussion starts with the result when P(n,r) is P(1000,0). If we calculate this mathematically, we wouldn't even need a calculator because this would be 1000!/(1000-0)! which is equal to 1. However, as we've seen above, the output when n = 1000 and r = 0 is NaN, or not a number. The reason behind this is the fact of the for loop when calculating the 1000 factorial. As we may know, 1000! = 1000 * 999 * 998 * … * 2 * 1. This results in a value with 2568 digits and 249 trailing zeros (https://coolconversion.com/math/factorial/How-many-digits-are-there-in_1000_factorial%3F).

The number of digits provided in that is too much for MATLAB to handle. So, instead of a value, MATLAB outputs the value Inf(infinite) for the value of 1000! This is also the same with the denominator which is also infinite. When infinite is divided by infinite, MATLAB doesn't have the ability to calculate it since MATLAB calculates computationally instead of analytically. So, the result is NaN from MATLAB, but the analytical answer is 1.

3. SIR Simulation of the Spread of Disease

3.1. Introduction

The goal in this problem is to analyze the equations for the SIR model and write down the discretized governing equations using forward Euler methods. Next, using the constants and initial values given on the problem, we can write a program that solves the system of differential equations using Euler methods and the discretized equations. Plot the graph of the infected and figure the max infected and when it occurs from the graph and data.

3.2. Model and Method

In this problem, we have a system of differential equations. MATLAB can solve these equations; however, it could not solve the equation using this form. We will need to change the equations into discretized form. Only from there, we can solve the differential equations using MATLAB. From the equations shown in the problem, we have 3 discretized form which are

S(k+1) = Sk + delta_t(-beta*Sk*Ik)
I(k+1) = Ik + delta_t(beta*Sk*Ik – gamma*Ik)
R(k+1) = Rk + delta_t(gamma*Ik)

Then, we can use the forward Euler method to calculate for each iteration.

From the given constants and initial values in the problem statement, we can start coding by writing down the given constants (beta, gamma) and the initial conditions (S(0),I(0), and R(0). The time range will be between t_start = 0 and t_final = 20 with time step delta_t = 0.1.

Next, we can proceed with the loop. The loop used in this code uses for loop since we know the number of iterations needed. Set the initial conditions into a new variable s_old, i_old, r_old. Inside the loop, each (k+1) will now be defined as s_new, i_new, or r_new. This value will be updated from its iteration. After getting the _new values, update this by changing to the new to variable _old.

```
i_old = i_new;
s_old = s_new;
r_old = r_new;
```

Then, we can plot the data which is the time t vs i_old.

```
plot(k,i_old,'*r')
hold on % use hold on so each point is set on the same plot
grid
```

'*r' is to make each data plotted as a star with color red. Then, we use hold on so that the next data can be plotted to the same graph.

Lastly, we can find the maximum number of infected student and when it occurred. Since we are not using an array, we would need t check each point and relate it to the other points. In my case, I used an if function to relate each point of infected students, to the next point. So, I used another variable maxI

```
if i_old > maxI
        maxI = round(i_old); % round maximum infected students to
nearest integer
        tmax = round(k); % round the t when max infected students
to nearest integer
    end
```

The way this statement works is that as long as the next point in the y-axis(infected students) get larger, the if statement will proceed and the value of maxI will always be updated. We know that it will stop updating when i_old is less than maxI. This is the peak since that the next data will be less than the previous data. Thus, we get the maximum infected with its time, k.

The output itself will output the plot of the infected students with the maximum infected students and when it occurred. This is rounded to the nearest whole number.

3.3. Calculations and Results

When the code is run, a plot with the maximum infected students with its time is also being output in the command window

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

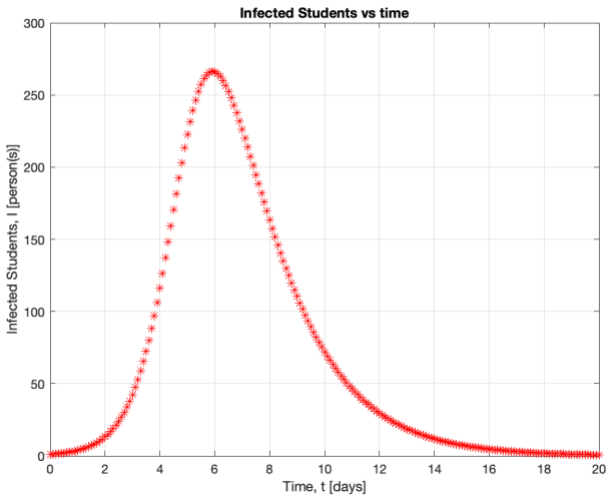

Figure 1 Infected Students (I) as a function of time (days)

We see that the plot, Figure 1, is the time from 0 to 20 in the horizontal axis and the infected students I(k) is in the y axis. From the plot, we can approximately locate the maximum infected students is at 6 s with the value around 266 using an if else statement which calculates the maximum value of the data.

3.4. Discussion

From this problem, we get to work by hand to perform a conversion from continuous ODE to discretized ODE in order for MATLAB to calculate and solve the ODE computationally. If we pay close attention to the plot of the infected students. The number of infected is low at low t times but gradually become higher until a peak which then will go down and eventually go to 0. This is because as time goes, the infected get recovered so the number of infected goes down.

If we try to graph all 3: susceptible, infected, and recovered, we can actually see that the number of susceptibles are decreasing since some are infected and then recovered, the remaining susceptibles are those who are never infected. The recovered rises from 0 to a very close to 700 (the total of students) because now all has been infected and recovered.

This problem uses the for loop well and also taught me to draw a plot even without the help of arrays. The value of the maximum infected students is obtained from the data. A method that is used is to check each data and compare it with the data after the data mentioned. This way, if the next point is less than the previous point, we now know that it is potentially the peak. We know for sure because it is indeed the global maximum.