David Curry
ID: 304755606
CEE/MAE M20
May 1, 2018

Homework 4

1 The Pendulum Problem

1.1 Introduction

The purpose of this script is to create a model for the motion of a simple pendulum. This can be done by changing the differential equations governing this motion to discretized equations using the forward Euler method. After defining the initial conditions, I used both the explicit and semi-implicit Euler method to create 2 graphs of the motion of the pendulum. I also created a graph of the total energy of the pendulum to determine which Euler method was more accurate.

1.2 Models and Methods

The script first defines all the constants (*g, L, Δt)* given from the problem statement. It then creates arrays filled with zeros with the `zeros()` function for the position, velocity, acceleration, height, and energy for both the explicit and semi-implicit methods. The script also defines the time array with the `linspace` function. Some of these are shown below.

```
t = linspace(0,20,t_steps);
w = zeros(1,t_steps);
theta = zeros(1,t_steps);
```

The script then uses a `for` loop to iterate thru all the *t_steps* needed to encapsulate the full time period. In the for loop, the script defines the k+1 values in the position and velocity arrays based on the explicit and semi-implicit euler methods. It also defines every k value in the acceleration, height, and energy arrays based on the given equations. The explicit method is shown below as an example.

```
for k = 1:t_steps - 1
      w(k+1) = -delta_t*(g/L)*sin(theta(k)) + w(k);
      theta(k+1) = delta_t*w(k) + theta(k);
      acc(k) = -(g/L)*sin(theta(k));
      h(k) = L - cos(theta(k))*L ;
      Energy(k) = g*h(k) + (.5*(L*w(k))^2);
```

The script then loops uses the following lines of code to continue looping thru the array. This sets the old value of the position and velocity equal to the new value and redoes the loop.

```
w(k) = w(k+1);
theta(k) = theta(k+1);
```

This semi-implicit method repeats these calculations but the position and velocity calculations are a little different. They are shown below.

```
theta2(k+1) = -delta_t^2*(g/L)*sin(theta2(k)) + delta_t*w2(k) +
theta2(k);
w2(k+1) = (theta2(k+1) - theta2(k))/delta_t;
```

This ends the `for` loop and the script moves on to plotting the graphs of this data. The script uses the `plot`, `xlabel`, `ylabel`, `title`, `legend`, and `figure` functions to accomplish this. The code for the explicit method is shown as an example below.

```
plot(t,theta, t, w, t, acc);
xlabel('Time (s)');
ylabel('Position (rad), Velocity (rad/s) , Acceleration
(rad/s^2)');
title('Explicit Angular Position, Velocity, and Acceleration
over Time');
legend('Position','Velocity', 'Acceleration');
figure;
```

1.3 Calculations and Results

The four plots created from the script is shown below.

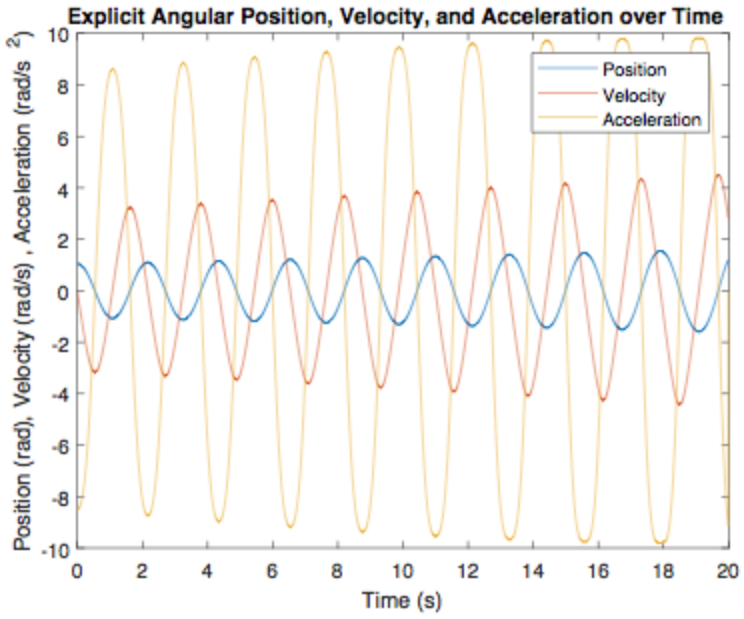

Figure 1.

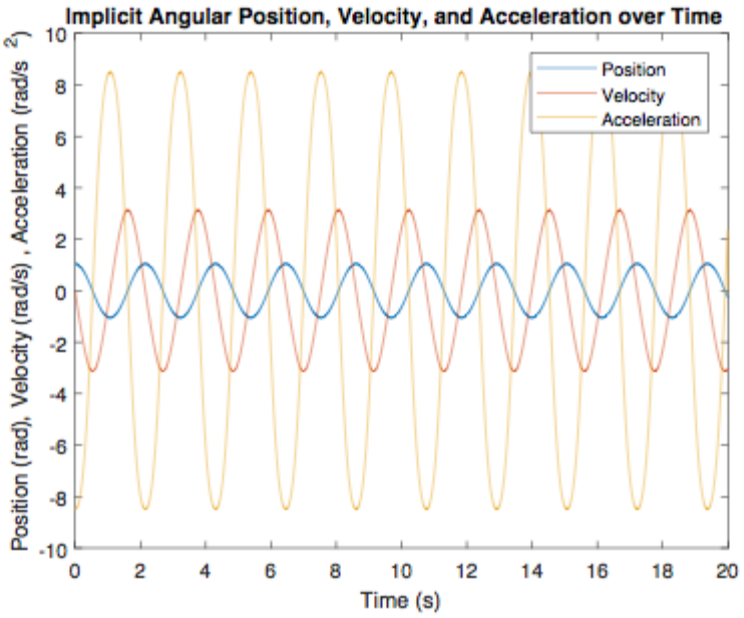

Figure 2.

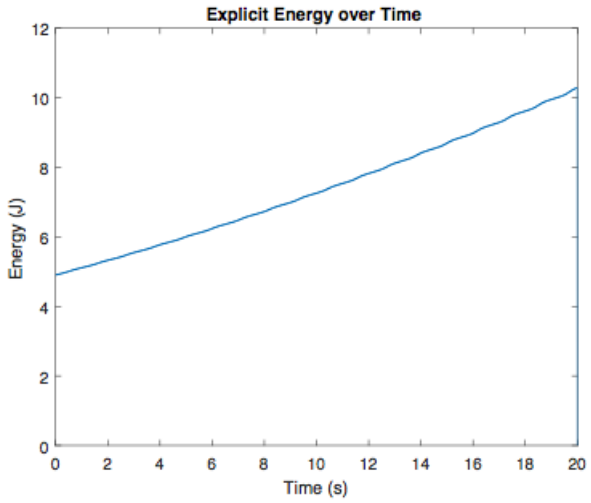

Figure 3.

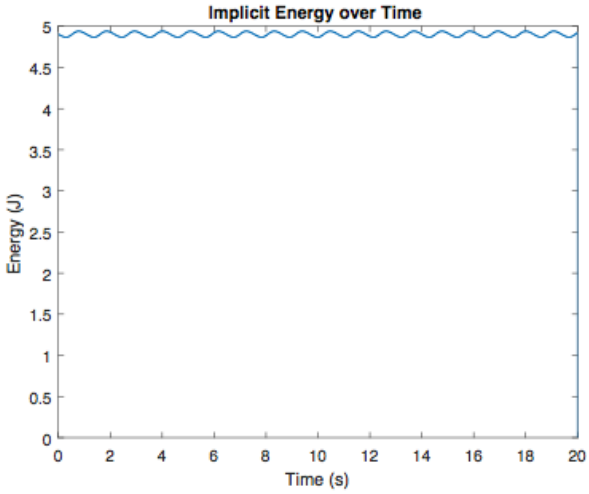

Figure 4.

Figure 1 shows that the position, velocity, and acceleration curves increase in amplitude as time goes on. The energy level for the explicit method also increases over time, shown in *Figure 3*. Contrastingly, the semi-implicit methods maintains constant amplitudes for all 3 curves, and the energy level remains constant over time.

1.4 Discussion

As shown above, *Figure 3* has an obvious upward trend in the energy curve. This means that the forward (explicit) Euler method does not conserve energy over time. This does not change by

using a smaller timestep. I tried a time step of 0.001 instead of 0.005 and printed the energy graph shown below.

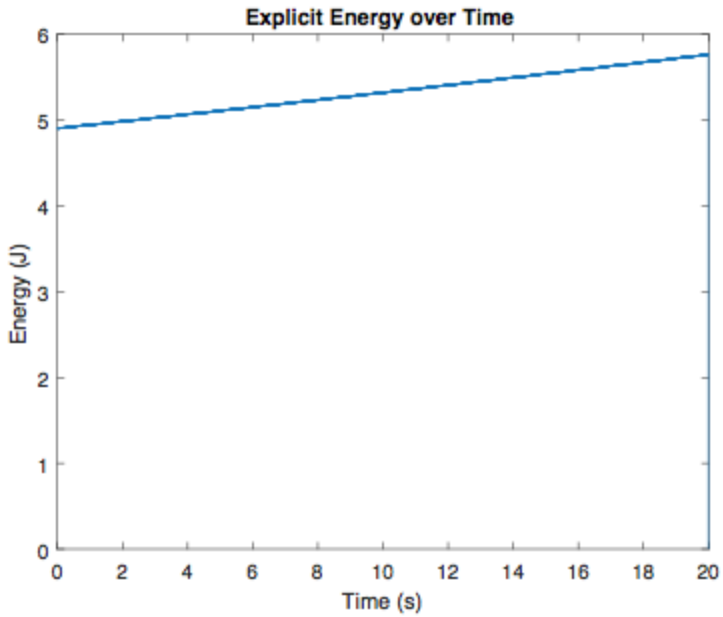

Figure 5.

Figure 5 also has an upward trend in energy, so I can conclude the explicit Euler method does not conserve energy for any time step. In contrast, the semi-implicit method does conserve energy. This is shown in *Figure 4*. The energy level does fluctuate slightly around 5 Joules, but this is only because the omega value in the energy equation is a function of sine. There is no overall upward trend in the energy, so the energy is conserved.

2 DNA Analysis

2.1 Introduction

The purpose of this script is to calculate the lengths of every protein-coding segment of a section of DNA. This can be done by iterating thru the DNA segment, searching for specific start and stop codons. After finding these proteins, their lengths are stored in a separate array just for proteins. After iterating thru the entire DNA segment, the script prints out the total number of protein segments found, the average length of these segments, and the minimum and maximum length of the segments.

2.2 Models and Methods

The script first loads the DNA segment and calculates the length of this segment with the `length` function. It also defines initial conditions and creates a separate length array filled with zeros using the `zeros()` function. This is shown below.

```
load('chr1_sect.mat');
numBases = length(dna);
startpoint = 0;
numProteins = 0;
LENGTH = zeros(1,numProteins);
```

Next the script loops thru the entire DNA segment with a `for` loop. It specifically loops every 3 bases so that it only checks full codons. In the for loop, the script first checks if the startpoint has been found. If not, it loops thru the segment searching for the ATG codon. Once it is found, the startpoint is updated. This is shown below.

```
for k = 1:3:numBases - 2
     if startpoint == 0
          if dna(k) == 1 && dna(k+1) == 4 && dna(k+2) == 3
               startpoint = k;
          end
```

Now that the startpoint has been established, the code looks for the stop codon. There are three different stop codons, so I used `||` statements. Once the stop codon is found, the script calculates the length of the protein from the start and stop points. The number of proteins is also updated, and this length value is placed in the LENGTH array that was defined earlier. This is shown below.

```
else %if startpoint has been established, it finds the endpoint
     if (dna(k)==4 && dna(k+1)==1 && dna(k+2)==1)||(dna(k)==4 && dna(k+1)==1
     && dna(k+2)==3)||(dna(k)==4 && dna(k+1)==3 && dna(k+2)==1)
          length = k - startpoint + 3;
          numProteins = numProteins + 1;
          %puts the length of this specific protein in the next slot in
          %the array LENGTH
          LENGTH(numProteins) = length;
          %resets startpoint
          startpoint = 0;
     end
end
```

Finally, the script computes the average, maximum, and minimum protein lengths by using the `mean`, `max`, and `min` functions. It then prints these values with the `fprintf` function. Examples of these are shown below.

```
avg = mean(LENGTH);
maxim = max(LENGTH);
minim = min(LENGTH);
fprintf('Total Protein Coding segments: %.0f\n',numProteins);
```

1.3 Calculations and Results

After running the script, the following is printed out.

```
Total Protein Coding segments: 4388
Average Length: 85.90
Maximum Length: 1149
Minimum Length: 6
```

1.4 Discussion

From the calculations shown above, not much of DNA is actually used for proteins. Out of the 1261563 bases in this DNA segment, only 4388 proteins of an average length of 85.90 bases were found. To actually calculate the percent of bases used for proteins, I used the `sum` function to total the number of bases used in proteins and divided by the total number of bases. This calculation determined that only 29.88% of the DNA segment was used for proteins. The output shown above says nothing about which stop codons were used the most and least so I included a nested if, elseif statement like the one shown below to determine how often each stop codon was used.

```
if dna(k)==4 && dna(k+1)==1 && dna(k+2)==1
     numCodon1 = numCodon1 + 1 ;
 elseif dna(k)==4 && dna(k+1)==1 && dna(k+2)==3
     numCodon2 = numCodon2 + 1;
 else
     numCodon3 = numCodon3 + 1;
 end
```

These values were not printed, so I checked the workspace to determine how many of each codon was used. Codon 1 (T A A) was used 1448 times, codon 2 (T A G) was used 998 times, and codon 3 (T G A) was used 1942 times for a total of 4388 stop codons.

Since we disregarded any other start codons in between a start and stop codon, this may not be the most accurate results. One way to fix this is shown in bold in the pseudocode shown below.

```
if startpoint not found
     if codon is startpoint
          set startpoint
     end
else
     if codon is stoppoint
          for values between start and stoppoint
               if there's another startpoint
                    reset startpoint
```

```
                end
            end
            find length based on start and stoppoint...
        end
    end
```