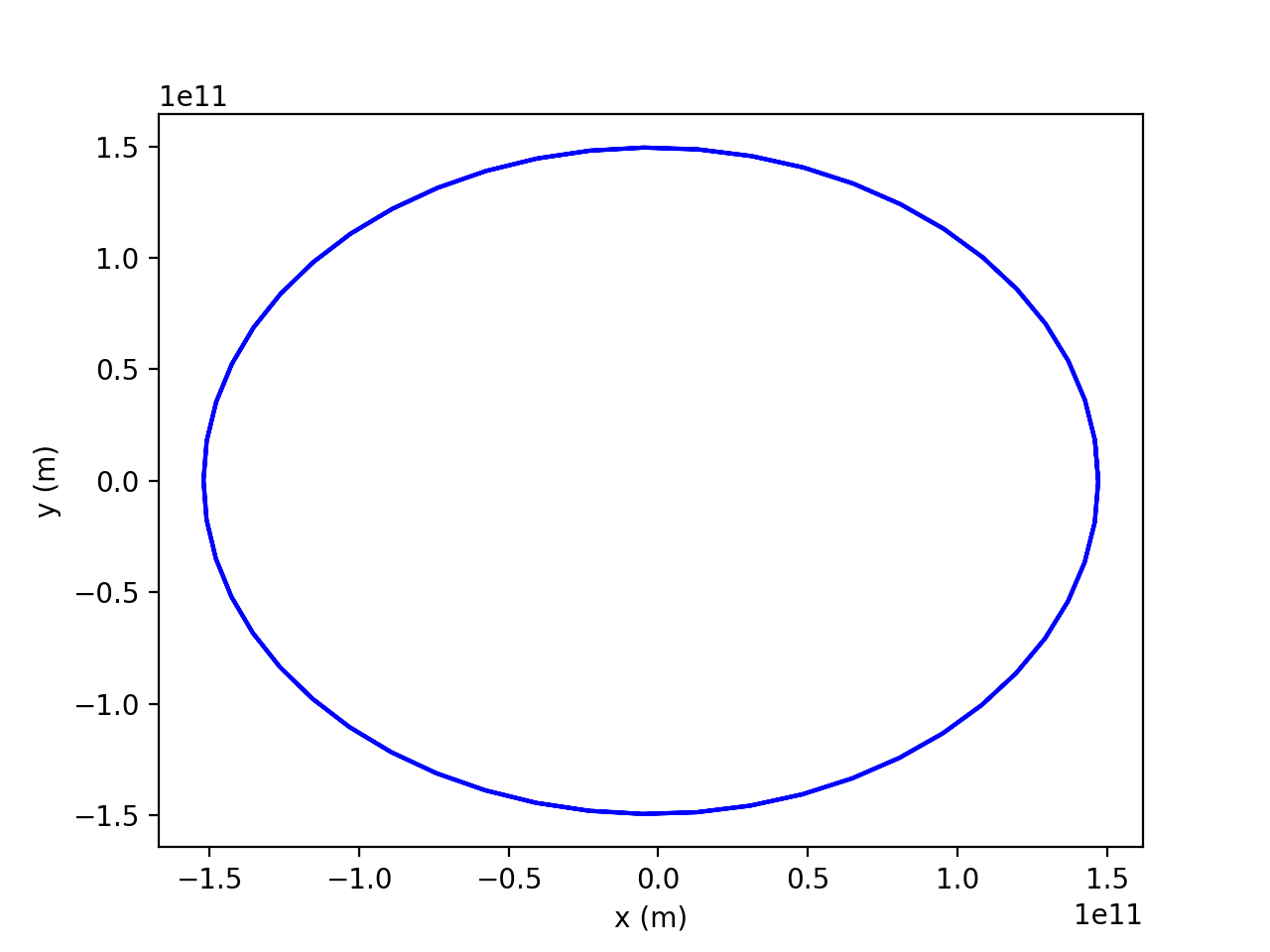
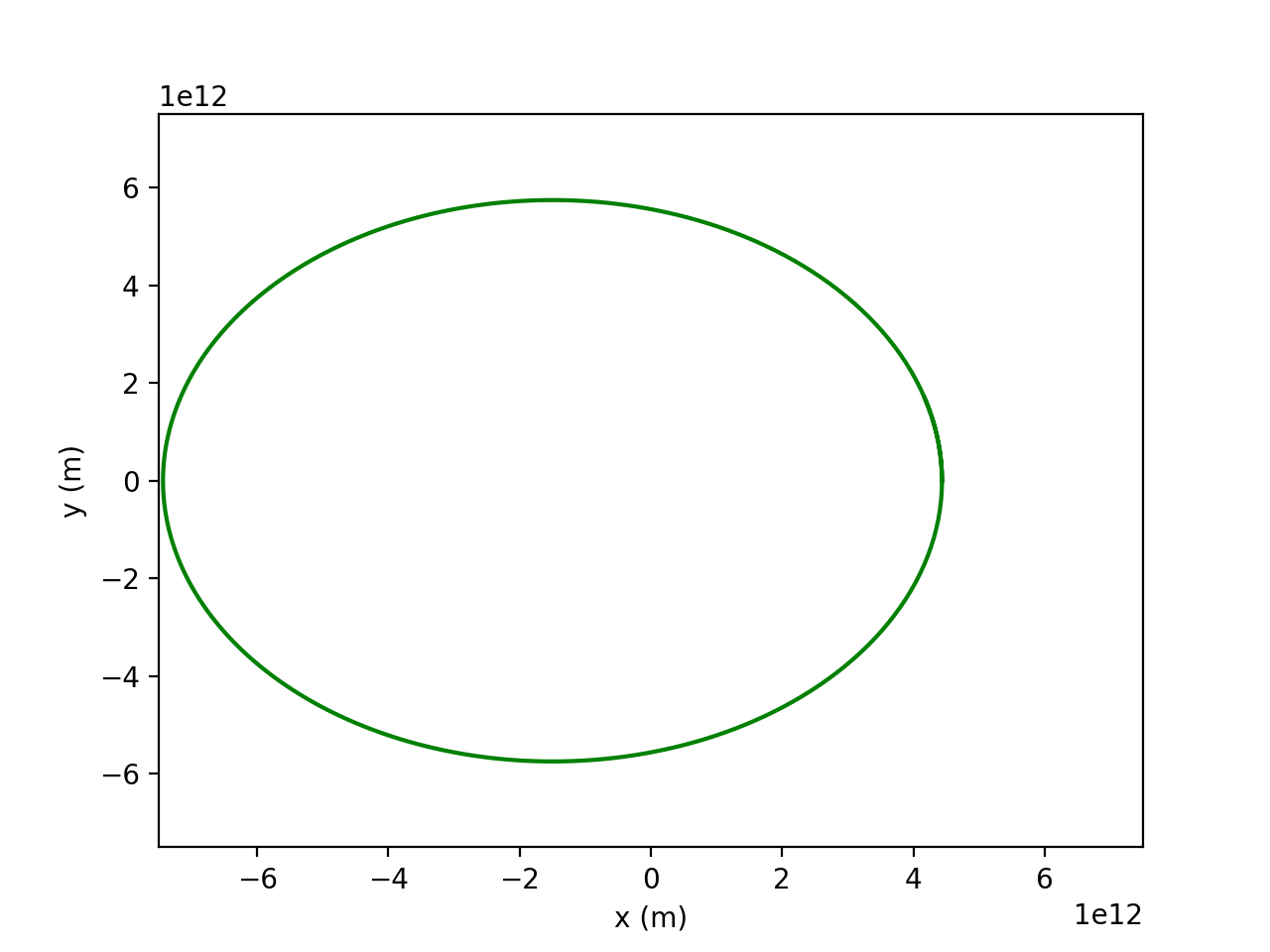
**Q1)**

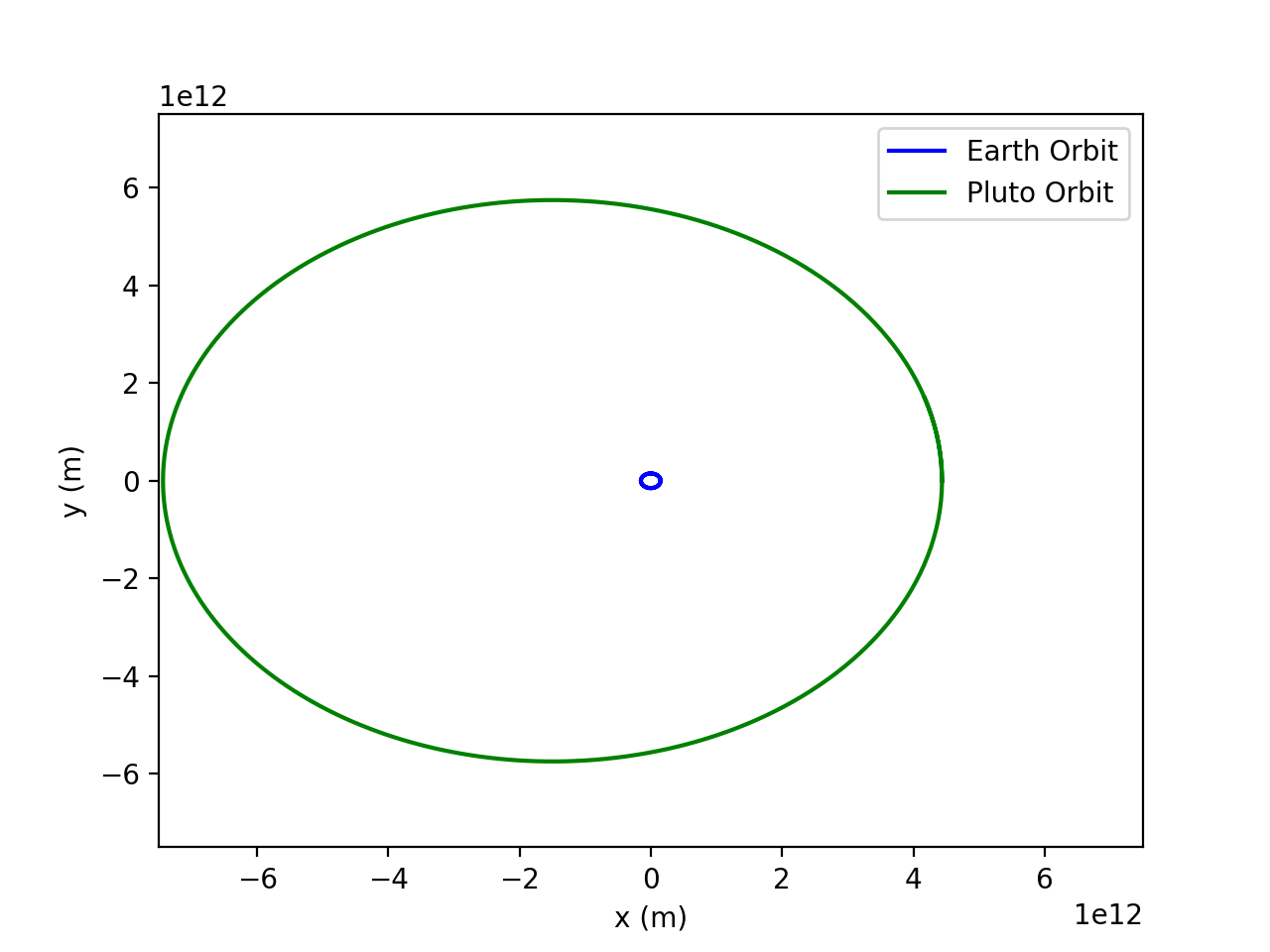
**a)**

The figure shows a plot of the Earth’s orbit after running the code.

**b)**

****

The figure shows Pluto’s orbit after inputting the code. The scale is different compared to the Earth’s plot, because Pluto’s orbit is much longer as expected. A combined plot to compare the relative orbital paths is shown in the figure below.



import numpy as np

import matplotlib.pyplot as plt

# Initializing and declaring variables

G = 6.6738e-11 \* ((24\*60\*60\*365)\*\*2) # Multiply by time in year in seconds

M = 1.9891 \* (10\*\*30)

H = 1 / 52 # 52 weeks in a year

x\_0 = 1.4710 \* (10\*\*11)

vx\_0 = 0

y\_0 = 0

vy\_0 = 3.0287 \* (10\*\*4) \* (24\*60\*60\*365)

delta = 1000 # 1km in meters

def fx(x, y):

return -G \* M \* x / np.sqrt(x \*\* 2 + y \*\* 2) \*\* 3

def fy(x, y):

return -G \* M \* y / np.sqrt(x \*\* 2 + y \*\* 2) \*\* 3

def f(r):

x = r[0]

vx = r[1]

y = r[2]

vy = r[3]

return np.array([vx, fx(x, y), vy, fy(x, y)], float)

def Bulirsch\_Stoer\_step(r, H):

def modified\_midpoint\_step(r, n):

r = np.copy(r)

h = H / n

k = r + 0.5 \* h \* f(r)

r += h \* f(k)

for i in range(n - 1):

k += h \* f(r)

r += h \* f(k)

return 0.5 \* (r + k + 0.5 \* h \* f(r))

target\_accuracy = H \* delta

def compute\_row\_n(R1, n):

def R\_n\_m(m):

return R2[m - 2] + (R2[m - 2] - R1[m - 2]) / ((n / (n - 1)) \*\* (2 \* (m - 1)) - 1)

# Compute R\_n,1

R2 = [ modified\_midpoint\_step(r, n) ]

# Compute the rest of the row

for m in range(2, n + 1):

R2.append(R\_n\_m(m))

# Convert to array to compute error

R2 = np.array(R2, float)

error\_vector = (R2[n - 2] - R1[n - 2]) / ((n / (n - 1)) \*\* (2 \* (n - 1)) - 1)

error = np.sqrt(error\_vector[0] \*\* 2 + error\_vector[2] \*\* 2)

if error < target\_accuracy:

return R2[n - 1]

else:

return compute\_row\_n(R2, n + 1)

return compute\_row\_n(np.array([modified\_midpoint\_step(r, 1)], float), 2)

plt.figure()

# Calculate Earth's orbit

xpoints = []

ypoints = []

r = np.array([x\_0, vx\_0, y\_0, vy\_0], float)

for i in range (2 \* 53):

xpoints.append(r[0])

ypoints.append(r[2])

r = Bulirsch\_Stoer\_step(r, H)

plt.plot(xpoints, ypoints, 'b', label= 'Earth Orbit')

plt.xlabel('x (m)')

plt.ylabel('y (m)')

# plt.show()

# Calculate Pluto's orbit

xpoints = []

ypoints = []

r = np.array([4.4368e12, 0, 0, 6.1218e3 \* 8760 \* 60 \* 60], float)

H = 1 # year

for i in range (260):

xpoints.append(r[0])

ypoints.append(r[2])

r = Bulirsch\_Stoer\_step(r, H)

plt.plot(xpoints, ypoints, 'g', label = 'Pluto Orbit')

plt.xlim(-7.5e12, 7.5e12)

plt.ylim(-7.5e12, 7.5e12)

plt.xlabel('x (m)')

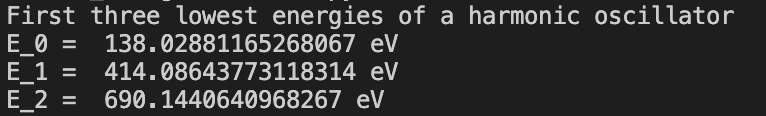
plt.ylabel('y (m)')

plt.legend()

plt.show()

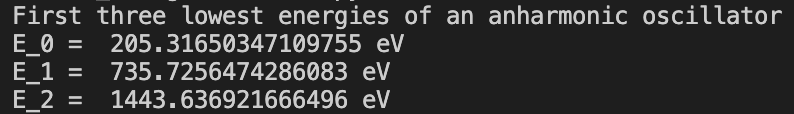
**Q2)**

**a)**

****

The figure above shows the first three lowest energies of a harmonic oscillator. The 4th order Runge-Kutta method was used along with the Simpson’s Rule. Despite not knowing the exact values, the energies seem to be correct, because the energies are increasing, and the difference between each energy level is decreasing as expected.

**b)**



The figure above is the output for the three lowest energies of an anharmonic oscillator. The energy values are different from the harmonic oscillator as expected, and the general trend of increasing energy values is the same. However, the difference between each energy level is increasing unlike the harmonic oscillator, and it seems to be correct in the case of an anharmonic oscillator to behave differently compared to a harmonic oscillator.

import numpy as np

import matplotlib.pyplot as plt

# Initial variables and constants

e = 1.602 \* (10 \*\* -19)

V0 = 50 \* e # J

a = 10 \*\* -11 # Angstrom

x\_0 = -10 \*\* -10

x\_f = 10 \*\* -10

psi\_0 = 0.0

hbar = 1.05457 \* (10 \*\* -34) # J\*s

m = 9.10938 \* (10 \*\* -31) # electron mass in kg

N = 1000 # number of steps to use in Runge-Kutta

h = (x\_f - x\_0) / N

# def V(x):

# return V0 \* x \*\* 2 / a \*\* 2

def V(x):

return V0 \* x \*\* 4 / a \*\* 4

def psi(E):

def f(r, x):

psi = r[0]

phi = r[1]

return np.array([phi, (2 \* m / hbar \*\* 2) \* (V(x) - E) \* psi], float)

r = np.array([psi\_0, 1.0] ,float)

wavefunction = []

for x in np.arange(x\_0, x\_f, h):

wavefunction.append(r[0])

k1 = h \* f(r, x)

k2 = h \* f(r + 0.5 \* k1, x + 0.5 \* h)

k3 = h \* f(r + 0.5 \* k2, x + 0.5 \* h)

k4 = h \* f(r + k3, x + h)

r += (k1 + 2 \* k2 + 2 \* k3 + k4) / 6

return np.array(wavefunction, float)

def secant\_root(E1, E2):

target\_accuracy = e / 1000

wavefunction = psi(E1)

psi2 = wavefunction[N - 1]

while abs(E1 - E2) > target\_accuracy:

wavefunction = psi(E2)

psi1, psi2 = psi2, wavefunction[N - 1]

E1, E2 = E2, E2 - psi2 \* (E2 - E1) / (psi2 - psi1)

#Simpson's Rule

mod\_squared = wavefunction \* wavefunction

integral = h / 3 \*(mod\_squared[0] + mod\_squared[N//2 - 1] + \

4 \* sum(mod\_squared[1 : N//2 : 2]) + 2 \* sum(mod\_squared[0 : N//2 + 1 : 2]) )

return (E2 / e), (wavefunction / np.sqrt(2\*integral))

# Harmonic Oscillator

# print('First three lowest energies of a harmonic oscillator')

# print('E\_0 = ', secant\_root(0, 0.5\*e)[0], 'eV')

# print('E\_1 = ', secant\_root(200\*e, 400\*e)[0], 'eV')

# print('E\_2 = ', secant\_root(500\*e, 700\*e)[0], 'eV')

# Anharmonic Oscillator

print('First three lowest energies of an anharmonic oscillator')

print('E\_0 = ', secant\_root(0, 0.5\*e)[0], 'eV')

print('E\_1 = ', secant\_root(400\*e, 600\*e)[0], 'eV')

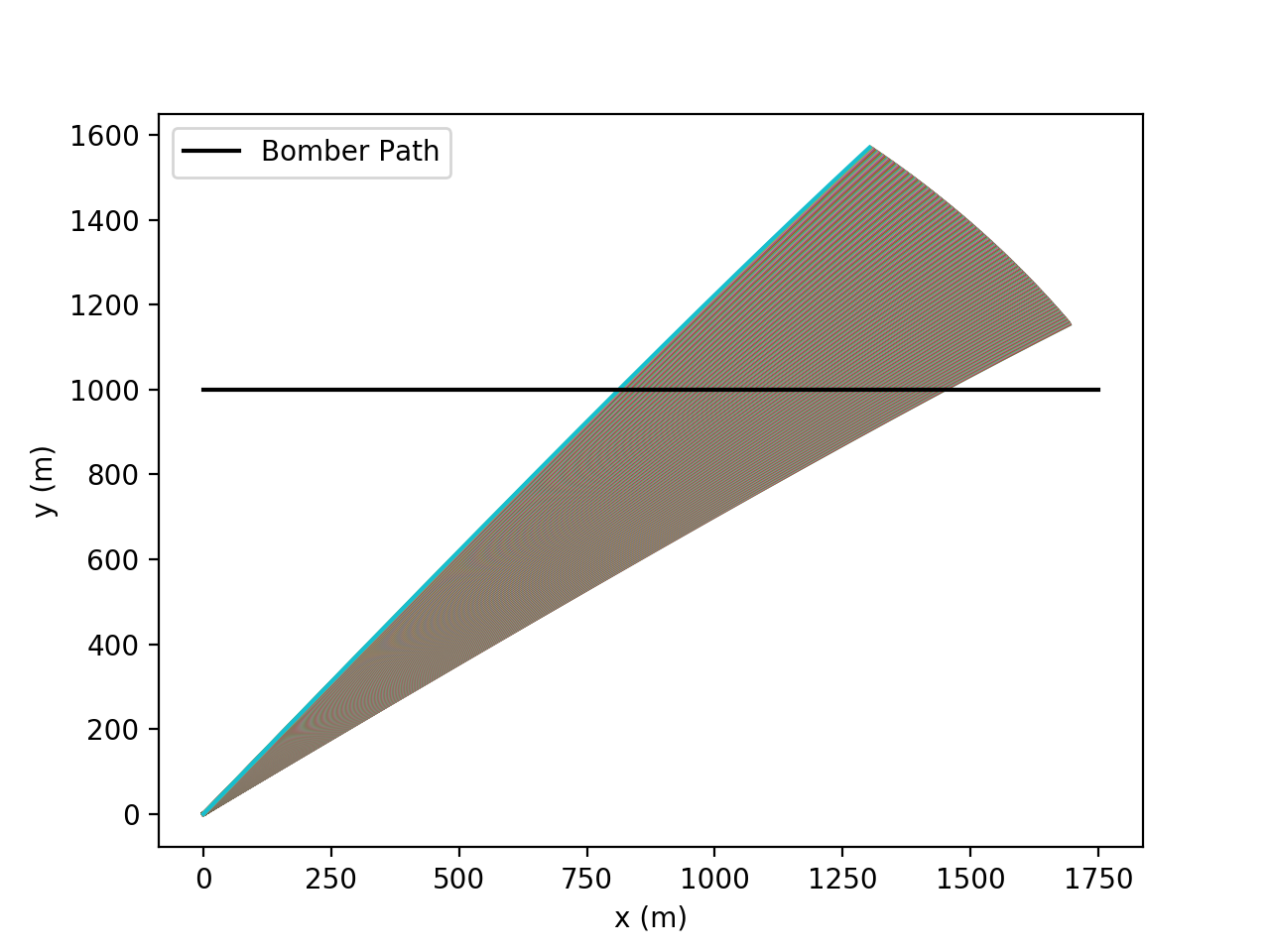
print('E\_2 = ', secant\_root(900\*e, 1100\*e)[0], 'eV')

**Q3)**

The equations for motion of the cannon ball can be derived using Newton’s Second Law and taking into account the air resistance:

Then the problem becomes a problem to change these two second-order equations into first-order equations and solving them. Using the information provided, the coefficients and constants can be substituted with the given values. Since the velocity of the cannonball and the angle of the cannon is not given, we have to find it. After testing for multiple values of the velocity of the cannonball, small values such as 150m/s did not output a plot where the cannonball could even reach 1000m to hit the bomber plane. Therefore, a larger value of around 1000m/s seemed to be a good estimate for this case (very unrealistic, but otherwise, the cannonball could not reach 1000m. Also, Newton’s cannonball experiment had his cannonball travelling at 11200m/s, so as a theoretical value, 1000m/s seems reasonable.) The angle could be calculated by simple trigonometry:

After finding the necessary angle values ranging from 4~8 seconds after the bomber is seen, the code had been structured. Using 4th order Runge-Kutta method to calculate the first order equations, the trajectory plot could be plotted for a range of theta values (600 values) ranging from (the angles were of course converted into radians). Assuming the cannonball can shoot extremely fast and accurately, the path of the bomber and the cannonball can be plotted in the same plot to show when they intercept.



Although the plot seems very clean and intuitive, a lot of non-realistic assumptions had to be made to output this plot. However, the physics behind the cannonball is real, and in the given question’s context, this plot should suffice as a predictive model of hitting the bomber with a cannonball.

import numpy as np

import matplotlib.pyplot as plt

# Initial constants

g = 9.8 #m/s^2

m = 10 # kg

R = 0.08 # m

theta\_array = []

v\_0 = 1000 # m/s

rho = 1.22 # kg/m^3

C = 0.47 # drag coefficient

t\_0 = 4

t\_f = 8

N = 1000

h = (t\_f - t\_0) / N #interval smaller than 0.5s

for i in range (1400,800,-1):

theta\_array.append(np.arctan([1000/i])[0])

c = np.pi \* R \*\* 2 \* rho \* C / 2

def combined\_constant(m):

return c/m

def f(r, t, m):

# x = r[0]

vx = r[1]

# y = r[2]

vy = r[3]

v = np.sqrt(vx \*\* 2 + vy \*\* 2)

return np.array([vx, - combined\_constant(m) \* vx \* v,

vy, -g - combined\_constant(m) \* vy \* v], float)

tpoints = np.arange(t\_0, t\_f, h)

def trajectory(m, theta\_0):

xpoints = []

ypoints = []

r = np.array([0, v\_0 \* np.cos(theta\_0), 0, v\_0 \* np.sin(theta\_0)], float)

for t in tpoints:

xpoints.append(r[0])

ypoints.append(r[2])

k1 = h \* f(r, t, m)

k2 = h \* f(r + 0.5 \* k1, t + 0.5 \* h, m)

k3 = h \* f(r + 0.5 \* k2, t + 0.5 \* h, m)

k4 = h \* f(r + k3, t + h, m)

r += (k1 + 2 \* k2 + 2 \* k3 + k4) / 6

return np.array(xpoints, float), np.array(ypoints, float)

# trajectory\_x\_array = []

# trajectory\_y\_array = []

#plotting the projectile

for theta\_0 in range(len(theta\_array)):

# for theta\_0 in range(10):

trajectory\_x = trajectory(m,theta\_array[theta\_0])[0]

trajectory\_y = trajectory(m,theta\_array[theta\_0])[1]

plt.plot(trajectory\_x,trajectory\_y)

# plot a horizontal line to show the intersection of the bomber path and the cannonball

plt.plot([0,1750],[1000,1000], 'k-', label ='Bomber Path')

plt.xlabel('x (m)')

plt.ylabel('y (m)')

plt.legend()

plt.show()