Two Dimensional Harmonic Oscillator

Kevin Francis

April 1, 2021

Introduction

The harmonic oscillator is something that comes up over and over again all throughout physics. Whether it is being used for a simple mass connected to a spring, or being used to classically model the motion of atoms inside a solid. It is apparent that this is a very useful tool.

Equations and Numerical Methods

The ordinary differential equations (ODE) solved in this problem come from Newtons second law

$$F_{net} = ma \tag{1}$$

Below are the steps to obtain the ODE's that model this problem:

$$\begin{gathered} U = \frac{1}{2}kr^2 - mgy \\ F_{net} = -\bigtriangledown U = -kr - mg \\ \text{where } r = \sqrt{x^2 + y^2} \end{gathered} \tag{2}$$

Now we take F_{net} from (1) and plug in what we found in (2). Doing this and rewriting ma will yield the following ODE's:

$$\begin{aligned} m\frac{d^2x}{dt^2} &= -kx - mg \\ m\frac{d^2y}{dt^2} &= -ky - mg \end{aligned} \tag{3}$$

Now I have what is needed to solve the problem, but first lets clean up (3) a little.

$$\begin{aligned} \frac{d^2x}{dt^2} &= -\omega^2 x - g \\ \frac{d^2y}{dt^2} &= -\omega^2 y - g \\ \text{where } \omega &= \sqrt{\frac{k}{m}} \end{aligned} \tag{4}$$

If the ODE's in (4) were first order then they could be solved as they are, however the methods used would be simpler if they are first order. To do this we can rewrite $\frac{d^2x}{dt^2}$ as $\frac{dv_x}{dt}$ and similarly for y. Now we must solve the following four ODE's

$$\frac{dv_x}{dt} = x \tag{5}$$

$$\frac{dv_y}{dt} = y \tag{6}$$

$$\frac{dv_x}{dt} = -\omega^2 x - g \tag{7}$$

$$\frac{dv_y}{dt} = -\omega^2 y - g \tag{8}$$

Now that there are only first order ODE's we can move on to the methods being used to solve these equations. In my project I will not just be modeling a 2D harmonic oscillator but also comparing different numerical techniques. The first method I will be using is the runga-kuta method. This is one of the most used algorithms for solving ODE's, but it does lack something that is important to this problem. This method does not conserve energy. So it will appear that the system has some type of dampening but there is no dampening force in F_{net}. Below is the algorithm and code for this method.

$$\begin{aligned}
&\frac{dx}{dt} = f(x,t)\\
&k_1 = hf(x,t)\\
&k_2 = hf(x + \frac{1}{2}k_1, t + \frac{1}{2}h)\\
&k_3 = hf(x + \frac{1}{2}k_2, t + \frac{1}{2}h)\\
&k_4 = hf(x + k_3, t + h)\\
&x(t+h) = x(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)
\end{aligned} \tag{9}$$

```python
def rk4(N): # does not conserve energy
    k = 2
    B = 3
    A = 0
    m = .5
    omega2 = k/m
    t_i = 0 #time in seconds
    t_f = 500 #time in seconds
    h = float((t_f - t_i)) / N #h is a tiny step in the time interval
    tpts = arange(t_i,t_f,h)
    p1 = array([A,B,0.,0.],float) #initialize position/velocity
    p_tx = zeros(tpts.size) #create an empty array to save positions
    p_ty = zeros(tpts.size)
    for i in range(1,tpts.size):
        # print(i)
        k1 = h*func1(p1,omega2)
        k2 = h*func1(p1+.5*k1,omega2)
        k3 = h*func1(p1+.5*k2,omega2)
        k4 = h*func1(p1+k3,omega2)
        p1 += (k1+2*k2+2*k3+k4)/6
        p_tx[i] = p1[0] #updating positions
        p_ty[i] = p1[1]

    #print('\n' + "for rgk " + str(N) + " steps!" + '\n' + str(p_t) + '\n')
    plot(tpts,p_tx,'-',label='X')
    plot(tpts,p_ty,'-',label ='Y')
    xlabel("Time")
    xlim(0,100)
    ylabel("Position")
    legend(loc='best')
    show()
    return tpts
def leap(r,omega2,m,k):
```

Listing 1: Function for applying the method in (9)

The other method that is used is the leap frog method. This method starts at a mid point, but instead of increasing each step by a half it moves one full step to the next mid point. For most calculation 4th order runga-kutta would be more accurate, however this problem requires that the energy of the system is conserved. This is something that is needed for this system. Here is the general algorithm for this method as well as the code to implement it.

$$\begin{aligned} x(t+h) &= x(t) + hf(x(t+.5h), t+.5h) \\ x(t+1.5h) &= x(t+.5h) + hf(x(t+h), t+h) \end{aligned} \tag{10}$$

```
    p_t1 = zeros(tpts.size) #create an empty array to save positions
    r1 = r
    r2 = r1 + .5 * h * func1(r1,omega2)
    x1 = [] #x position
    y1 = [] #y position
    v1 = [] #x velocity
    v2 = [] #y velocity
    K = [] #kinetic energy
    Usp = [] #potential spring
    Utot = [] #potential total
    Ugrav = [] #potential grav
    tot = [] # total energy
    for t in tpts:
        x1.append(r1[0])
        y1.append(r1[1])
        v1.append(r1[2])
        v2.append(r1[3])
        K.append(KE(r1[2:],m))
        Usp.append(PEsp(r1[0:2],k))
        Ugrav.append(PEgrav(r1[1],m))
        Utot.append(Usp[-1]+Ugrav[-1])
        tot.append(Utot[-1]+K[-1])
        r1 += h * func1(r2,omega2)
        r2 += h * func1(r1,omega2)
    return x1,y1,v1,v2,K,Usp,Ugrav,Utot,tot #Solve func1 using leap frog
    method
def OneD():
```

Listing 2: Function for applying the method in (10)

Results

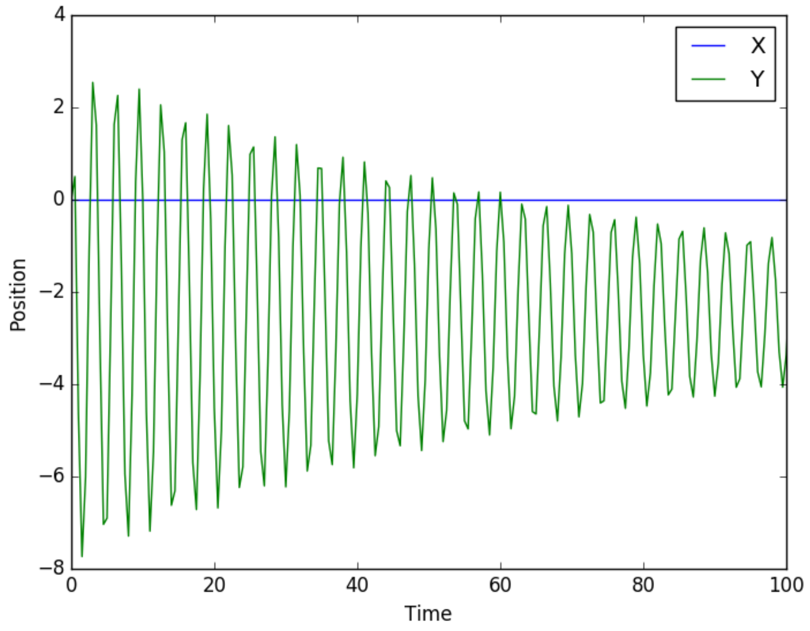

Figure 1: This shows the position vs time using runga-kutta for a one dimensional oscillator. The ”dampening” from this method can be seen.

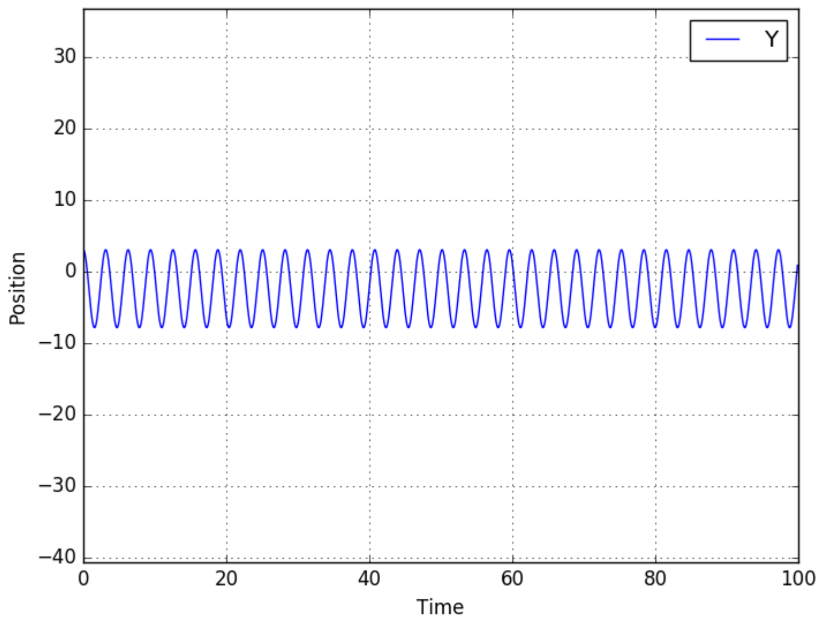

Figure 2: This shows the position vs time using leap frog for a one dimensional oscillator.

It can be seen from 1 and 2 that one will conserves energy while the other will not. If positions of the system are not periodic and have some type of dampening then the energy will not be conserved.

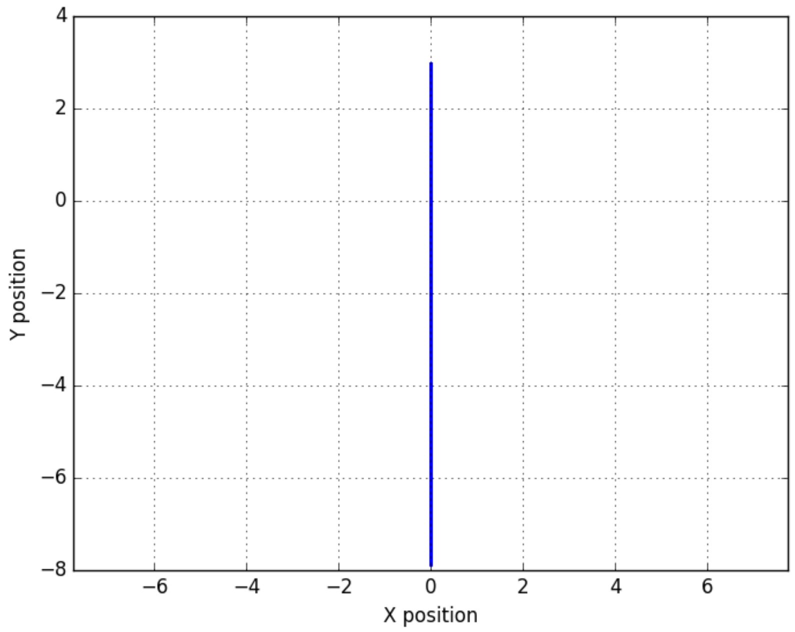

Figure 3: This shows the position of a one dimensional harmonic oscillator.

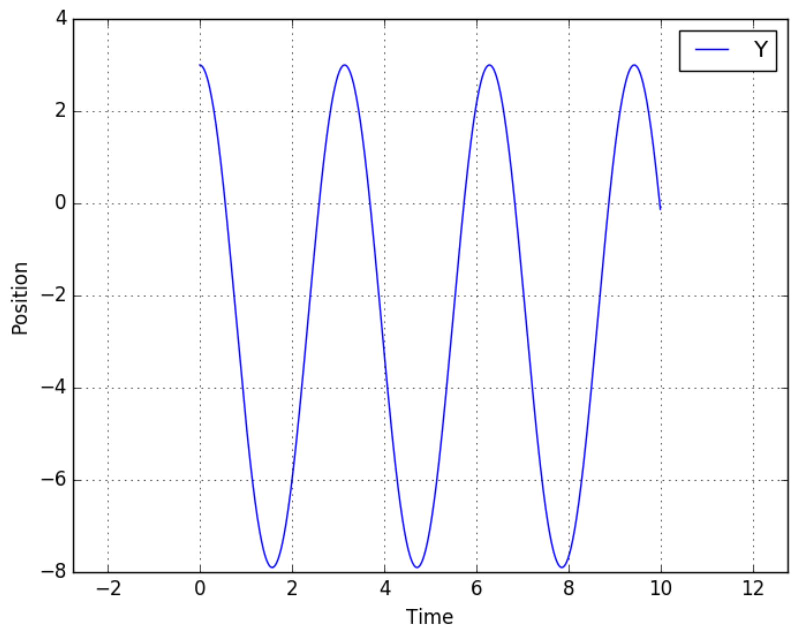

Figure 4: This shows the position vs time for a one dimensional oscillator

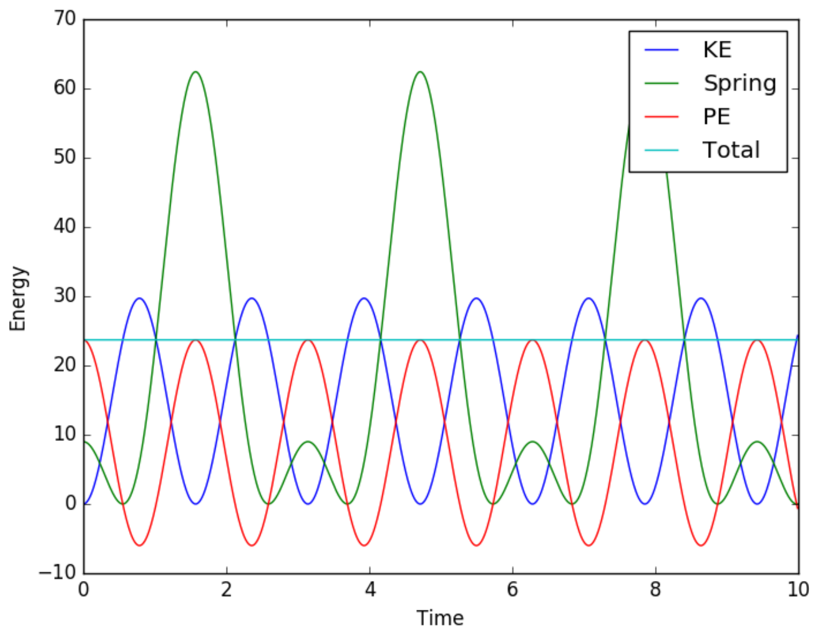

Figure 5: This shows the energy vs time for a one dimensional oscillator

```
    k = 2
    B = 3
    A = 0
    m = .5
    omega2 = k/m
    r = array([A,B,0.,0.],float) #initialize position/velocity
    x1,y1,v1,v2,K,Usp,Ugrav,Utot,tot = leap(r,omega2,m,k)
    figure(1)
    plot(tpts,K,'-',label="KE")
    plot(tpts,Usp,label="Spring")
    plot(tpts,Utot,label="PE")
    plot(tpts,tot,label="Total")
    legend(loc="best")
    xlabel("Time")
    #xlim(0,100)
    ylabel("Energy")
    figure(2)
    plot(x1,y1,'-')
    xlabel("X position")
    xlim(-10,10)
    ylabel("Y position")
    axis('equal')
    grid(True, which='both')
    figure(5)
    plot(tpts,y1,'-',label='Y')
    xlabel("Time")
    xlim(0,100)
    ylabel("Position")
    axis('equal')
    legend(loc='best')
    grid(True, which='both')
    show() #1d harmonic oscillator #1d harmonic #1d harmonic oscillator
def SpringPendulum():
```
Listing 3: Function for the one dimensional harmonic oscillator

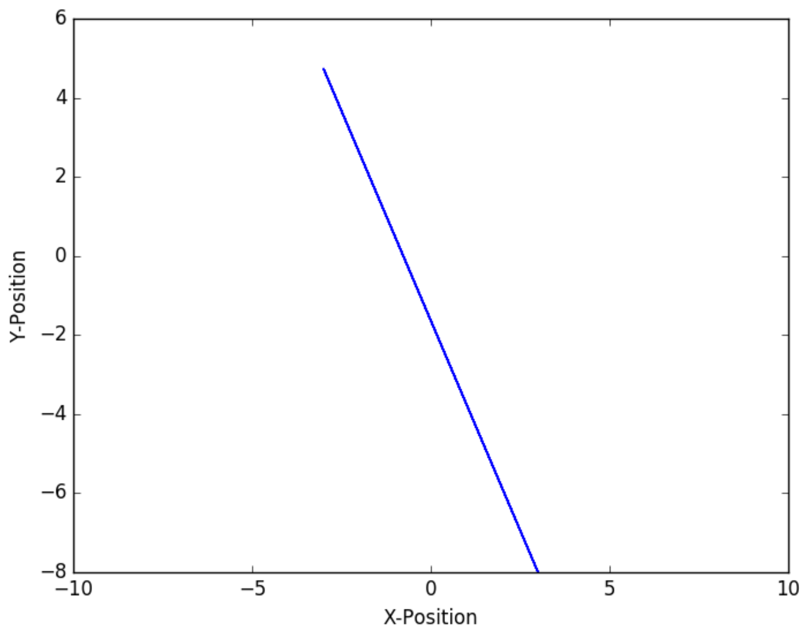

Figure 6: This shows the x position vs y position for an oscillator that behaves like a pendulum with the mass attached to a spring

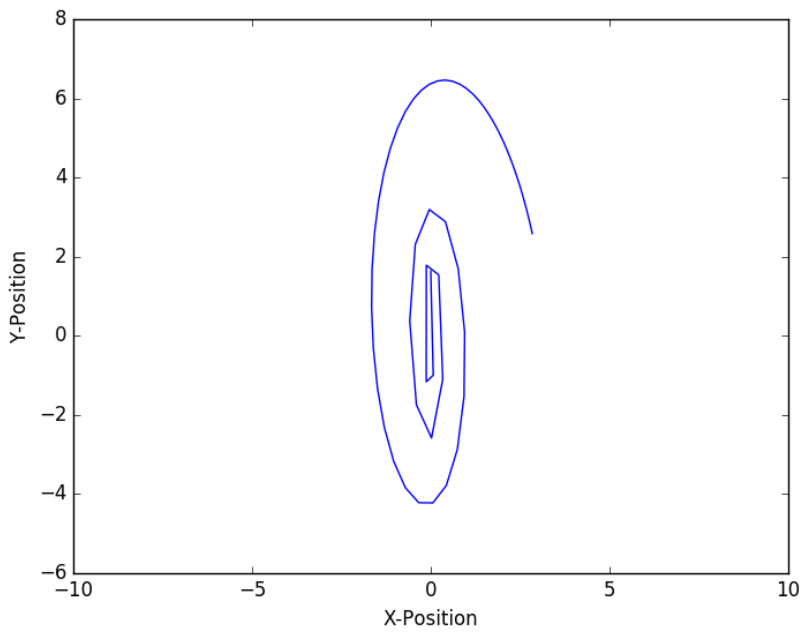

Figure 7: This plot takes into account the angle theta that the pendulum would have. This plot only shows about the first 75 steps out of 1000

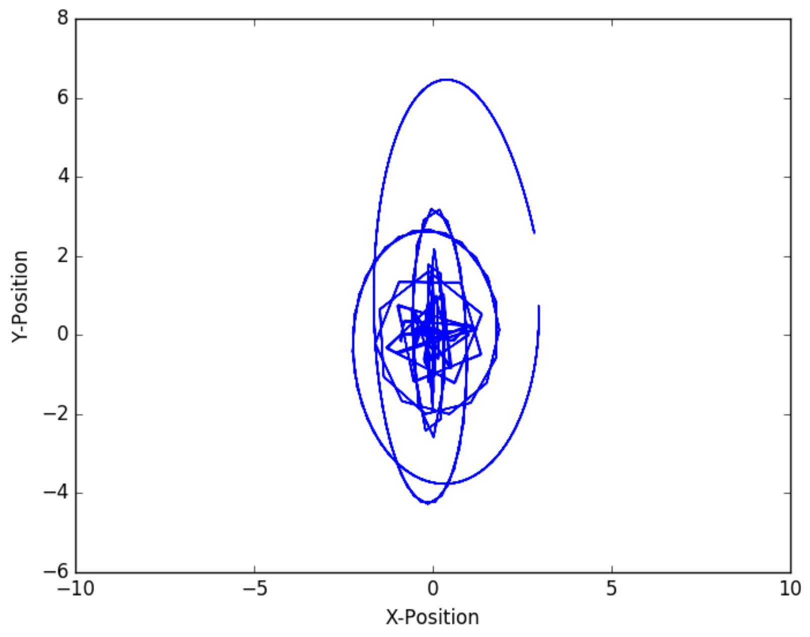

Figure 8: This plot takes into account the angle theta that the pendulum would have. This plot shows all steps.

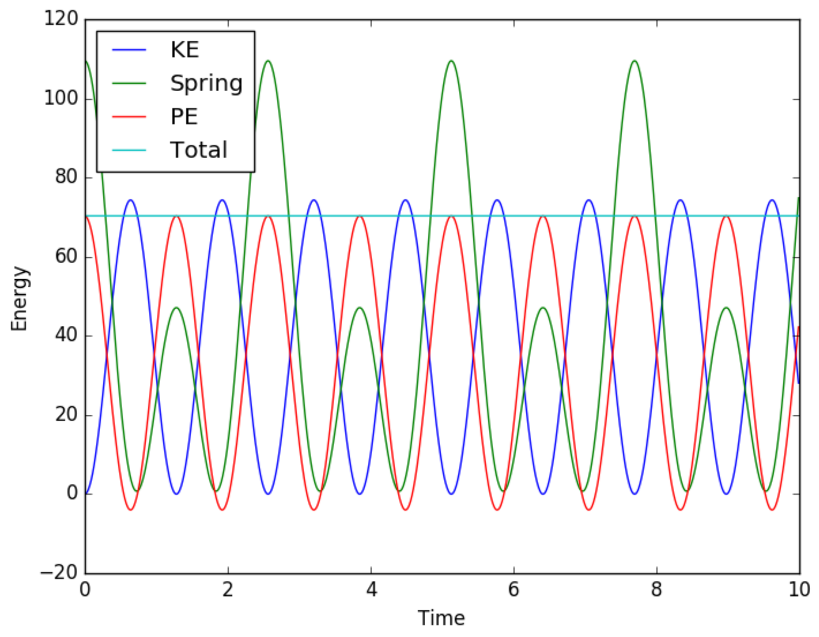

Figure 9: This shows the energy vs time for a pendulum with a spring

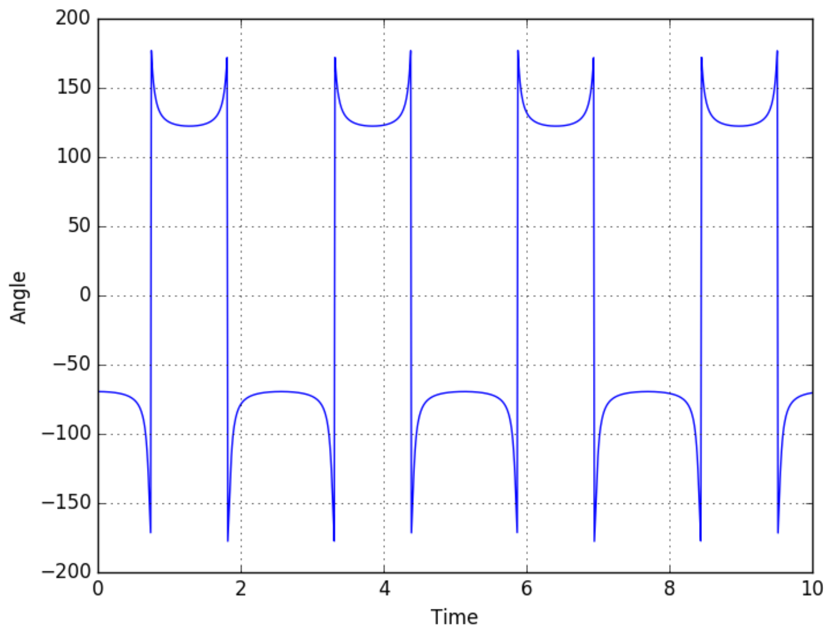

Figure 10: This plot shows how the angle theta changes as the system evolves.

It is worth noting what values were used to generate plots 6 through 10. The initial stretch was in the fourth quadrant, but the values that can change what the plots look like are the mass and the spring constant k. The mass used was $\frac{1}{2}$ and the spring constant was 3, as seen in the code below.

```
k = 3
B = -8.
A = 3.
m = .5
omega2 = k/m
theta = []
x2 = []
y2 = []
R = []
r = array([A,B,0.,0.],float) #initialize position/velocity
x1,y1,v1,v2,K,Usp,Ugrav,Utot,tot = leap(r,omega2,m,k)
for i in range(0,len(x1)):
    theta.append(atan2(y1[i], x1[i]) * 57.2958)
figure(6)
plot(x1,y1,'-')
xlabel("X-Position")
xlim(-10,10)
ylabel("Y-Position")
show()
for i in range(0,len(x1)):
    x1[i] = x1[i] * cos(theta[i])
    y1[i] = y1[i] * sin(theta[i])
for i in range(0,65):
    x2.append(x1[i])
```

```
        y2.append(y1[i])
    for i in range(0,len(x1)):
        R.append(sqrt(x1[i]**2 + y1[i]**2))
    figure(3)
    plot(tpts,K,'-',label="KE")
    plot(tpts,Usp,label="Spring")
    plot(tpts,Utot,label="PE")
    plot(tpts,tot,label="Total")
    legend(loc="best")
    xlabel("Time")
    ylabel("Energy")
    figure(4)
    plot(tpts,theta,'-')
    xlabel("Time")
    xlim(0,10)
    ylabel("Angle")
    grid(True, which='both')
    figure(6)
    plot(tpts,R,'-')
    xlabel("Time")
    xlim(0,10)
    ylabel("R Vector")
    show()#pendulum w/ spring #pendulum with spring
def TimeSteps():
```

Listing 4: Function for the one dimensional harmonic oscillator

Discussion

In the lab a one and two dimensional harmonic oscillator were modeled, as well as two numerical techniques for solving ODE's. All plots after 1 were generated using the leap frog method, however the runga-kutta method requires less steps to achieve the same accuracy. The reason that the leap frog algorithm was used is because of the conservation of energy as mentioned before. Now on to plots 7 and 8. These two plots have the x-position vs the y-position. The plots show how the pendulum slowly reaches equilibrium, which is right around the origin of the plot. In 10 the change in the angle theta can be seen. To help verify and understand this plot the spring constant in the pendulum can be set to one. Doing this will yield in what looks like a sine wave that represents the change in theta, as it should.