PHY407
Lab #6: Ordinary Differential Equations, Pt. 1

Q1: Nicholas Pavanel, Q2: Nicholas and Hayley, Q3: Hayley Agler

October 2020

Q1

See Figure 1.

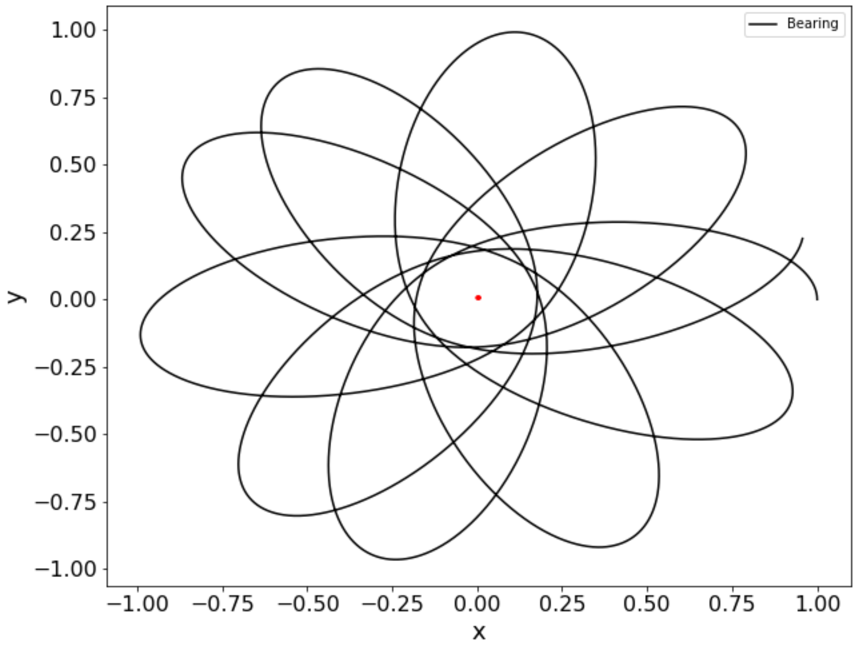

Figure 1: This plot shows the orbit of a piece of space junk; a spherical ball bearing (represented by the black line) orbits around a host steel rod (represented by the red dot) in zero gravity. The rod is assumed to have mass large enough such that the bearing's influence on it is negligible. Note that the orbit of the bearing is around the rod's mid-point, in a plane that is perpendicular with the rod.

The orbit shown in Figure 1 was computed with the 4^{th}-order Runge-Kutta (RK4) method for solving ODEs. RK4 was used to solve the two 2^{nd} order ODEs that are the bearing's equations of motion, Equations 1 and 2.

$$\frac{d^2x}{dt^2} = -GM\frac{x}{r^2\sqrt{r^2+\frac{L^2}{4}}} \tag{1}$$

$$\frac{d^2y}{dt^2} = -GM\frac{y}{r^2\sqrt{r^2+\frac{L^2}{4}}} \tag{2}$$

The code that solved Equations 1 and 2 is shown below:

```
#pseudocode
#define all variables: G, M, L, and time domain
#define empty lists to hold x, y values
#define initial conditions
#define a function to compute RHS of eqn
#define one function that takes all variables (vx,x,vy,y) and t for RK4
#step through time points, performing RK4 on the function

#global variables
G = 1
M = 10
L = 2
#define time start and time stop, number of points in time domain, steps, and time domain
t1= 0
t2=10
N = 5000
h = (t2 - t1) / N
t = np.arange(t1,t2,h)
#define lists for output info
xs = []
ys = []
#define inital conditions
r = [1., 0, 0, 1.]

#define RHS eqn
def RHS(G,M,L,x_y,r):
    return - G * M * (x_y)/(r**2 * np.sqrt(r**2 + (L**2/4)))

#define equation to use in RK4
def f(ics,t):
    x = ics[0]
    vx = ics[1]
    y = ics[2]
    vy = ics[3]
    r = np.sqrt(x**2 + y**2)
    RHS_x = RHS(G,M,L,x,r)
    RHS_y = RHS(G,M,L,y,r)
    return np.array([vx, RHS(G,M,L,x,r), vy, RHS(G,M,L,y,r)],float)

#perform RK4
for i in t:
    xs.append(r[0])
    ys.append(r[2])
    k1 = h * f(r, i)
    k2 = h * f(r + 0.5 * k1, i + 0.5 * h)
    k3 = h * f(r + 0.5 * k2, t + 0.5 * h)
    k4 = h * f(r + k3, t + h)
    r += (k1 + 2 * k2 + 2 * k3 + k4) / 6
```

Simulation Number	P1 Initial Conditions (x,y,vx,vy)	P2 Initial Conditions (x,y,vx,vy
1	(4,4,0,0)	(5.2,4,0,0)
2	(4.5,4,0,0)	(5.2,4,0,0)
3	(2,3,0,0)	(3.5,4.4,0,0)

Table 1: A table displaying the initial conditions for both particles in all simulations.

Q2

a)

Nothing to submit.

b)

See Figures 2, 3, and 4. In this Section we study molecular dynamics with three simulations of two particles. Table 1 shows the initial conditions for both particles in all simulations. The two particles in the simulations that follow are under the influence of only the Lennard-Jones potential $V(r) = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}]$, where ϵ is the depth of the potential well, r is the distance between the two particles and σ is the size of the particle (truly the particle-prickle potential energy-zero distance). All simulations are computed using the Verlet method for solving ordinary differential equations. The outline for how we implemented the Verlet method is shown below:

```
#pseudocode
#define time step of dt=0.01
#define time domain of 100steps
#define initial velocitites to zero
#define a function to calculate the acceleration of particle due to the other particle
#define a function to perform verlet
    #function will take in initial x and y positions
    #create arrays for all values to be stored in
    #initialize first velocities, eqn 7 from lab handout, using the acceleration function just defined
    #have a for loop that loops over all timesteps
        #for every time step, perform the eqns 8, 9, 10, 11 from lab handout
    #return the positions of the particles
#plot positions of the particles and trajectories
```

c)

Figure 2 shows that the initial conditions in Simulation 1 produce oscillatory motion in the two simulated particles. We believe this is because the initial conditions in Simulation 1 provide a unique value of potential energy needed to produce oscillatory motion in the particles. If the initial potential energy of the two particles is too high to begin with, as in Simulation 2, the particles will convert their potential energy into so much kinetic energy that they will escape each other's attractive force. That is, the high kinetic energy of each particle will overcome the potential of the other particle until both particles are separated by a distance in which their attractive potential's are negligible. Simulation 2 shows this in Figure 3. On the other hand, if the initial potential energy of the two particles is too low to begin with, as in Simulation 3, the particles will not have enough potential energy to convert into kinetic energy that will allow them to travel a distance in which they perform oscillatory motion in the time-span of our simulations. Interestingly, if we increase the time-span of our simulations by a factor of 10, Simulation 3 does actually perform oscillatory motion. Thus, we conclude that the initial conditions of Simulation 1 produce oscillatory motion over the time-span of our simulations because they produce a potential energy that is high enough to attract the particles, but low enough that the particle's repulsion does not force them out of each other's range of influence.

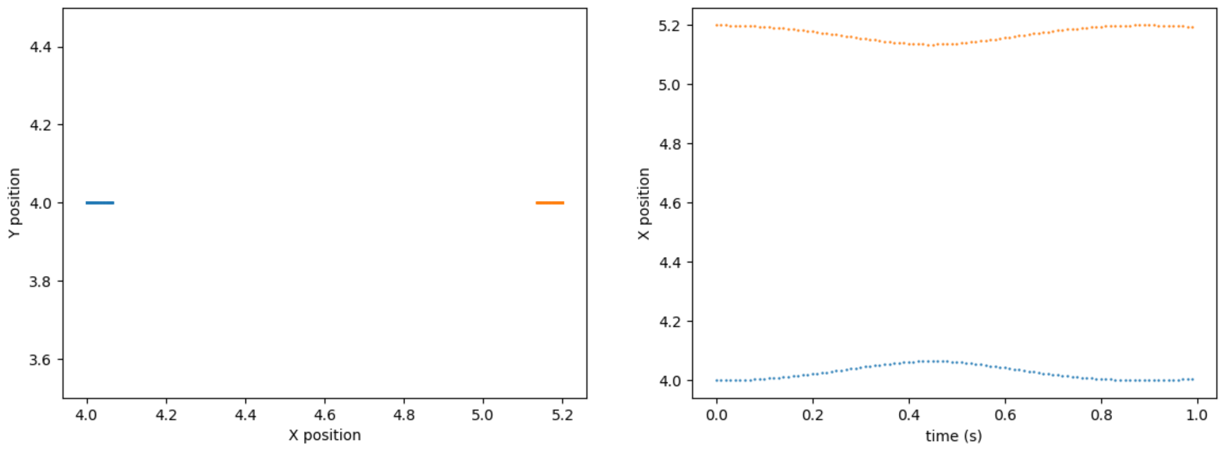

Figure 2: The plot on the left shows x and y positions of particles for Simulation 1. The plot on the right shows the trajectories of two simulated particles for Simulation 1. The initial conditions used can be found in Table 1. Notice that the trajectories display an oscillatory behaviour.

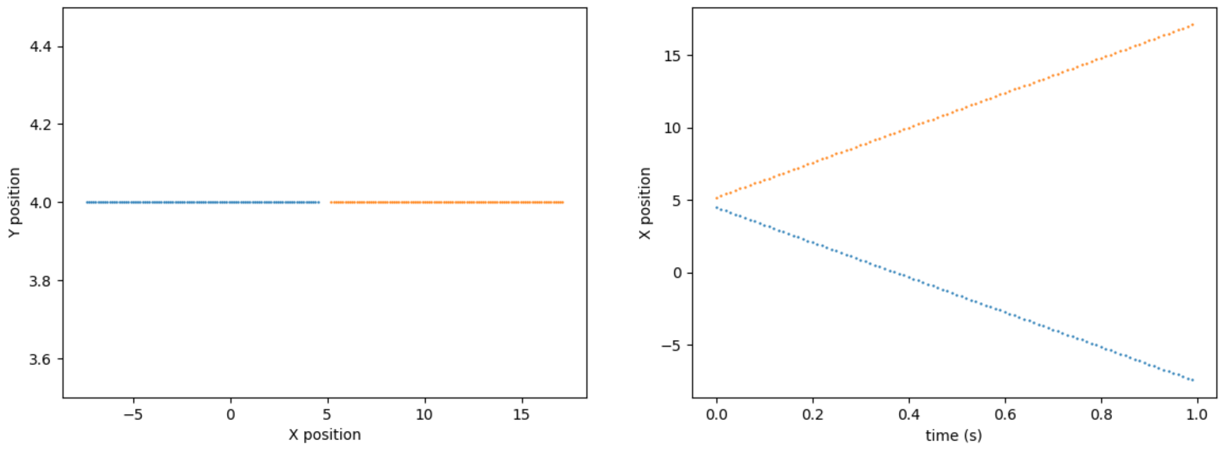

Figure 3: The plot on the left shows x and y positions of particles for Simulation 2. The plot on the right shows the trajectories of two simulated particles for Simulation 2. The initial conditions of the particles can be found in Table 1. Notice that the trajectories display behaviour consistent with strong initial repulsion.

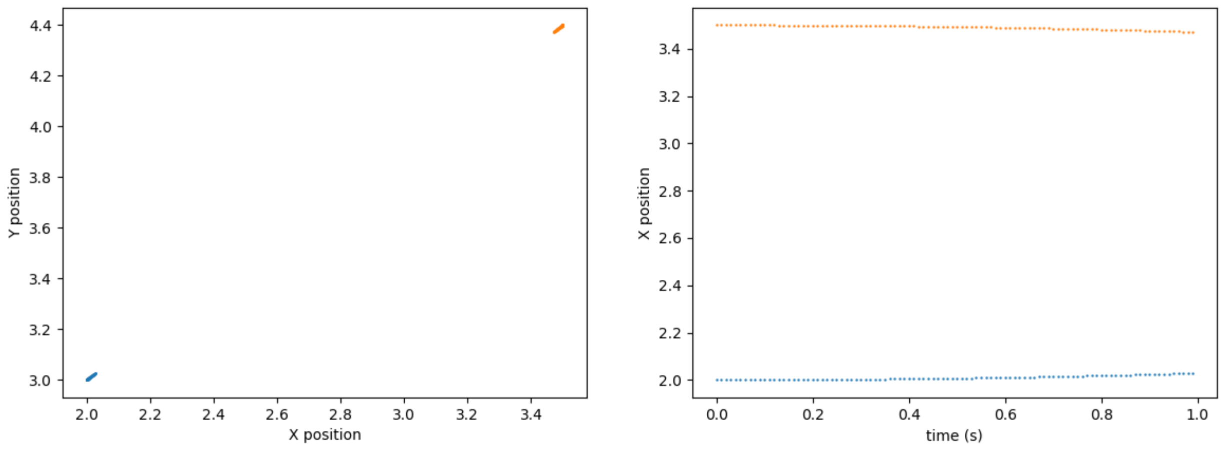

Figure 4: The plot on the left shows x and y positions of particles for Simulation 3. The plot on the right shows the trajectories of two simulated particles for Simulation 3. The initial conditions of the particles can be found in Table 1. Notice that the trajectories display behaviour consistent with weak initial attraction.

Q3

a)

The code and pseudocode from Q2 were updated to simulate the motion of a simukation with N=16 particles. The net force on each partcile 'i' was calculatyed as a sum of the interaction forces between 'i' and all the other particles. The outline for how this was implemented is below:

```
#pseudocode
#define time step of dt=0.01
#define time domain of 1000 steps
#define initial velocitites to zero using the code snippet from lab handout
#define a function that calculates the distance from particle i to each particle j
#define a function to calculate the acceleration of particle i due to the other particles j
    #sum all of the accelerations for particle i to get a total acceleration of particle i due to partic
#perform verlet:
    #create arrays for all values to be stored in, separate into x and y components for convenience
    #initialize first velocities of each particle, eqn 7 from lab handout, using the acceleration functi
    #have a for loop that loops over all timesteps
        #another for loop that loops over 16 particles
            #for every time step, and every particle, perform the eqns 8, 9, 10, 11 from lab handout
#plot positions of the particles and trajectories
```

Unfortunately, the code written does not perform the required task. The positions of the particles should only change a small amount based on the interaction between particles. Instead we see the x and y positions of the particles blow up in a linear fashion, indicating that the interaction between particles is not being properly calculated.

b)

The energy of the system at each time step was computed by calculating the sum of the potential and kinetic energies of each particle. The potential energy was calculated using the Lennard-Jones potential, and the kinetic energy was calculated using $E_k = \frac{1}{2}mv^2$. Due to the errors in a), the energy clearly is not being conserved, as is evident from the plot 5.

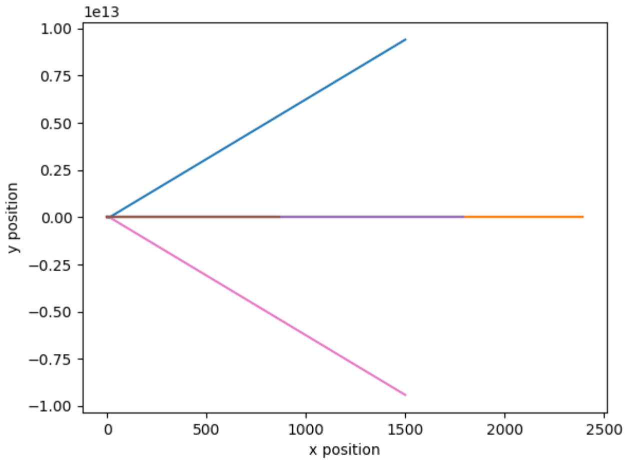

Figure 5: This plot shows the x and y positions of the particles. Clearly the influence of the other particles is not calculated correctly as the particles have incredibly large x and y positions.

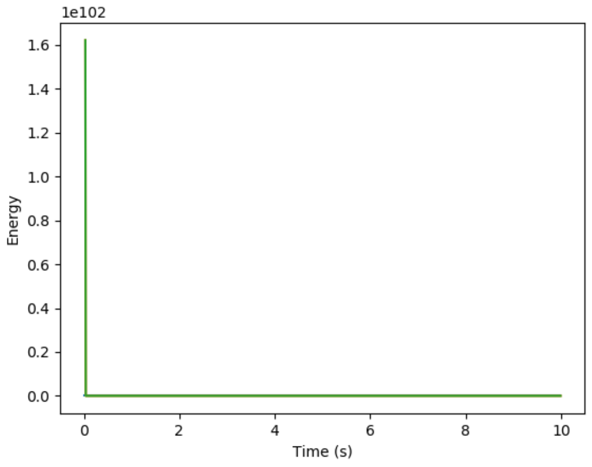

Figure 6: This plot shows the energy of the system. The energy in this plot starts very high and then drops to 0. Clearly the energy is not being conserved, which indicates a problem in how the interactions between particles in being calculated.

The code can be seen below.

```
import numpy as np
import matplotlib.pyplot as plt

#define constants
dt = 0.01
T = 1000
t1 = 0
t2 = 0 + dt*T
h = (t2-t1)/T
t = np.arange(t1,t2,h)

N = 16
Lx = 4.0
Ly = 4.0
dx = Lx/np.sqrt(N)
dy = Ly/np.sqrt(N)
x_grid = np.arange(dx/2, Lx, dx)
y_grid = np.arange(dy/2, Ly, dy)
xx_grid, yy_grid = np.meshgrid(x_grid, y_grid)
x_initial = xx_grid.flatten()
y_initial = yy_grid.flatten()


def radius(x_initial, y_initial):
    x_ij=np.zeros([N,N])
    y_ij=np.zeros([N,N])
    for i in range(len(x_initial)):
        for j in range(len(x_initial)):
             x_ij[j][i] =x_initial[j]-x_initial[i]
             y_ij[j][i]=y_initial[j]-y_initial[i]
    return x_ij, y_ij


def acceleration(x_ij, y_ij):
    acc_x=np.zeros([N,N])
    acc_y=np.zeros([N,N])
    acc_x_tot=np.zeros(N)
    acc_y_tot=np.zeros(N)
    for i in range(N):
        for j in range(N):
            if i==j:
                acc_x[j][i]=0
                acc_y[j][i]=0
            else:
                acc_x[j][i]=(-12 * (x_ij[j][i]**2) * ((1/(((x_ij[j][i])**2+(y_ij[j][i])**2)**4))-2/(((x
                acc_y[j][i]=(-12 * (y_ij[j][i]) * ((1/(((x_ij[j][i])**2+(y_ij[j][i])**2)**4))-2/(((x_ij
                acc_x_tot[j]+=acc_x[j][i]
                acc_y_tot[j]+=acc_y[j][i]
    return acc_x_tot,acc_y_tot
#make this be valid for j
x_ij, y_ij=radius(x_initial,y_initial)
```

```
print(acceleration(x_ij,y_ij))


def potential(x_ij,y_ij): #where x_ij, y_ij are distances between two particles like in acceleration
    pot_x=np.zeros([N,N])
    pot_y=np.zeros([N,N])
    pot_x_tot=np.zeros(N)
    pot_y_tot=np.zeros(N)
    for i in range(N):
        for j in range(N):
            if i==j:
                pot_x[j][i]=0
                pot_y[j][i]=0
            else:
                pot_x[j][i]=4*(1/(x_ij[j][i]**12)-1/(x_ij[j][i]**6))
                pot_x[j][i]=4*(1/(y_ij[j][i]**12)-1/(y_ij[j][i]**6))
            pot_x_tot[j]=np.sum(pot_x[j])
            pot_y_tot[j]=np.sum(pot_y[j])
    return pot_x_tot, pot_y_tot




x_pos=np.zeros([N,1000])
y_pos=np.zeros([N,1000])
x_pos[:,0]=x_initial #sets initial x positions as first column
y_pos[:,0]=y_initial
kx=np.zeros([N,1000])
ky=np.zeros([N,1000])
vx=np.zeros([N,1000])
vy=np.zeros([N,1000])
vx_half=np.zeros([N,1000])
vy_half=np.zeros([N,1000])
pot_ex=np.zeros([N,1000])
pot_ey=np.zeros([N,1000])
kin_ex=np.zeros([N,1000])
kin_ey=np.zeros([N,1000])
kin_ex_v=np.zeros([N,1000])
kin_ey_v=np.zeros([N,1000])


#for the first step only:
#set first element of every row of vx_half to:
vx_half[:,0]=0+1/2*h*acceleration(x_ij,y_ij)[0]
vy_half[:,0]=0+1/2*h*acceleration(x_ij,y_ij)[1]

#now do verlet
for i in range(1000-1):
    for j in range(N):
        x_pos[j,i+1]=x_pos[j,i]+h*vx_half[j,i]
        y_pos[j,i+1]=y_pos[j,i]+h*vy_half[j,i]

        kx[:,i+1]=h*np.array(acceleration(radius(x_pos[:,i],y_pos[:,i])[0], radius(x_pos[:,i],y_pos[:,i
        ky[:,i+1]=h*np.array(acceleration(radius(x_pos[:,i],y_pos[:,i])[0], radius(x_pos[:,i],y_pos[:,i

        vx[j,i+1]=vx_half[j,i]+0.5*kx[j,i+1]
        vy[j,i+1]=vy_half[j,i]+0.5*ky[j,i+1]

        vx_half[j,i+1]=vx_half[j,i]+kx[j,i+1]
        vy_half[j,i+1]=vy_half[j,i]+ky[j,i+1]

        pot_ex[:,i]=potential(radius(x_pos[:,i],y_pos[:,i])[0], radius(x_pos[:,i],y_pos[:,i])[1])[0]
        pot_ey[:,i]=potential(radius(x_pos[:,i],y_pos[:,i])[0], radius(x_pos[:,i],y_pos[:,i])[1])[1]

        kin_ex_v[j,i]=i+h*vx[j,i]
        kin_ey_v[j,i]=i+h*vy[j,i]
        kin_ex[j,i]=0.5*1* kin_ex_v[j,i]**2
        kin_ey[j,i]=0.5*1* kin_ey_v[j,i]**2

#%%
total_kin_j=kin_ex+kin_ey
total_pot_j=pot_ex+pot_ey

total_kin=np.sum(total_kin_j,axis=0)
total_pot=np.sum(total_pot_j,axis=0)
#%%
for i in range(N):
    plt.plot(x_pos[i],y_pos[i])
    plt.xlabel("x position")
    plt.ylabel("y position")


#%%

plt.plot(t,total_kin, label="kinetic")
plt.plot(t,total_pot, label="potential")
plt.plot(t,total_kin+total_pot, label="total energy")
plt.xlabel('time (s)')
plt.ylabel("Energy")
```

c)