Matthew Lacaire
UID: 205288439
CEE/MAE M20
October 24, 2020

Homework 3

1. Golden Ratio

a. Introduction

The first homework problem involves tasks involving calculating the "golden ratio" in three different ways- an "exact" explicit calculation (2cos(36)), a Taylor series approach, and a Fibonacci series-based approach. A tolerated error between the exact value and approximate values is inputted by the user, and the program uses the algorithms within to calculate approximations to that accuracy. Then, the approximate values are actually calculated, and the results are displayed along with the number of iterations to arrive there.

b. Model and Methods

The program first begins with the input of the "uncertainty", the maximum tolerated error value. I put limits on the input:

```
if (uncertainty <= 0 || uncertainty > 0.1)
    error('Invalid Input.\n') %Prints error if inputs are
invalid.
End
```

Simply for cleanliness (and to emphasize that uncertainty should be a positive number), I exclude all negative numbers. I exclude zero, because this is impossible (the exact value will never equal the approximate value for a finite number of calculations). I also exclude inputs larger than 0.1, because error is usually not much larger. If error is an integer (or larger than 0.1), the program would not make any sense, as the difference between approximations and exact values should be small decimal values. Also, through experimentation, the program doesn't really give logical results for uncertainty > 0.1, because so few calculations are performed.

After the algorithms are performed (see below), I output the number of times each calculation was iterated, and add fifteen digits of decimal precision.

```
%% Output; Output calculations
fprintf('Approximate Taylor Series value: %.15f, done after %.0f
Taylor terms.\n',result,iterations);
fprintf('Approximate Fibonacci Series value: %.15f, done after
%.0f Fibonnaci digits, with the last being
%.0f\n',fibratio,nthterm, newval);
```

The algorithm details are explicitly detailed below.

c. Calculations and Results

Two algorithms form the core of this program: the Taylor algorithm, and the Fibonacci sequence algorithm.

In the Taylor algorithm, I define the exactVal (the "exact" value we're comparing against), a boolean, and two variables instantiated to 0:

```
exactVal = 2 * cos(deg2rad(36));
continueLoop = true;
iterations = 0;
result = 0;
```

The boolean is a type of flag variable, and is used in the main loop of the algorithm. The "iterations" variable describes the number of calculations performed, and also has a role in calculation. "Result" is the actual sum of n Taylor terms, which is compared explicitly with exactVal. The loop is below:

```
while(continueLoop == true)
    prefix = 2*((-1)^iterations);
    xval = ((deg2rad(36))^(2*iterations));
    factorialpart = factorial((2 * iterations));
    result = result + (prefix * xval) / factorialpart;
    iterations = iterations + 1;
    if (abs(result - exactVal) <= uncertainty)
     continueLoop = false;
    end
End
```

To explain, while continueLoop (as defined above) remains true, we compute the terms of the Taylor series one by one. The "prefix" is the 2*(-1)^n part of each term (multiplied by two because we're looking for 2cos(26) and not cos(36)). The xval is the x^2n portion, and the factorialpart is the denominator of each term. This is something I did just to break up the code and make it easier to read. I combine all of these and add this new term to the previous result in the fourth line within the loop. Note that in the first loop (when iterations = 0), we begin with n = 0, as per the Taylor series. At the end of the code, iterations is increased by one. In this way, after the first term is calculated (with iterations "n" = 0), we can now use iterations (now equal to one) to describe the number of calculations. This is why iterations describes the *total number of terms calculated.*

The nested if loop checks for the absolute value of difference between the calculated result and exact result for iterations equal to some value. If the difference is smaller than the tolerated uncertainty, the loop stops, as continueLoop is declared false and the while loop is no longer satisfied.

The second algorithm involves a Fibonacci sequence algorithm. As with before, I define a flag variable "continueSeries" and set it equal to true. The initial conditions and setup can be a bit confusing, so I put them here:

```
continueSeries = true;
nthterm = 2;
    valminustwo = 0;
    valminusone = 1;
```

Essentially, the "nthterm" tells us which maximum Fibonacci term is used in calculating the golden ratio. It is set default to 2 because we assume the series begins (0,1,1) and that the second term is 1. The second two variables here represent the first two values (0,1) and later are redefined in the loop to actually be two numbers third-from-highest, and second-from-highest, respectively. To explain further, look at the algorithm:

```
while(continueSeries)
  newval = valminusone + valminustwo;
  fibratio = newval / valminusone;
  nthterm = nthterm + 1;
  valminustwo = valminusone;
  valminusone = newval;
  if (abs(fibratio - exactVal) <= uncertainty)
    continueSeries = false;
  end
End
```

The logic at the end of the loop is almost identical to that in the first loop- it checks if the error is small enough between exact and approx. values, and then it decides to continue calculating if not. The beginning is a little more complicated.

Essentially, the program creates a "newval", which is the sum of the previous two values. This is the new, larger fibonacci value, defined as a sum of the previous two. Then, I actually calculate the fibratio for this value divided by the value immediately before. I add one to "nthterm", to express which term of the fibonacci series is now being calculated, and to keep track of which number term this particular loop corresponds to. In a final step to prime the program for the next loop, I redefine the value two to the left (valueminustwo) as the value one to the left of the larger number, and I redefine valueminusone as the newvalue. This essentially means that I'm "shifting" the sequence rightward by one digit, and I'm now ready to calculate a new digit by the logic at the beginning.

The following is some example code with tolerance 1e-10:

```
Input your acceptable uncertainty value
1e-10
Approximate Taylor Series value: 1.618033988734122, done after
6 Taylor terms.
Approximate Fibonacci Series value: 1.618033988670443, done
after 27 Fibonnaci digits, with the last being 121393
```

d. Discussion

I find it extremely important to emphasize that my Fibonacci series is indexed such that the first three terms are (0,1,1). Thus, for error (1e-10), I find that the Fibonacci series is accurate after the 27th digit (which is 121393). Others may consider this the 26th digit, but I simply elected to begin my counting from 0,1,1.

Also worth noting is that the "exact" value 2cos(36) is not exact. It's an approximation as well, considering that MATLAB and computers in general have finite computing power. It's likely that this value is also generated from Taylor series, but probably from many more terms than the approximations.

2. Guess the Number

a. Introduction

This is a program in which a random number from 1 to 50 is generated, and the user has five tries to guess the number. For each incorrect try, the program tells the user to guess higher or lower depending on the specific guess. If the user wins or loses (in other words, gets through the game in its totality), they are prompted on whether to try again with a new random number or leave the game.

b. Model and Methods

The code is constructed such that there is a while loop with two nested if loops inside. As with the first problem, I use a flag variable gameOver as a condition. Before the while loop, I do the following:

```
tries = 0;
value = randi([1,50]);
gameOver = false;
```

I instantiate "tries" to zero, which is the number of times the player has guessed. Also, the "value" is assigned a pseudorandom number from 1 to 50. Lastly, boolean "gameOver" is defaulted to false, considering the game, well, isn't over.

The while loop works as follows: as long as gameOver is false, the program continues to loop the while statement. Considering that #tries, and the randomized value have already been instantiated, we can immediately start taking input:

```
while(~gameOver)
    guess = input('Input your guess\n');
    if (guess < 1 || guess > 50 || guess ~= floor(guess))
        guess = input('Invalid input. Try again.\n');
    end
    tries = tries + 1;
```

The while() argument means the loop will continue while gameOver is false. The "guess" value is taken. If it's not within 1 to 50 or is a decimal (its floor doesn't equal its value), we call it invalid input and ask the user to try again. I could write a while loop here

to allow the program to check multiple times for incorrect inputs, but I'm not sure that's necessary. The program then automatically tallies one more try (adds 1 to the try value).

Then, in the bulk of the section, we have:

```
    if (guess == value && tries <= 5)
        fprintf('Nice Guess, you found my number!\n');
        winresult = input('Do you want to play again?
(Y/N)\n','s');
        if (winresult == 'Y')
            tries = 0;
            value = randi([1,50]);
        elseif (winresult == 'N')
            gameOver = true; %gameOver is now true. In other
words, the loop is ceased.
            fprintf('Game ended.\n');
        else
            error('Invalid input.\n');
        end
    elseif (guess ~= value && tries == 5)
        fprintf('Out of tries, better luck next time.\n');
        loseresult = input('Do you want to play again?
(Y/N)\n','s');
        if (loseresult == 'Y')
            tries = 0;
            value = randi([1,50]);
        elseif (loseresult == 'N')
            gameOver = true;
            fprintf('Game ended.\n');
        else
            error('Invalid input.\n');
        end
```

To summarize this algorithm, the program first checks to see if number of tries is less than or equal to 5 (it's a while loop, and will keep going forever without limit unless some value is checked), and that the guess equals the value. In this case, the player is commended for winning, and asked if they want to keep playing. Then, they can enter Y or N. Y resets tries to zero and generates a new random number, allowing the user to play again. If N is input, gameOver is made true and the loop ceases. A game over message is printed as well.

Second, if the guess isn't equal to the value and we're on the fifth try, the game is logically over. In the same logic as the last paragraph, the program asks for a Y or N on continuing the game. A slight bit of this code is missing; see next section.

c. Calculations and Results

The code doesn't necessarily involve any complicated calculations. The only real "algorithm" that can be described is seen in the statements comparing the guess to the actual value, immediately below the code above:

```
    elseif (guess < value)
        fprintf('Go higher!\n');
    elseif (guess > value)
        fprintf('Go lower!\n');
```

In this snippet, you can see that if guess is less than value, the program hints the user to guess higher. The program does the opposite if the guess is more.

The following is an example of console-user interaction:

```
Input your guess
25
Go higher!
Input your guess
35
Go higher!
Input your guess
45
Go lower!
Input your guess
42
Go lower!
Input your guess
41
Nice Guess, you found my number!
Do you want to play again? (Y/N)
```

On the last try, I find the number. A similar sequence continues if the user elects to play again.

d. Discussion

The code itself is only 37 lines long, and is entirely within a while loop except for initial values being declared. The entire loop is run through once per input, and I consider the program to be simple and effective. I originally tried with while loops and nested for loops but have come to the conclusion that using flag variables and a minimum number of nested loops is an effective means of simplifying code.

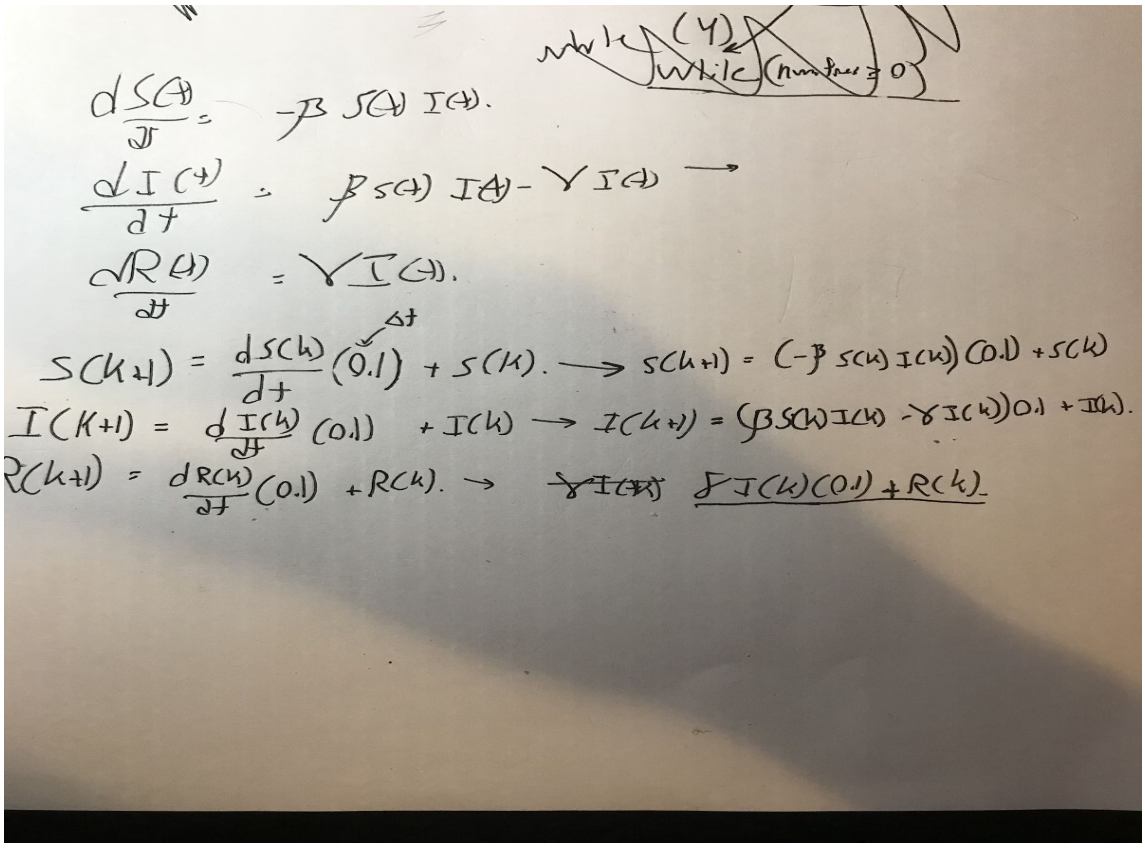
3. SIR Simulation of the Spread of Disease

a. Introduction

Using the Euler method to estimate over time, I use SIR differential equations to graph a situation with given constants in which one person is initially infected. I then graph the values, and visually determine the peak number of infected students.

b. Model and Methods

I first rearrange the equations such that they fit the Euler approximation format. See



Then, I substitute these equations into a for loop. See calculations and results section for a full explanation.

c. Calculations and Results

See large code snippet below:

```
%% Defining Variables
beta = 0.0026;
gamma = 0.5;
R = 0; %Initial R value R(0)
S = 700; %Initial S value S(0)
I = 1; %Initial I value I(0)
for t = (0.1:0.1:20)
    Snew = (-beta * S * I * 0.1) + S;
    Inew = (beta * S * I - gamma * I)*0.1 + I;
    Rnew = (gamma * I * 0.1) + R;
```

```
    if (Inew > I)
        maxI = I;
        maxInfected = round(maxI);
    end
    S = Snew;
    I = Inew; %I do the same for I.
    R = Rnew; %I do the same for R.
    plot(0, 1,'d');
    hold on
    plot(t, Inew,'d');
    hold on
end
```

In the first section, I define the constants (gamma, beta) and the initial R(0), S(0), and I(0) values. Then, I set up a for loop that starts at 0.1, increments by 0.1, and ends at 20. It starts at 0.1 and not 0.0 because at 0.0, the values are the initial values. Starting at t = 0.1, the results are approximated by the euler formula.

I then plug in the equations I solved for on paper, and get Snew,Inew,Rnew values. I then define these as S, I, R in order to prepare for the next loop that will build off of these values and repeat the process. I plot (0, 1, 'd') to plot the point (0, 1) for (t, I(t)), since it would not be included normally as t starts from 0.1. I put "hold on" to ensure that any other plotted points don't erase those before. Then, for each loop, I print the current t, and the associated I value (Inew). The 'd' argument makes the graph points look like colorful diamonds, which simply looks nice.

Note that within the code, I insert a brief "if" statement to check if each successive I is more than the last. The underlying logic is that if a new I is greater than the I before, it is assigned to "maxI". Then, I check for the next I value. In theory, this systematically compares each value and logs the highest one until the slope turns negative. Then, I round the result as instructed and print.

Then, I simply print:

```
  fprintf('Maximum Number of Infected Students: %.0f',maxI); %I
print the value calculated above.
```

d. Discussion

In theory, I could store the I values as a part of a 1D array and then find the max among these values, but we haven't gotten to that in lecture yet. I believe this actually would be a more foolproof and efficient method than my "if" statement within the for loop for finding maxI. I just felt it necessary to mention this. Also, I'm aware that there are other differential

solvers in MATLAB including ODE45, so this Euler approximation can, in some real life situations, be avoided.