A Study of Trajectories and The Effects of Friction Upon Them

David Lawton

17th November 2023

Contents

1 Abstract

In this experiment, our objective was to model and analyse the force acting upon, and the trajectory of, a small spherical object (Diameter of order 10^-4) under gravitational acceleration 'g'. We used various computational methods and approximations throughout, finding what I would describe as quite interesting results to conclude a standard computational experiment.

2 Introduction

In this study, the trajectories of different masses were investigated, with and without dissipative forces.

We began by creating a function to evaluate the friction force on a spherical particle, using the below equations.

$$\mathrm{f}(v) = \mathrm{b} \cdot |\vec{V}| + \mathrm{c} \cdot \vec{V}^2 \tag{1}$$

$$\vec{F} = -\mathrm{f}(v) \cdot \hat{V}, \quad \hat{V} = \frac{\vec{V}}{|\vec{V}|} \tag{2}$$

The constants 'b' and 'c' depend on the size and shape of the object, where $\mathrm{b} = \mathrm{B} \cdot D,\ \mathrm{c} = \mathrm{C} \cdot D^2$. 'B' and 'C' are constraints of the medium through which the projectile flies. We also evaluated the ranges for which the quadratic and linear terms can and cannot be included.

Next, we modelled the velocity of a particle's vertical velocity as a function of time, dropped from rest. The differential equation dictating the linear approximation of the vertical velocity is below, which we found to be an apt approximation for our case when we obtained the terminal velocity of our falling spherical particle.

$$\frac{\mathrm{d}V_\mathrm{y}}{\mathrm{d}t} = -\mathrm{g} - \frac{\mathrm{b}}{\mathrm{m}} \cdot V_\mathrm{y} \tag{3}$$

There is also an analytic solution to this equation, found easily using the integration factor method, which the values of the computational solution were analysed against, with the error of the computational being the difference between the two solutions.

$$V_\mathrm{y} = \frac{\mathrm{mg}}{\mathrm{b}}(e^{-\mathrm{b}t/\mathrm{m}} - 1) \tag{4}$$

The following step was to add a velocity component in the x-direction, and plot the trajectory of the particle with the linear approximation, this required the formula below,

$$\frac{\mathrm{d}\vec{V}}{\mathrm{d}t} = -\frac{\mathrm{b}}{\mathrm{m}} \cdot \vec{V} - \mathrm{g}\hat{y} \tag{5}$$

which leads to the following two formulae.

$$\frac{\mathrm{d}V_x}{\mathrm{d}t} = -\frac{\mathrm{b}}{\mathrm{m}} \cdot V_x, \quad \frac{\mathrm{d}V_y}{\mathrm{d}t} = -\mathrm{g} - \frac{\mathrm{b}}{\mathrm{m}} \cdot V_y \tag{6}$$

A function was created to find the X-value when the projectile lands based on these, from which the optimum angle for distance was calculated and plotted for a range of masses.

Finally, the trajectory under the quadratic approximation was calculated and plotted against the linear approximation and the frictionless particle. The motion of the particle under the quadratic approximation is dictated by the formula below,

$$\frac{\mathrm{d}\vec{V}}{\mathrm{d}t} = -\frac{\mathrm{c}}{\mathrm{m}} \cdot |\vec{V}| \cdot \vec{V} - g\hat{y} \tag{7}$$

which, like the linear approximation is seperated into two components.

$$\frac{\mathrm{d}V_x}{\mathrm{d}t} = -\frac{\mathrm{c}}{\mathrm{m}} \cdot |\vec{V}| \cdot V_x, \quad \frac{\mathrm{d}V_y}{\mathrm{d}t} = -\mathrm{g} - \frac{\mathrm{c}}{\mathrm{m}} \cdot |\vec{V}| \cdot V_y \tag{8}$$

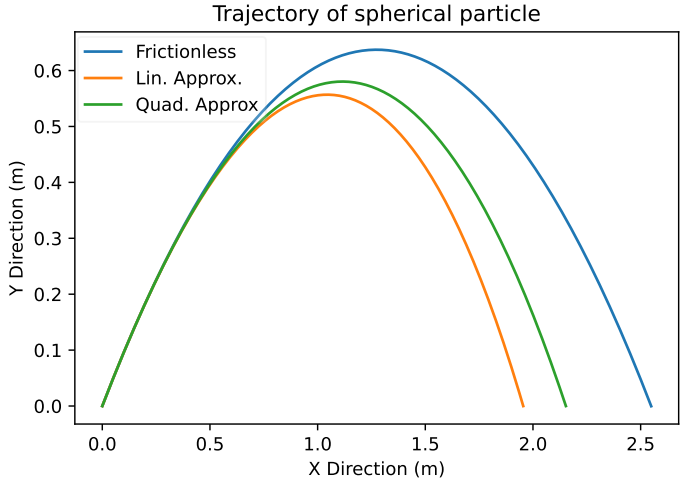


Figure 1: A trajectories with different approximations of dissipation.

3 Methodology

3.1 Force Evaluation and Approximation

The first section of the method, focusing on the dissipative force acting on a particle as in Eqs. 1, 2, is quite short and simple. A function was created to evaluate the instantaneous force acting on a spherical particle moving in a vertical fashion, based on entered values of diameter, velocity and the system constraints, 'b' and 'c'.

```
def f(v, b, c, d):
```

```
    p = b * d
    q = c * d**2
    f = p * v + q * (v ** 2)
    return f
```

As this force can be made into a function of $\mathrm{D} \cdot V$, arrays were made ranging various magnitudes, which represented the values of $\mathrm{D} \cdot V$, and the force was seperated into seperate linear and quadratic parts. The quadratic and linear parts of the force were stored in lists, for values across the range of the arrays, giving us ranges for which the force could be decently approximated to one of the approximations, and for which neither approximation was applicable.

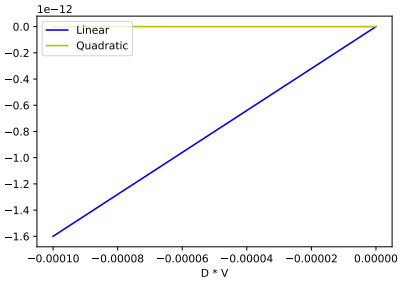
```
Kvals = np.arange(i, j, (j-i)*10**-6)
f1vals = []
f2vals = []


def f1(b, c):
    for K in Kvals:
        f1 = b * K
        f2 = c * K**2
        f1vals.append(f1)
        f2vals.append(f2)
```

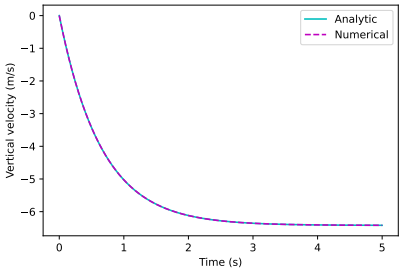
This was then done for specific cases also, with several different cases of particular diameter and velocity.

3.2 Velocity Evaluation and Approximation

In this section we deal with the case of a small spherical projectile with a y-velocity dictated by Eq. 3. For this spherical projectile, we approximated which term could be ignored by finding the terminal velocity 'V_T' of the particle to six decimal places. Given that this value of V_T is the maximum by definition, all other values of *Diameter* $\times$ *Speed* are smaller, and, in this case, the quadratic term can be ignored, as can be seen in Figure 2.



(A) *Diameter* × *speed* at order of V_T.



(B) Numerical vs. analytical integration.

Figure 2: Plots for subsection 3.2

Of course for the linear approximation, there is an analytic solution to Eq. 3, which is Eq. 4. This analytic solution was then illustrated versus the numerical solution in figure 2 (B). Below is the code used to plot the analytic solution.

```
Vyvals2 = []
timesteps = np.arange(0, 5, 0.0001)


def func2(p, q):
    t = 0
    dt = 0.0001
    g = 9.81
    for t in timesteps:
        Vy = ((q*g) / p) * (np.exp(-p*t/q)-1)
        t = t + dt
        Vyvals2.append(Vy)
```

The analytic solution of course is more accurate, so the next step was to plot the error of the numerical solution against time.
The function of course varies with mass, so the following course of action was to see the change in fall time with density, for a constant object diameter and initial height. To compute these values a function was made to calculate the time taken to reach the ground, and used it over a logarithmically spaced array of densities, producing a list of fall times.

```
rho_vals = np.logspace(1, 5, 120)


def time(H, p):
    timesteps = np.arange(0, 4, 0.00001)
    VyR = 10
    t = 0
```

```
    dt = 0.0001
    YR = 5
    for i in range(len(timesteps)-1):
        if YR < 0:
            break
        YR = YR + VyR * dt
        deltaVyR = -9.81 * dt - (B/(p*Vol)) * VyR * dt
        VyR = VyR + deltaVyR
        t = t + dt
    return t


fall_times = []
for p in rho_vals:
    tfall = time(5, p)
    fall_times.append(tfall)
```

3.3 Trajectory

This section deals with plotting and analysing the actual trajectory of the particle.

3.3.1 Plotting

For the sake of simplicity, the trajectory was plotted for a horizontal plane. To do this, lists were created, which were filled with the components of position (x, y), as the velocity changed over the projectile motion. This was done for two cases, the linear approximation and the frictionless case. The code below was for the linear case, as I thought the frictionless too simple to be woth adding. Regardless it is in the appendix anyway.

```
timesteps = np.arange(0, 4, 0.0001)

VyRvals = [10]
VxRvals = [4]
VyR = 10
VxR = 4
t = 0
dt = 0.0001
XR = 0
YR = 0
XRvals = [0]
YRvals = [0]
for i in range(len(timesteps)-1):
    if YR < 0:
```

```
        break
    XR = XR + VxR * dt
    YR = YR + VyR * dt
    deltaVyR = -9.81 * dt - (B/m) * VyR * dt
    deltaVxR = -(B/m) * VxR * dt
    VyR = VyR + deltaVyR
    VxR = VxR + deltaVxR
    t = t + dt
    VyRvals.append(VyR)
    VxRvals.append(VxR)
    XRvals.append(XR)
    YRvals.append(YR)
```

Of course in this case, the y coordinate is plotted against the x coordinate, resulting in a plot similar to figure 1, but without the quadratic case.

3.3.2 Analysis

Our analysis of the motion began by creating a function to produce the maximum distance in the x-direction before the particle hits the ground, for a given object density and initial velocity ($|\vec{V}| \cdot \cos(\theta)$, $|\vec{V}| \cdot \sin(\theta)$), as below.

```
def xmax(rho, D, v, dt, theta):
    Vol = (4/3) * np.pi * (D/2)**3
    m = rho * Vol
    B = 1.6*(10**-4)*D
    timesteps = np.arange(0, 4, dt)

    VyR = v*np.sin(theta)
    VxR = v*np.cos(theta)
    t = 0
    XR = 0
    YR = 0

    for i in range(len(timesteps)-1):
        if YR < 0:
            return XR
            break
        XR = XR + VxR * dt
        YR = YR + VyR * dt
        deltaVyR = -9.81 * dt - (B/m) * VyR * dt
        deltaVxR = -(B/m) * VxR * dt
        VyR = VyR + deltaVyR
        VxR = VxR + deltaVxR
        t = t + dt
```

This function uses the same code as in 3.3.1, but instead of saving the coordinates, just produces the final x-value. The other benefit of this function of course is that has density and angle as variables of the function. It then followed to find the optimum angle, the angle which produces the largest maximum x-value, for a range of masses, and plot the result.

To do this we first had to create a function which found the optimum angle, θ' for a given density. This function utilised the previous maximum x-value function.

```
def opt_theta(rho, D, v, dt):
    theta = np.arange(0.01, np.pi/2, 0.01)
    FXvals = []
    for the in theta:
        if the == 0.001:
            X = xmax(rho, D, v, dt, the)
            FXvals.append(X)
        if the != 0.001:
            X = xmax(rho, D, v, dt, the)
            FXvals.append(X)
            if X != max(FXvals):
                return the
                break
```

The function finds the maximum x-value for each theta, and breaks once θ starts decreasing, returning the θ that produced the largest x-value. This was then acted over a range of logarithmically scaled density values.

```
rho_vals = np.logspace(1.0, 7.0, num=15, endpoint=True)
opt_t = []
for p in rho_vals:
    opt_t.append(opt_theta(p, 10**-4, 10, 0.00001))
```

Finally, we graphed the produced list against the array of densities on a logarithmic scale.

3.4 Quadratic Approximation

The last portion of the report, revolving around the quadratic approximation described in Eqs. 7, 8, has one large difference in methodology to the linear approximation. This is that each component of the velocity depends on the speed of the projectile. This requires an extra term inside the 'for' loop for time, the term for total velocity '$|\vec{V}|$'. This is evaluated after its commponents are changed at each timestep, and then used in the next timestep to evaluate the next value of it's components.

```
timesteps = np.arange(0, 10, 0.001)
dt = 0.0001
t = 0
```

```
theta = np.pi/4
v = 5
VyRq = v * np.sin(theta)
VxRq = v * np.cos(theta)
VyRqvals = [VyRq]
VxRqvals = [VxRq]
XRq = 0
YRq = 0
XRqvals = [XRq]
YRqvals = [YRq]
for i in range(len(timesteps)-1):
    if YRq < 0:
        break
    XRq = XRq + VxRq * dt
    YRq = YRq + VyRq * dt
    Vq = np.sqrt(VxRq ** 2 + VyRq ** 2)
    deltaVyRq = -9.81 * dt - (A/m) * Vq * VyRq * dt
    deltaVxRq = -(A/m) * Vq * VxRq * dt
    VyRq = VyRq + deltaVyRq
    VxRq = VxRq + deltaVxRq
    t = t + dt
    VyRqvals.append(VyRq)
    VxRqvals.append(VxRq)
    XRqvals.append(XRq)
    YRqvals.append(YRq)
```

This was then plotted along with the linear and frictionless cases, producing, figure 1.

4 Results

4.1 Result 1

The first observed result of this experiment revolved around the error between the computational and analytic solutions to Eq. 3. As can be seen in figure 3, the computational integration, for the used time-step, was quite accurate, with little noticable error. However the plot of the error over time, figure 3, gives in an interesting result.

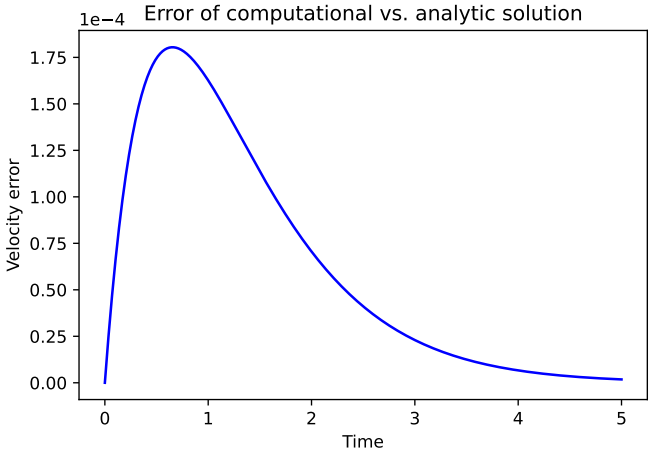


Figure 3: Error of computational versus numerical solution

As can be seen here, the error of the computational result increases fairly linearly initially. However, the error actually begins to decrease after a time. And, if compared to figure 2, this happens as the velocity begins to approach V_T.

4.2 Result 2

The next result found concerned the time taken, to fall to ground, of projectiles of constant diameter, initial height, and varying density.

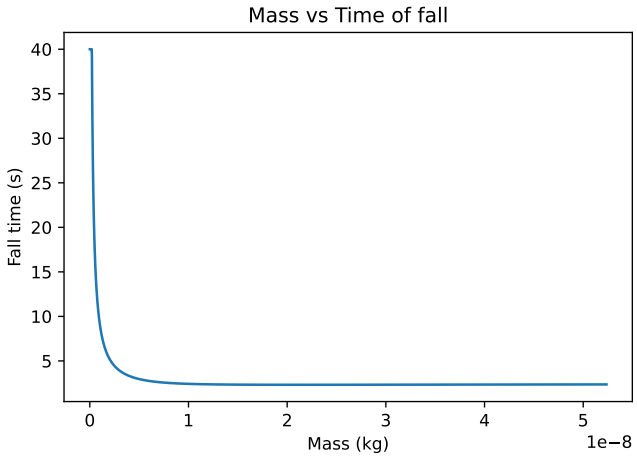


Figure 4: Mass vs. Time of fall

This shows that in reality, not all masses fall at the same rate, as is often presumed in physics.

4.3 Result 3

Our third result deals with the trajectory of the particle. Under the linear approximations,as expected, the particle always travels a shorter distance along the horizontal plane, and has a lower peak, as in figure 5.

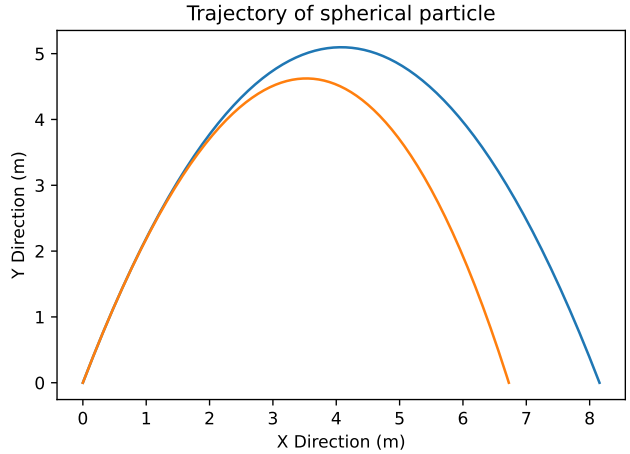
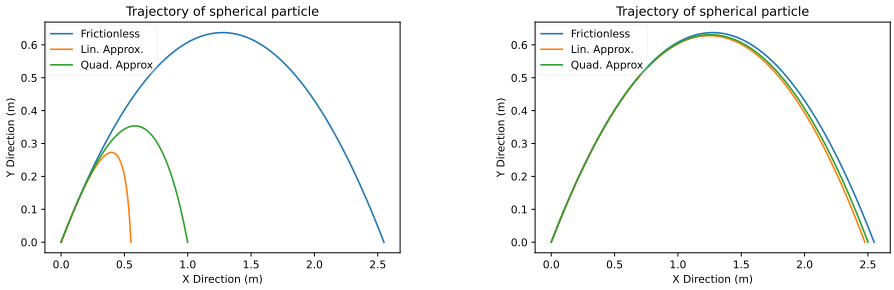


Figure 5: Trajectory of our projectile under a linear approximation of the friction force.

Next, when we considered the quadratic approximation for the same particle, and compared its trajectory with our previous cases, the following plots in figure 6 along with figure 1 are produced.



(A) Lower density (B) Higher density

Figure 6: Trajectories for different densities, under different approximations.

As can be seen in these plots, the quadratic approximation always produces a trajectory in between the linear and frictionless cases.

In reality, the particle doesnt travel as far as either of these trajectories.

4.4 Result 4

The final result concerns how the optimum angle for distance varies with mass. Given the nature of the result, I found it best represented in a logarithmic scale as below.

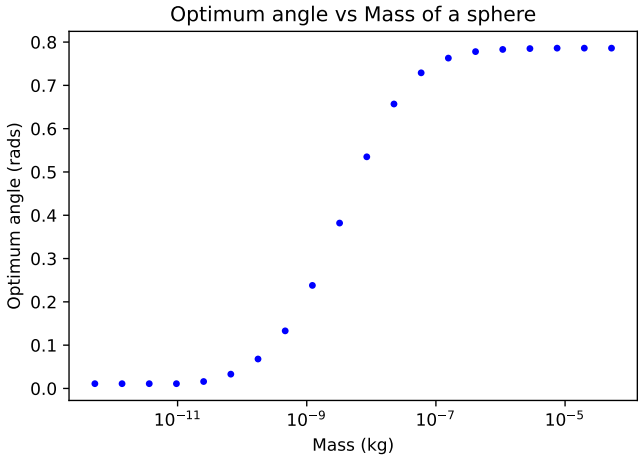


Figure 7: Optimum angle (for distance in the horizontal direction) vs. Mass.

The value of θ approaches 0 radians at the lower magnitudes, and about $\frac{\pi}{4}$ radians at the higher magnitudes.

5 Conclusion

There are several conclusions that can be taken from this experiment.

Firstly, the error of computational integration is asymptotic as a function converges to a constant value. That is to say the computational result approaches the actual value of the function to a high degree, as observed in figures 3, 2(B)

Secondly, as can be observed in figure 4, the time taken to reach the ground is highly variable with respect to time at low masses. The usual approximation of constant fall time is only a good approximation at masses greater than about 2×10^{-8} kilograms.

Thirdly, and finally, the way in which the optimum angle changes with mass has quite strange behaviour, as illustrated in figure 7. In that case (for a diameter of 10**-4), the optimum angle approaches zero as the order of magnitude of the mass decreases, and approaches $\frac{\pi}{4}$ as the order of magnitude increases, with the optimum angle increasing approximately logarithmically between the

two. I would conclude that this is because as as the mass grows larger, the dissipative force becomes negligible, and as the mass becomes small, the initial motion becomes negligble, and the particle becomes static.

6 Bibliography

2023. 2nd year Physics Lab Manual. School of Physics, Trinity College Dublin, Dublin 2.

7 Appendix

7.1 Force Evaluation

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

# V is velocity
# f(V) is the magnitude of the force resisting the velocity.
# b, c are factors depending on the medium
# d is the diameter of the projectile.

Kvals = np.arange(0.0009, 0.0011, 10**-7)
f1vals = []
f2vals = []

def f(a, b, c, d):
    p = b * d
    q = c * d**2
    f = p * a + q * (a ** 2)
    return f

def f1(b, c):
    for K in Kvals:
        f1 = b * K
        f2 = c * K**2
        f1vals.append(f1)
        f2vals.append(f2)

f1(1.6*(10**-4), 0.25)
print(f1vals)

plt.figure()
plt.xlabel('D * V')
plt.plot(Kvals, f1vals, 'b-', markersize=0.3)
plt.plot(Kvals, f2vals, 'y-', markersize=0.3,)
plt.legend(['b * V', 'c * V^2'], loc='upper left' )
# ================================================================
# plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/linVSquad2.pdf'
# ================================================================
plt.show()
plt.close()

#quad term negligible for D*V < ~0.0001
#both taken into account for ~0.0001 < D*V < ~0.005
```

```
#lin term negligible for D*V > ~0.005

#Baseball Case
D = 0.07
V = 5
print(D*V)
#optimal case for baseball: take only quad term, lin neg.

#Oil droplet case
D = 1.5*(10**-6)
V = 5*(10**-5)
print(D*V)
#optimal case for oil droplet: take only linear term, quad neg.

#Raindrop case
D = 1*(10**-3)
V = 1
print(D*V)
#optimal case for raindrop: take both terms into account, neither neg.
```

7.2 Velocity Evaluation

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

# spherical grain of dust
rho = 2*(10**4)
D = 10**-4
Vol = (4/3) * np.pi * (D/2)**3
m = rho * Vol

F = - m * 9.81
print(F)

root1 = 0
root2 = 0

A = 0.25*D**2
B = 1.6*(10**-4)*D
C = F
root1 = (-B + np.sqrt(B ** 2 - 4*A*C)) / (2 * A)
root2 = (-B - np.sqrt(B ** 2 - 4 * A * C)) / (2 * A)

print(root2)
```

```
# take positive root
Kvals = np.arange(0, -1e-04, -1e-06)
f1vals = []
f2vals = []


def f(a, b, c, d):
    p = b * d
    q = c * d**2
    f = p * a + q * (a ** 2)
    return f


def f1(b, c):
    for K in Kvals:
        f1 = b * K
        f2 = c * K**2
        f1vals.append(f1)
        f2vals.append(f2)


f1(B, A)

Vyvals = [0]
timesteps = np.arange(0, 5, 0.0001)


def func(p, q):
    Vy = 0
    t = 0
    dt = 0.0001
    for i in range(len(timesteps)-1):
        deltaVy = -9.81*dt - (p/q)*Vy*dt
        Vy = Vy + deltaVy
        t = t + dt
        Vyvals.append(Vy)


Vyvals2 = []


def func2(p, q):
    t = 0
    dt = 0.0001
    g = 9.81
    for t in timesteps:
```

```
        Vy = ((q*g) / p) * (np.exp(-p*t/q)-1)
        t = t + dt
        Vyvals2.append(Vy)


func(B, m)
func2(B, m)
Vyvals2 = np.array(Vyvals2)
Vyvals = np.array(Vyvals)

V_T = -0.642062
print(D*V_T)
# much less than 0.0001, quad term negligible

plt.figure()
plt.xlabel('D * V')
plt.plot(Kvals, f1vals, 'b-', markersize=0.3)
plt.plot(Kvals, f2vals, 'y-', markersize=0.3,)
plt.legend(['Linear', 'Quadratic'], loc='upper left')
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex2a/Vert1.pdf')
plt.show()
plt.close()
# Clearly quadratic term can be ignored


plt.figure()
plt.xlabel('Time (s)')
plt.ylabel('Vertical velocity (m/s)')
# ================================================================
# plt.axis([0, 0.26, -1, 0])
# ================================================================
plt.plot(timesteps, Vyvals2, '-c')
plt.plot(timesteps, Vyvals, '--m')
plt.legend(['Analytic', 'Numerical'])
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex2b/Vcomp.pdf')
plt.show()
plt.close()

plt.figure()
plt.title('Error of computational vs. analytic solution')
plt.xlabel('Time')
plt.ylabel('Velocity error')
plt.ticklabel_format(style='sci', axis='y', scilimits=(0, 0))
plt.plot(timesteps, abs(Vyvals2-Vyvals), '-b')
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex2b/Ecomp.pdf')
plt.show()
```

7.3 Velocity Evaluation 2

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

# spherical grain of dust
rho = 2*(10**5)
D = 10**-4
Vol = (4/3) * np.pi * (D/2)**3
m = rho * Vol

F = - m * 9.81
print(F)

A = 0.25*D**2
B = 1.6*(10**-4)*D
C = F

rho_vals = np.logspace(1, 5, 120)


def time(H, p):
    timesteps = np.arange(0, 4, 0.00001)
    VyR = 10
    t = 0
    dt = 0.0001
    YR = 5
    for i in range(len(timesteps)-1):
        if YR < 0:
            break
        YR = YR + VyR * dt
        deltaVyR = -9.81 * dt - (B/(p*Vol)) * VyR * dt
        VyR = VyR + deltaVyR
        t = t + dt
    return t


fall_times = []
for p in rho_vals:
    tfall = time(5, p)
    fall_times.append(tfall)

plt.figure()
plt.xlabel('Mass (kg)')
plt.ylabel('Fall time (s)')
```

```
plt.title('Mass vs Time of fall')
plt.plot(rho_vals*Vol, fall_times)
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/rhovst.pdf')
plt.show()
```

7.4 Trajectory 1

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

#spherical grain of dust
rho = 2*(10**5)
D = 10**-4
Vol = (4/3) * np.pi * (D/2)**3
m = rho * Vol

F = - m * 9.81
print(F)

A = 0.25*D**2
B = 1.6*(10**-4)*D
C = F


timesteps = np.arange(0, 4, 0.0001)

VyRvals = [10]
VxRvals = [4]
VyR = 10
VxR = 4
t = 0
dt = 0.0001
XR = 0
YR = 0
XRvals = [0]
YRvals = [0]
for i in range(len(timesteps)-1):
    if YR < 0:
        break
    XR = XR + VxR * dt
    YR = YR + VyR * dt
    deltaVyR = -9.81 * dt - (B/m) * VyR * dt
    deltaVxR = -(B/m) * VxR * dt
    VyR = VyR + deltaVyR
```

```
    VxR = VxR + deltaVxR
    t = t + dt
    VyRvals.append(VyR)
    VxRvals.append(VxR)
    XRvals.append(XR)
    YRvals.append(YR)

Vyvals = [10]
Vxvals = [4]
Vy = 10
Vx = 4
t = 0
dt = 0.0001
X = 0
Y = 0
Xvals = [0]
Yvals = [0]
for i in range(len(timesteps)-1):
    if Y < 0:
        break
    X = X + Vx * dt
    Y = Y + Vy * dt
    deltaVy = -9.81 * dt
    Vy = Vy + deltaVy
    t = t + dt
    Vyvals.append(Vy)
    Xvals.append(X)
    Yvals.append(Y)

plt.figure()
plt.title('Trajectory of spherical particle')
plt.xlabel('X Direction (m)')
plt.ylabel('Y Direction (m)')
plt.plot(Xvals, Yvals)
plt.plot(XRvals, YRvals)
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex3/traj1.pdf')
plt.show()
```

7.5 Trajectory 2

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

# spherical grain of dust
```

```
Vol = (4/3) * np.pi * ((10**-4)/2)**3


def xmax(rho, D, v, dt, theta):
    Vol = (4/3) * np.pi * (D/2)**3
    m = rho * Vol
    B = 1.6*(10**-4)*D
    timesteps = np.arange(0, 4, dt)

    VyR = v*np.sin(theta)
    VxR = v*np.cos(theta)
    t = 0
    XR = 0
    YR = 0

    for i in range(len(timesteps)-1):
        if YR < 0:
            return XR
            break
        XR = XR + VxR * dt
        YR = YR + VyR * dt
        deltaVyR = -9.81 * dt - (B/m) * VyR * dt
        deltaVxR = -(B/m) * VxR * dt
        VyR = VyR + deltaVyR
        VxR = VxR + deltaVxR
        t = t + dt


timesteps = np.arange(0, 4, 0.0001)
V = 15
theta = np.arange(0.001, np.pi/2, 0.001)


def opt_theta(rho, D, v, dt):
    theta = np.arange(0.01, np.pi/2, 0.001)
    FXvals = []
    for the in theta:
        if the == 0.001:
            X = xmax(rho, D, v, dt, the)
            FXvals.append(X)
        if the != 0.001:
            X = xmax(rho, D, v, dt, the)
            FXvals.append(X)
            if X != max(FXvals):
                return the
```

```
                break


rho_vals = np.logspace(0.0, 8.0, num=20, endpoint=True)
opt_t = []
for p in rho_vals:
    opt_t.append(opt_theta(p, 10**-4, 10, 0.00001))

print(opt_t)


plt.figure()
plt.title('Optimum angle vs Mass of a sphere')
plt.xlabel('Mass (kg)')
plt.ylabel('Optimum angle (rads)')
plt.xscale('log')
plt.plot(rho_vals*Vol, opt_t, '.b')
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex3/opt_t.pdf')
plt.show()
```

7.6 Quadratic Approximation

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rcParams['mathtext.fontset'] = 'cm'
mpl.rcParams['mathtext.rm'] = 'serif'

# spherical grain of dust
rho = 0.5*(10**4)
D = 10**-4
Vol = (4/3) * np.pi * (D/2)**3
m = rho * Vol


A = 0.25*D**2
B = 1.6*(10**-4)*D

timesteps = np.arange(0, 10, 0.001)
t = 0
dt = 0.0001
theta = np.pi/4
v = 5

Vy = v * np.sin(theta)
```

```
Vx = v * np.cos(theta)
Vyvals = [Vy]
Vxvals = [Vx]
X = 0
Y = 0
Xvals = [X]
Yvals = [Y]
for i in range(len(timesteps)-1):
    if Y < 0:
        break
    X = X + Vx * dt
    Y = Y + Vy * dt
    deltaVy = -9.81 * dt
    Vy = Vy + deltaVy
    t = t + dt
    Vyvals.append(Vy)
    Xvals.append(X)
    Yvals.append(Y)

t = 0
VyR = v * np.sin(theta)
VxR = v * np.cos(theta)
VyRvals = [VyR]
VxRvals = [VyR]
XR = 0
YR = 0
XRvals = [XR]
YRvals = [YR]
for i in range(len(timesteps)-1):
    if YR < 0:
        break
    XR = XR + VxR * dt
    YR = YR + VyR * dt
    deltaVyR = -9.81 * dt - (B/m) * VyR * dt
    deltaVxR = -(B/m) * VxR * dt
    VyR = VyR + deltaVyR
    VxR = VxR + deltaVxR
    t = t + dt
    VyRvals.append(VyR)
    VxRvals.append(VxR)
    XRvals.append(XR)
    YRvals.append(YR)

t = 0
VyRq = v * np.sin(theta)
VxRq = v * np.cos(theta)
```

```
VyRqvals = [VyRq]
VxRqvals = [VxRq]
XRq = 0
YRq = 0
XRqvals = [XRq]
YRqvals = [YRq]
for i in range(len(timesteps)-1):
    if YRq < 0:
        break
    XRq = XRq + VxRq * dt
    YRq = YRq + VyRq * dt
    Vq = np.sqrt(VxRq ** 2 + VyRq ** 2)
    deltaVyRq = -9.81 * dt - (A/m) * Vq * VyRq * dt
    deltaVxRq = -(A/m) * Vq * VxRq * dt
    VyRq = VyRq + deltaVyRq
    VxRq = VxRq + deltaVxRq
    t = t + dt
    VyRqvals.append(VyRq)
    VxRqvals.append(VxRq)
    XRqvals.append(XRq)
    YRqvals.append(YRq)


plt.figure()
plt.title('Trajectory of spherical particle')
plt.xlabel('X Direction (m)')
plt.ylabel('Y Direction (m)')
plt.plot(Xvals, Yvals, label='Frictionless')
plt.plot(XRvals, YRvals, label='Lin. Approx.')
plt.plot(XRqvals, YRqvals, label='Quad. Approx')
plt.legend(loc='upper left', bbox_to_anchor=(-0.01, 1.01),
           markerscale=2, fancybox=True, framealpha=0.2)
plt.savefig('/home/dj-lawton/Documents/SF Lab Plots/Lab3 Plots/Ex3/ex4tr1.pdf')
plt.show()
```