EA4 Lab 3:

11/8/19

Adolphus Adams, Preston Collins, Shane Dolan, and Braden Rocio

Certification Page

We certify that the members of the team named below worked on this labusing only our own ideas, possibly with the help from the TAs and our professors,specifically in the writing of the report, the programming and the analysis required for each task. We did not collaborate with any members of any other team nor did we discuss this lab or ask assistance from anyone outside of our team, the TAs and our professors. If we accessed any websites in working on these labs, these websites are explicitly listed in our report (with the exception of websites used only to get information on specific Matlab programs). We understand that there will be severe consequences if this certification is violated and there is unauthorized collaboration.

1. Shane Dolan -  - 11/8/19

2. Braden Rocio -  - 11/8/19

3. Adolphus Adams  - 11/8/19

4. Preston Collins -  - 11/8/19

Introduction

Lab 3 seeks to study the behavior of non-linear oscillators and the effects of springs that do not obey the linearity of Hooke's law. In this lab we will look at the transients and steady state responses of forced, damped oscillators. We will also use interpolation to obtain more information from numerical solutions to differential equations. Finally, we will learn about hysteresis which allows two different responses for a forcing frequency, a phenomenon that only occurs in nonlinear oscillator systems. In Task 1 we will create an adaptive scheme to solve the oscillator system and use this new scheme with different oscillators to test and measure the solver's efficacy. In Task 2 we created response curves to see the change in amplitude dependent on the change in frequency. In Task 3, we compared the effects of solving over the range of omega from both directions. First, it was solved forward and then reverse, and the differences were observed in the hysteresis. In addition to these steps we also will check $\beta = 0$ to see if the hysteresis remains in a linear version of the equation.

Task 1:

Introduction:

Task 1 is to write equation (2) from the lab guide as a first order system for x and $y = x'$ and create an m-file (*prime.m*) to be called by our adaptive Improved Euler function from lab 2. We will adapt *yprime.m* to accept both independent variables x and y because *adapt.m* must now handle systems of equations instead of a scalar equation. We will use the following equation:

$$mx'' + cx' + kx + \beta x^3 = F_0 cos(\omega t)$$

$$y = x'$$

$$my' + cy + kx + \beta x^3 = F_0 cos(\omega t)$$

$$y' = -\frac{c}{m}y - \frac{k}{m}x - \frac{}{m}x^3 + \frac{F_0}{m}cos(\omega t)$$

In MATLAB, this looks like:

```
f = -(c/m)*y -(k/m)*x -(B/m)*x^3 +(F0/m)*cos(W*t);
```

Task 1:

Run 1 - c = 0.1, ω = 1

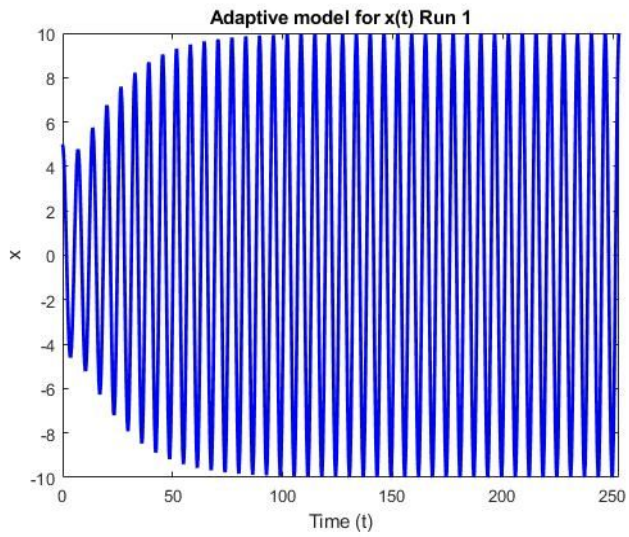

Figure 1a: *x(t) for run 1 of Task 1.*

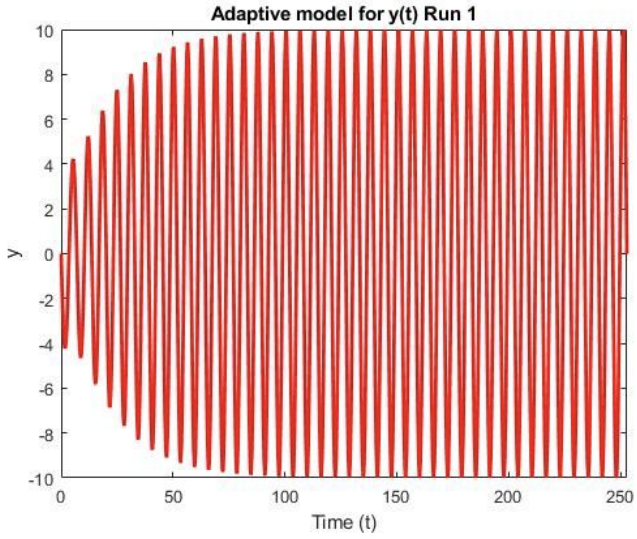

Figure 1b: *y(t) for run 1 of Task 1.*

Run 2 - c = 0.1, ω = 0.5

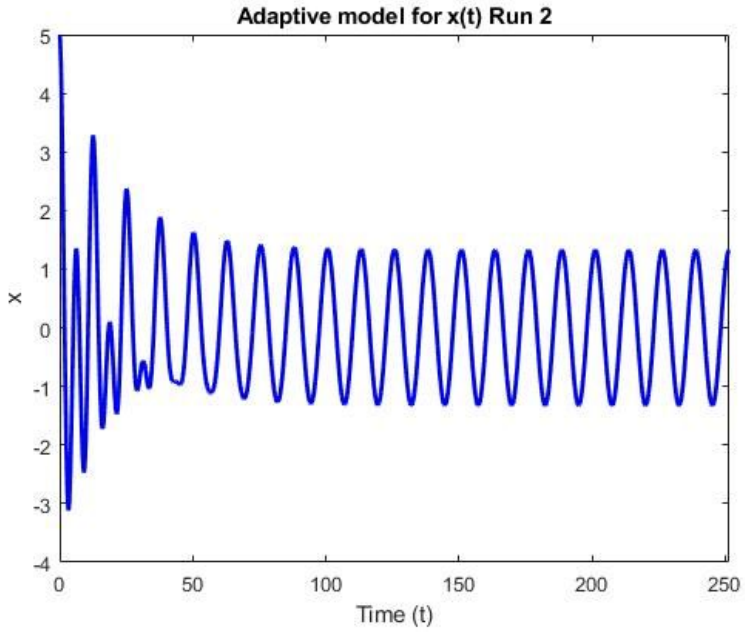

Figure 2a: *x(t) for run 2 of Task 1.*

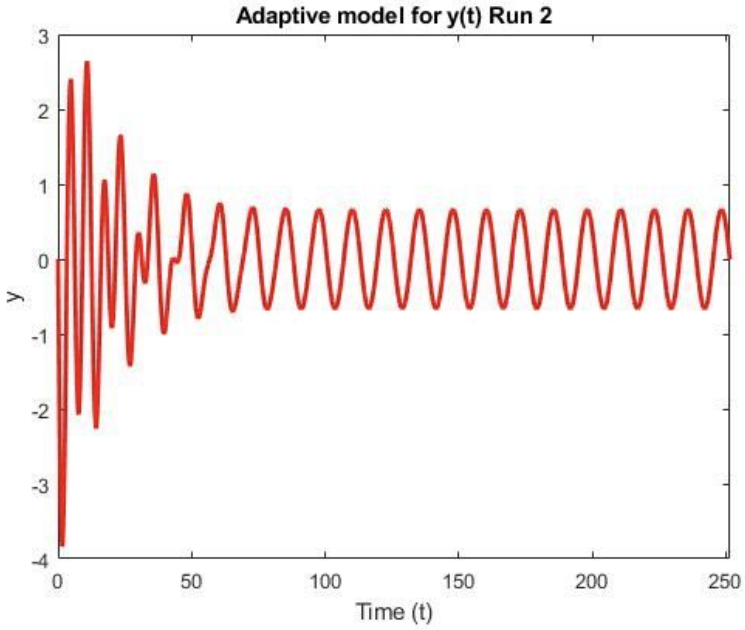

Figure 2b: *y(t) for run 2 of Task 2.*

Run 3 - c = 1.5, ω = 0.5

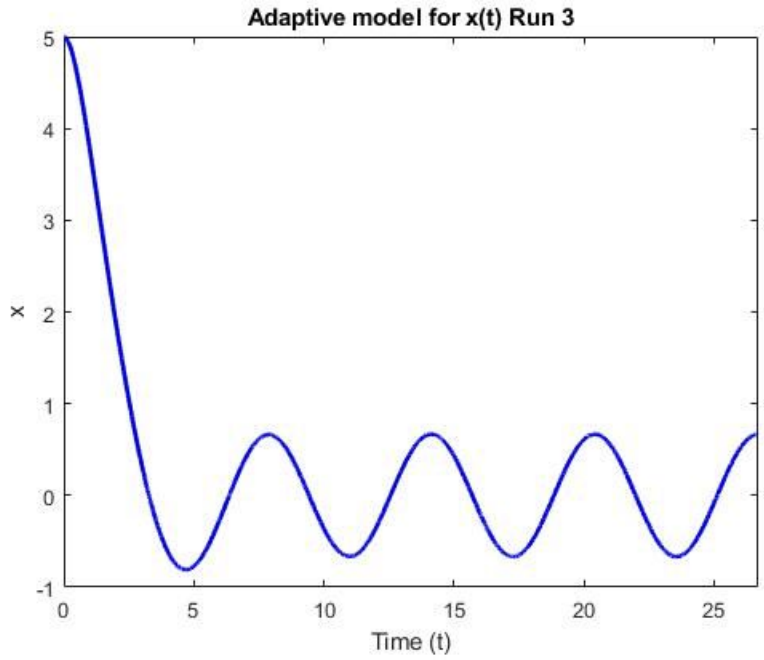

Figure 3a: *x(t) for run 3 of Task 1.*

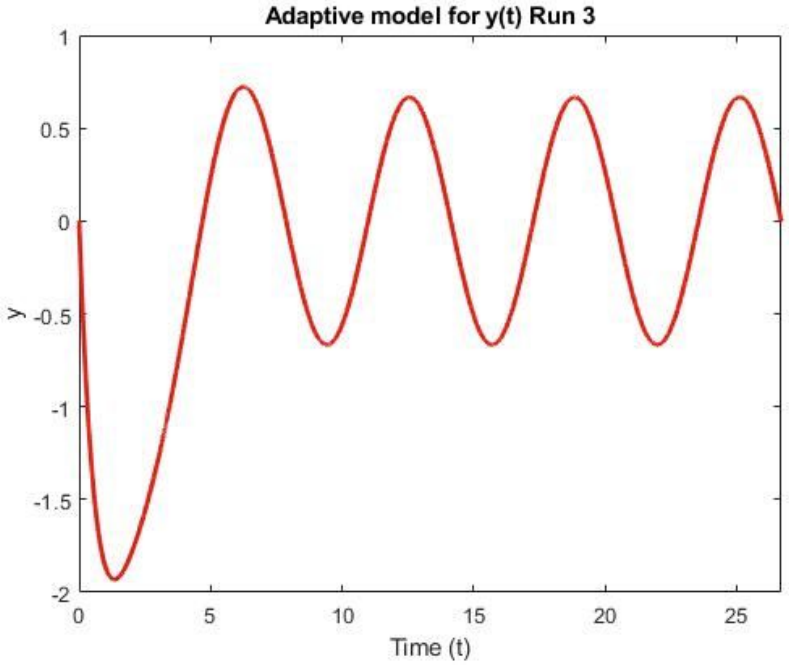

Figure 3b: *y(t) for run 3 of Task 1.*

Run 4 - c = 1.5, ω = 0.5

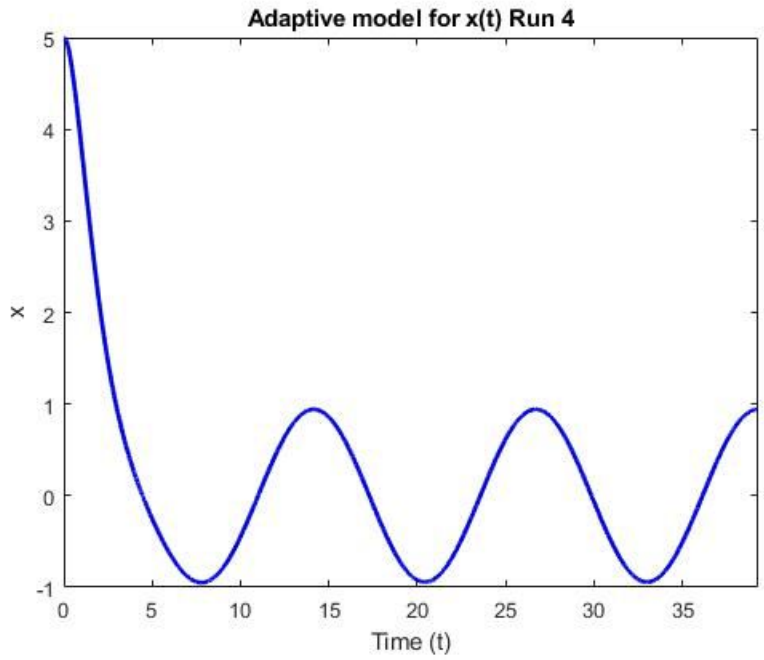

Figure 4a: *x(t) for run 4 of Task 1.*

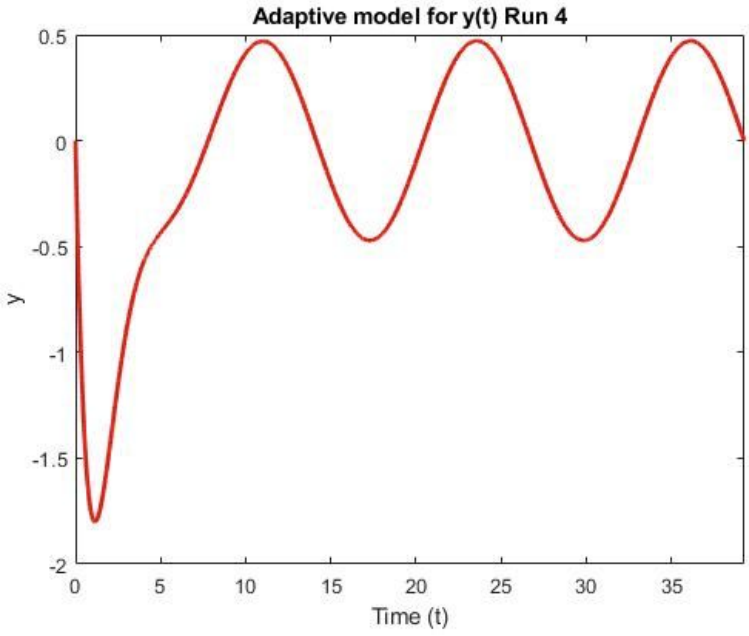

Figure 4b: *y(t) for run 4 of Task 1.*

Task 1 Runs					
	finalamp *(m)*	***cexact*** *(m)*	**Amplitude Convergence Criteria?**	$\mathbf{T_{ss}}$ (est.)	***acceptr*** *(2nsteps/count)*
Run 1	10.0000	10.0000	Met	100	0.9342
Run 2	1.3304	1.3304	Met	95	0.9343
Run 3	0.6667	0.6667	Met	8	0.9337
Run 4	0.9428	0.9428	Met	12	0.9337
Run 5	1.3282	1.3304	Not Met	100	0.9343
Run 6	1.3304	1.3304	Met	90	0.8321

Table 1: *Table 1 contains the results for each of the trials for Task 1.*

Task 1 Discussion and Analysis:

In Task 1, we first converted the equation for a forced nonlinear oscillator to a first order system and created an m-file: *prime.m* that contained global variables *m, c, k, β,* F_0 and *ω as global variables. Adapt.m* read these variables just as in lab 2. We added a counter to track the number of time *prime.m* was called and *nsteps* to track the number of steps accepted within the error tolerance, ε. Additional functionality was added to *adapt.m* to ensure that the absolute error between high and low estimates for both *x* and *y* was less that the error threshold, ε. In the previous lab, we only needed to check *y*.

```
%creating the values k0, ylow, xlow, k1, yhigh, and xhigh that will be used to
check the
% error within this equation to make sure it is less than epsilon
      k0 = prime(t(j),x(j),y(j));
      ylow = y(j)+h*k0;
      xlow = x(j)+h*y(j);
      k1 = prime(tnew,xlow,ylow);
      yhigh = y(j)+((h/2)*(k0+k1));
      xhigh = x(j)+((h/2)*(y(j)+ylow));

%checking that the error is less than epsilon and if it is, then raising h
% by a factor of hraise, adding 1 to j and then yetting y(j) to yhigh and
% t(j) to tnew in order to set the intial values for the next loop
      if abs((yhigh-ylow)/abs(yhigh)) && abs((xhigh-xlow)/abs(xhigh)) <
epsilon
      h = h*hraise;
      j = j+1;
      y(j) = yhigh;
      x(j) = xhigh;
      t(j) = tnew;

%add 1 to the value of nsteps
      nsteps = nsteps+1;
```

Figure A: *Figure showing method used to determine if the error is less than espamp. Full code in Appendix 1.*

In addition to these changes, we also added a section to perform *linear interpolation* in order to estimate the time at which x' changes from positive to negative. This piece of code creates a linear function (line) between the final positive point and the first negative point and find the time when this line intersects the x axis. This point, t_{max}, is plugged in to estimate the local maximums of *x.* The program stops stepping once the difference between each successive approximated local maximum is less than *epsamp.* The program finishes by printing out the final local max as *finalamp* and defines *tfin* as the current value of *t.* If *tfin* was reached before the difference in the last two local maximums was less than *epsamp,* then it was concluded that the solution did not meet the amplitude convergence criteria defined by the procedure.
From the trials in Task 1, we see in run 1, that:

A. t_{ss} for runs 1,2, and 5 is larger than for runs 3 and 4 because the value of *c* for runs 3 and 4 is significantly larger than *c* used in 1, 2, and 5. In a physical sense, this indicates a larger damper constant meaning that the damping effect is larger and the system is forced into steady state faster than a week damper.

B. Run 5 did not meet amplitude convergence criteria because the value of ε changed. We see convergence in run 2 (which happens to be the same as run 5 besides differing values of ε). Because the tolerance for convergence has increased, adaptive scheme used is able to choose a larger time step and satisfy the new value for epsilon. The leads to less accurate solutions to the differential equation. In this new case, the transient has a larger impact on whether or not the program determines a solution has been found for steady state. Because the impact of the transient should approach 0 as *t* goes to ∞, we do expect amplitude convergence to be found at some point if the code were to solve for a longer duration.

C. *acceptr* is a good measure of the effectiveness of the adaptive method used because it effectively creates a scoring metric. The formula is: 2**nsteps/count*. We see that the increasing *nsteps* increases the value of *acceptr* and increasing the number of times *prime.m* is called decreases the score of *acceptr*. We can interpret this as the number of times we must call *prime.m* to get an accepted solution. From a computational power perspective, the fewer the better. Therefore, a larger score of *acceptr* is better than a smaller score. In run 6 we see that the value of *acceptr* is less than each of the other trials due to the increase in *hraise. Hraise* is responsible for changing the time step size, h. Because *hraise* is larger for run 6, it is likely that this value changes *h* too much given the period of oscillation for this run. This means that the code changes h very frequently and must re-call *prime.m* a lot of times using a new step size thus decreasing the score represented by *acceptr.*

D. It can be speculated that the reason for the *acceptr* being similar for runs 1-5 is that *hraise* is the same for all of these. Because *acceptr* relies on *nsteps*, raising the value of *hraise* will reduce the amount of steps needed for the loop to complete, thus changing *acceptr*.

Task 2:

Introduction:

In Task 2, we will modify *adapt.m* to generate a response curve for *nomega*. The response curve shows the effects of a range of values for *omega* on the exact amplitude of the system. The results will be computed numerically and analytically.

Task 2:

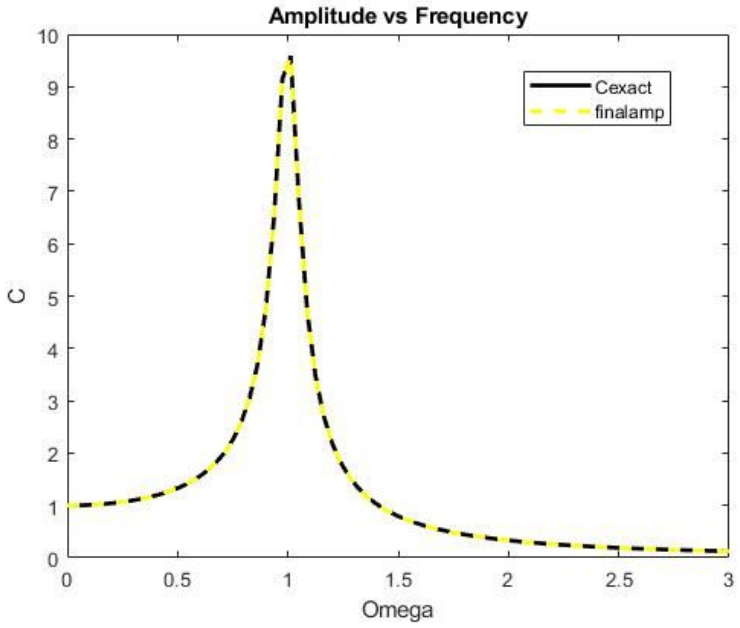

Figure 5: *Computed and exact response curves for varying omega values.*

Task 2 Analysis:

To generate this response curve, we had to modify the code from Task 1 to calculate the values of Cexact and finalamp over a range of omega values instead of just one. To do this, a vector of omega values was created ranging from the given *omegalow* to *omegatop* with nomega even spacing (using the Matlab linspace function), and then a for loop was added prior to the existing while loop so that the loop could be ran through for each omega within the vector. The results from task 2 are depicted in Figure 5, as seen above, where the computed and exact response curves are plotted on top of each other, and you can see how both the Cexact and finalamp amplitude values are basically the same. This proves that the Matlab improved Euler computations for linear springs and their amplitudes are highly accurate when compared to the exact computation for it.

Task 3:

Introduction:

In Task 3, we model a nonlinear oscillator with β = 0.04. In this Task, we will compare the effects of solving our system forwards or backwards, and compare the steady state results. We will graph two response curves on the same axis, one solved from the maximum value of omega to the least and one the opposite. We will also set $\beta = 0$ to observe the effects of removing the nonlinear term in the equation and ensure that no hysteresis is formed.

Task 3:

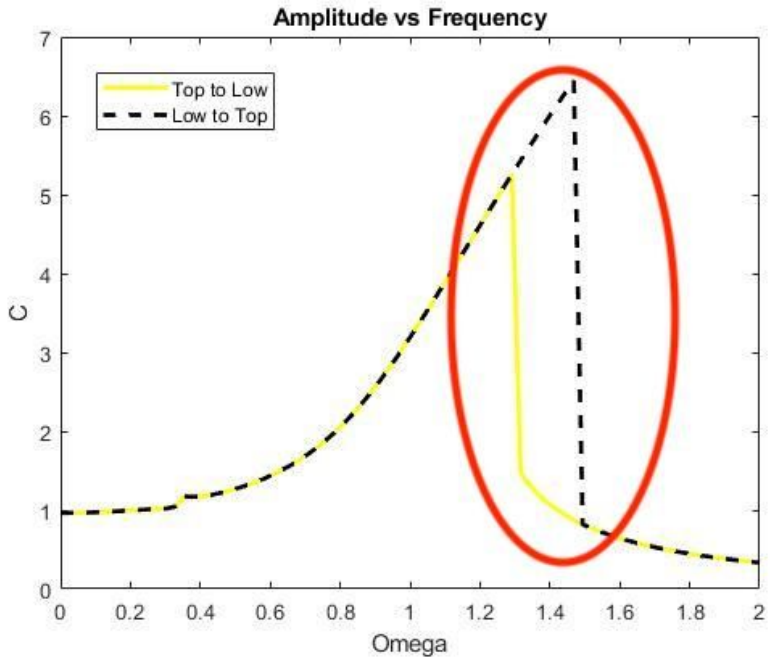

Figure 6: *Computed response curves for varying omega values when $\beta = 0.04$. In one case omega is calculated from omegatop to omegalow and the other case omega is calculated from omegalow to omegatop. The red circled region highlights the hysteresis.*

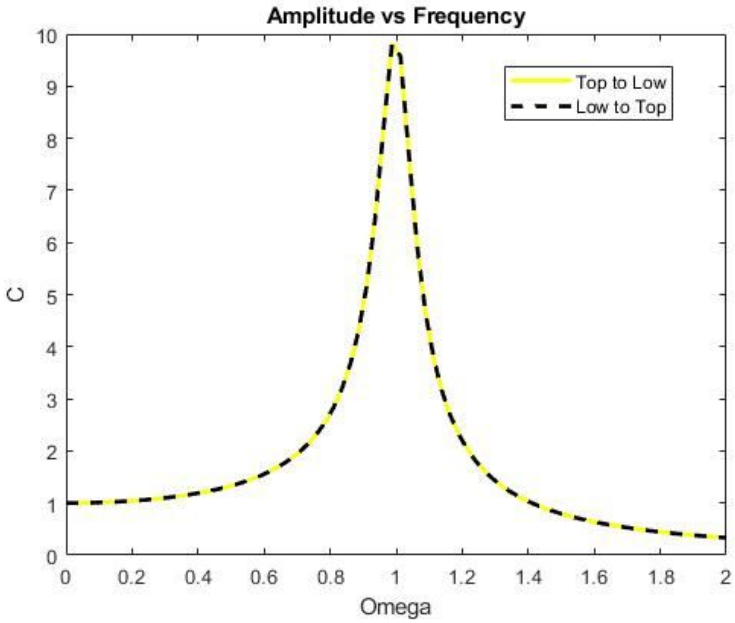

Figure 7: *Computed response curves for varying omega values when $\beta = 0$. In one case omega is calculated from omegatop to omegalow and the other case omega is calculated from omegalow to omegatop.*

Task 3 Discussion and Analysis:

In Task 3, βx^3 was added to the oscillator equation making it nonlinear. This caused the graph to develop a hysteresis which is the gap that is seen between the *omegatop* to *omegalow* plot and the *omegalow* to *omegatop* plot depicted in Figure 6. This hysteresis generated when the computed amplitude values differed between the *omegatop* to *omegalow* and *omegalow* to *omegatop* sequences due to the history of the system affecting the resulting amplitudes of the spring at that range of omega (around 1.3 to 1.5), since pre existing x and y values are used as inputs throughout the for loop. In Figure 7 when the βx^3 was removed from the oscillator equation and the code was ran again under the same conditions, it can be seen that this hysteresis disappears, proving that this only appears within a nonlinear spring system. Note that the one line change to the program that allowed for the reverse *omegatop* to *omegalow* computation was switching the *omegalow* and *omegatop* positions in the linspace vector creation (Highlighted in Appendix 3).

Conclusion & Summary

1. In Task 1 we converted *yprime.m* code from the previous lab that was used to solve a differential equation for y as a function of time. Our modifications allowed allow us to solve systems of equations, x(t) and y(t). This functionality included adding an adaptive method improved euler method for each and code to check if the oscillator had reached steady state yet. We added a scoring system *acceptr* that measures the efficiency of the solve parameters for solving an equation. We ran several trials with different parameters and drew several conclusions. Firstly, the effects of different values of *c* on the time taken to reach steady state. Secondly: In run 5, we see that increasing the magnitude of epsilon can actually cause a less precise solution. Finally, we see that *accept* can be used as a metric of assessing the efficiency of our solve parameters: more times *prime.m* called per accepted solution leads to a lower score.

2. In Task two we changed *adapt.m* and *prime.m* to generate a response curve. We replaced ω from *prime.m* with three new variables in order to compute the response for *nomega* both numerically and analytically. We tested the amplitude response to a range of values for ω rather than just one value used in the previous Task. After running the tests we found that *Cexact* and *finalamp* were essential the same for all values of omega tested. This shows that the MATLAB solution to the final amplitude of the oscillator is very similar to the analytical solution indicating a high level of precision.

3. In Task 3, we observe the effects of solving the response curve forward and backwards in order to see the hysteresis created by βx^3 term in the nonlinear oscillator equation. When solved forward and backwards, from omega high to omega low, we see that a hysteresis is indeed formed in the nonlinear case. This difference is very significant in the response curves. We also see that when $\beta = 0$ no hysteresis is formed. **Figure 7.**

Appendix 1: Task 1

prime.m

```
function f = prime(t,x,y)

%Calling global variables
global m
global c
global k
global B
global F0
global W
global count

%Equation for function
f = -(c/m)*y -(k/m)*x -(B/m)*x^3 +(F0/m)*cos(W*t);

%Adding 1 to count when the function is called
count = count+1;
end
```

adapt.m

```
%Calling and setting global variables
global m
m = 1;
global c
c = 0.1;
global k
k = 1;
global B
B = 0;
global F0
F0 = 1;
global W
W = 1;
global count
count = 0;

%Setting initial conditions
[yinit, xinit, tinit, tfin, h, epsilon, epsamp, hbig, hsmall, hcut, hraise] =
textread('adapt.dat','%f %f %f %f %f %f %f %f %f %f %f');

%Creating arrays
y = [];
x = [];
t = [];
xmax = [];
```

```
%Initializing variables for the loop
j = 1;
i = 1;
y(j) = yinit;
x(j) = xinit;
t(j) = tinit;
xmax(i) = 0;
nsteps = 0;
xmaxdiff = 1+epsamp;

%Adaptive improved Euler while loop
while (t(j) < tfin)
      tnew = t(j)+h;

%Checking that if tnew is greater than tfin, it will set tnew=tfin to run
%the loop one more time
      if tnew > tfin
      tnew = tfin;
      h = tnew-t(j);
      end

%Creating the values k0, ylow, xlow, k1, yhigh, and xhigh that will be used to
%check the error within this equation to make sure it is less than epsilon
      k0 = prime(t(j),x(j),y(j));
      ylow = y(j)+h*k0;
      xlow = x(j)+h*y(j);
      k1 = prime(tnew,xlow,ylow);
      yhigh = y(j)+((h/2)*(k0+k1));
      xhigh = x(j)+((h/2)*(y(j)+ylow));

%Checking that the error is less than epsilon and if it is, then raising h
%by a factor of hraise, adding 1 to j and then setting y(j) to yhigh, x(j) to
%xhigh, and t(j) to tnew in order to set the intial values for the next loop
      if abs((yhigh-ylow)/abs(yhigh)) && abs((xhigh-xlow)/abs(xhigh)) <
epsilon
      h = h*hraise;
      j = j+1;
      y(j) = yhigh;
      x(j) = xhigh;
      t(j) = tnew;

%Add 1 to the value of nsteps
      nsteps = nsteps+1;

%Computation of when the displacement attains a local max
      if (y(j-1) > 0) && (y(j) < 0)
            i = i+1;
            tmax = interp1([y(j-1) y(j)],[t(j-1) t(j)],0);
            xmax(i) = interp1([t(j-1) t(j)],[x(j-1) x(j)],tmax);
            xmaxdiff = abs(xmax(i)-xmax(i-1));
      end
%If h ends up bigger than hbig when we raise it by hraise, then set it equal
%to hbig
      if h > hbig
```

```
            h = hbig;
      end

%If the error was not less than epsilon we divide h by hcut to then run the
%next loop with a small h
      else
      h = h/hcut;

%If h is too small then we output an error message that its too small
      if h < hsmall
      error('Too small of a time step')
      end
      end

%End while loop if 2 successive maxima differ by less than epsamp
      if xmaxdiff < epsamp
      disp('Met amplitude convergence criteria')
      finalamp = xmax(i);
      fprintf('finalamp is %.4f meters\n',finalamp)
      Cexact = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      fprintf('Cexact is %.4f meters\n',Cexact)
      tfin = t(j);
      end

%Display finalamp and Cexact if program reaches tfin without having met the
%amplitude convergence criteria
      if (xmaxdiff > epsamp) && (tnew == tfin)
      disp('Amplitude convergence criteria not met')
      finalamp = xmax(i);
      fprintf('finalamp is %.4f meters\n',finalamp)
      Cexact = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      fprintf('Cexact is %.4f meters\n',Cexact)
      end
end

%Outputting acceptr variable
acceptr = 2*nsteps/count;
fprintf('acceptr is %.4f\n',acceptr)

%Plotting x(t) and y(t)
figure(1)
plot(t,x,'b','LineWidth',2.0)
xlabel('Time (t)')
xlim([0 tfin])
ylabel('x')
title('Adaptive model for x(t) Run 1')

figure(2)
plot(t,y,'r','LineWidth',2.0)
xlabel('Time (t)')
xlim([0 tfin])
ylabel('y')
title('Adaptive model for y(t) Run 1')
```

adapt.dat (Run1)

```
0 5 0 800 0.1 1e-5 0.00001 1 1e-9 2 1.05
```

Appendix 2: Task 2

adaptr.m

```
global W
global m
m = 1;
global c
c = 0.1;
global k
k = 1;
global B
B = 0;
global F0
F0 = 1;
global count
count = 0;

%Setting initial conditions
[yinit, xinit, tinit, tfin, h, epsilon, epsamp, hbig, hsmall, hcut, hraise,
omegalow, omegatop, nomega] = textread('adaptr.dat','%f %f %f %f %f %f %f %f
%f %f %f %f %f %f');
nsteps = 0;

%Creating arrays
w = linspace(omegalow, omegatop, nomega);
Cexact = [];
finalamp = [];

%Creating for loop for each value of omega
for ii = 1:nomega
      %Setting Omega
      W = w(ii);

      %Reinitializing variables for while loop
      h = 0.1;
      tfin = 800;

      %Initializing x and y for while loop first omega value
      if W == w(1)
      j = 1;
      i = 1;
      y = [];
      x = [];
      t = [];
      xmax = [];
      t(j) = tinit;
      xmax(i) = 0;
      xmaxdiff = 1+epsamp;
      y(j) = yinit;
      x(j) = xinit;
```

```
      %Initializing x and y for while loop all other omega values
      else
      ysave = y(j);
      xsave = x(j);
      j = 1;
      i = 1;
      y = [];
      x = [];
      t = [];
      xmax = [];
      t(j) = tinit;
      xmax(i) = 0;
      xmaxdiff = 1+epsamp;
      y(j) = ysave;
      x(j) = xsave;
      end

%Adaptive improved Euler while loop
while (t(j) < tfin)
      tnew = t(j)+h;

%Checking that if tnew is greater than tfin, it will set tnew=tfin to run
%the loop one more time
      if tnew > tfin
      tnew = tfin;
      h = tnew-t(j);
      end

%Creating the values k0, ylow, xlow, k1, yhigh, and xhigh that will be used to
%check the error within this equation to make sure it is less than epsilon
      k0 = prime(t(j),x(j),y(j));
      ylow = y(j)+h*k0;
      xlow = x(j)+h*y(j);
      k1 = prime(tnew,xlow,ylow);
      yhigh = y(j)+((h/2)*(k0+k1));
      xhigh = x(j)+((h/2)*(y(j)+ylow));

%Checking that the error is less than epsilon and if it is, then raising h
%by a factor of hraise, adding 1 to j and then setting y(j) to yhigh, x(j) to
%xhigh, and t(j) to tnew in order to set the intial values for the next loop
      if abs((yhigh-ylow)/abs(yhigh)) && abs((xhigh-xlow)/abs(xhigh)) <
epsilon
      h = h*hraise;
      j = j+1;
      y(j) = yhigh;
      x(j) = xhigh;
      t(j) = tnew;

%Add 1 to the value of nsteps
      nsteps = nsteps+1;

%Computation of when the displacement attains a local max
      if (y(j-1) > 0) && (y(j) < 0)
            i = i+1;
```

```
            tmax = interp1([y(j-1) y(j)],[t(j-1) t(j)],0);
            xmax(i) = interp1([t(j-1) t(j)],[x(j-1) x(j)],tmax);
            xmaxdiff = abs(xmax(i)-xmax(i-1));
      end
%If h ends up bigger than hbig when we raise it by hraise, then set it equal
%to hbig
      if h > hbig
            h = hbig;
      end

%If the error was not less than epsilon we divide h by hcut to then run the
%next loop with a small h
      else
      h = h/hcut;

%If h is too small then we output an error message that its too small
      if h < hsmall
      error('Too small of a time step')
      end
      end

%End while loop if 2 succesive maxima differ by less than epsamp
      if xmaxdiff < epsamp
      disp('Met amplitude convergence criteria')
      finalamp(ii) = xmax(i);
      Cexact(ii) = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      tfin = t(j);
      end

%Pause program if program reaches tfin without having met the
%amplitude convergence criteria and assign Cexact and finalamp values if user
%decides to continue
      if (xmaxdiff > epsamp) && (tnew == tfin)
      disp('Amplitude convergence criteria not met')
      pause
      finalamp(ii) = xmax(i);
      Cexact(ii) = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      end
end
end

%Plotting computed and exact response curves)
figure (5)
plot(w,Cexact,'k','LineWidth',2.0)
hold on
plot(w,finalamp,'y--','LineWidth',2.0)
hold off
xlabel('Omega')
ylabel('C')
title('Amplitude vs Frequency')
legend('Cexact', 'finalamp')
```

adaptr.dat

```
0 5 0 800 0.1 0.00001 0.00001 1 1e-9 2 1.05 0 3 81
```

Appendix 3: Task 3

```
global W
global m
m = 1;
global c
c = 0.1;
global k
k = 1;
global B
B = 0.04;
global F0
F0 = 1;
global count
count = 0;

%Setting initial conditions
[yinit, xinit, tinit, tfin, h, epsilon, epsamp, hbig, hsmall, hcut, hraise,
omegalow, omegatop, nomega] = textread('adaptr.dat','%f %f %f %f %f %f %f %f
%f %f %f %f %f %f');
nsteps = 0;

%Creating arrays
%This is the one line change for this code
w = linspace(omegatop, omegalow, nomega);
Cexact = [];
finalamp = [];

%Creating for loop for each value of omega
for ii = 1:nomega
      %Setting Omega
      W = w(ii);

      %Reinitializing variables for while loop
      h = 0.1;
      tfin = 800;

      %Initializing x and y for while loop first omega value
      if W == w(1)
      j = 1;
      i = 1;
      y = [];
      x = [];
      t = [];
      xmax = [];
      t(j) = tinit;
      xmax(i) = 0;
      xmaxdiff = 1+epsamp;
      y(j) = yinit;
      x(j) = xinit;


      %Initializing x and y for while loop all other omega values
```

```
      else
      ysave = y(j);
      xsave = x(j);
      j = 1;
      i = 1;
      y = [];
      x = [];
      t = [];
      xmax = [];
      t(j) = tinit;
      xmax(i) = 0;
      xmaxdiff = 1+epsamp;
      y(j) = ysave;
      x(j) = xsave;
      end

%Adaptive improved Euler while loop
while (t(j) < tfin)
      tnew = t(j)+h;

%Checking that if tnew is greater than tfin, it will set tnew=tfin to run
%the loop one more time
      if tnew > tfin
      tnew = tfin;
      h = tnew-t(j);
      end

%Creating the values k0, ylow, xlow, k1, yhigh, and xhigh that will be used to
%check the error within this equation to make sure it is less than epsilon
      k0 = prime(t(j),x(j),y(j));
      ylow = y(j)+h*k0;
      xlow = x(j)+h*y(j);
      k1 = prime(tnew,xlow,ylow);
      yhigh = y(j)+((h/2)*(k0+k1));
      xhigh = x(j)+((h/2)*(y(j)+ylow));

%Checking that the error is less than epsilon and if it is, then raising h
%by a factor of hraise, adding 1 to j and then setting y(j) to yhigh, x(j) to
%xhigh, and t(j) to tnew in order to set the intial values for the next loop
      if abs((yhigh-ylow)/abs(yhigh)) && abs((xhigh-xlow)/abs(xhigh)) <
epsilon
      h = h*hraise;
      j = j+1;
      y(j) = yhigh;
      x(j) = xhigh;
      t(j) = tnew;

%Add 1 to the value of nsteps
      nsteps = nsteps+1;

%Computation of when the displacement attains a local max
      if (y(j-1) > 0) && (y(j) < 0)
            i = i+1;
            tmax = interp1([y(j-1) y(j)],[t(j-1) t(j)],0);
```

```
            xmax(i) = interp1([t(j-1) t(j)],[x(j-1) x(j)],tmax);
            xmaxdiff = abs(xmax(i)-xmax(i-1));
      end
%If h ends up bigger than hbig when we raise it by hraise, then set it equal
%to hbig
      if h > hbig
            h = hbig;
      end

%If the error was not less than epsilon we divide h by hcut to then run the
%next loop with a small h
      else
      h = h/hcut;

%If h is too small then we output an error message that its too small
      if h < hsmall
      error('Too small of a time step')
      end
      end

%End while loop if 2 succesive maxima differ by less than epsamp
      if xmaxdiff < epsamp
      disp('Met amplitude convergence criteria')
      finalamp(ii) = xmax(i);
      Cexact(ii) = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      tfin = t(j);
      end

%Pause program if program reaches tfin without having met the
%amplitude convergence criteria and assign Cexact and finalamp values if user
%decides to continue
      if (xmaxdiff > epsamp) && (tnew == tfin)
      disp('Amplitude convergence criteria not met')
      pause
      finalamp(ii) = xmax(i);
      Cexact(ii) = F0/sqrt((k-m*W^2)^2+(c*W)^2);
      end
end
end
```

adaptr.dat

```
0 5 0 800 0.1 0.00001 0.00001 1 1e-9 2 1.05 0 2 80
```