Theory

Case(1)

The problem is a classic projectile motion problem with air assistance. When a projectile is affected by air resistance, the equations of motion can be derived from newton's second law and the drag force equation:

$$F_d = -bv = ma \qquad (1)$$

While,

$$v_x = \frac{dx}{dt}, v_y = \frac{dy}{dt}, a_x = \frac{d^2x}{dt^2}, a_y = \frac{d^2y}{dt^2} \qquad (2)$$

Combining (1) and (2) for x direction, we obtain.

$$\boldsymbol{m\frac{d^2x}{dt^2} = -b\frac{dx}{dt} \Rightarrow \frac{d^2x}{dt^2} = \frac{-b}{m}\frac{dx}{dt}} \qquad \textbf{(3)}$$

for y direction, we have to tale gravitational force (mg) into account, by combining (1) and (2) and mg, we obtain.

$$\boldsymbol{m\frac{d^2y}{dt^2} = -b\frac{dy}{dt} - mg \Rightarrow \frac{d^2y}{dt^2} = -g - \frac{b}{m}\frac{dy}{dt}} \qquad (\mathbf{4})$$

We have successfully, obtain the equations for case I by using appropriate formula and laws.

Case(II)

Two problems are highly similar but the drag for equations are different.

$$F_d = -cv^2 = ma \qquad (5)$$

We now consider the drag force as a vector and we have,

$$\overrightarrow{F_d} = -cv^2\hat{v} \qquad (6)$$

And the velocity vector is.

$$\vec{v} = v_x\hat{\imath} + v_y\hat{\jmath} \qquad (7)$$

The magnitude of the velocity is

$$v = \sqrt{v_x^2 + v_y^2} \qquad (8)$$

By using (7) and (8), we can find the unit vector in the direction of velocity,

$$\hat{v} = \frac{v_x}{v}\hat{\imath} + \frac{v_y}{v}\hat{\jmath} \qquad (9)$$

We can now express the drag force in x-direction and y-direction,

$$F_{dx} = -cv^2 \frac{v_x}{v} = -cvv_x \qquad (10)$$

$$F_{dy} = -cv^2 \frac{v_y}{v} = -cvv_y \qquad (11)$$

Consider the x-direction and y-direction,

$$F_{dx} = -cvv_x = ma_x \qquad (12)$$

$$F_{dy} = -cvv_y - mg = ma_y \qquad (13)$$

With (2), (8), (12), we derive,

$$-c\sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2}\frac{dx}{dt} = m\frac{d^2x}{dt^2} \qquad (14)$$

Rearrange

$$\frac{d^2x}{dt^2} = \frac{-c}{m}\frac{dx}{dt}\sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2} \qquad (15)$$

Similarly for y direction, with (2), (8), and (13), we derive,

$$\frac{d^2y}{dt^2} = -g - \frac{c}{m}\frac{dy}{dt}\sqrt{(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2} \qquad (16)$$

We have successfully derived the formula of case 2.

Case(III)

$$p = p_0\left(1 - \frac{Lh}{h_0}\right)^{\frac{gM}{RL}} \qquad (17)$$

The barometric formula in case three is originated from this formula with taking approximation of the constants of the power term into 4.256. It can be derived from ideal gas law.

Ideal gas law:

$$p = \frac{\rho}{M}RT \qquad (18)$$

Assuming that all the pressure is hydrostatic:

$$dP = -\rho g\, dh$$

Divide both side of the equation by P, we have:

$$\frac{dP}{P} = -\frac{Mgdh}{RT}$$

Integrating this expression from the surface to altitude h we get:

$$P = P_0 e^{\left(-\int_0^h Mgdh/RT\right)}$$

Assuming linear temperature change $T = T_0 - Lh$ and constant molar mass and gravitational acceleration. We obtain.

$$P = P_0 . \left[\frac{T}{T_0}\right]^{\frac{Mg}{RL}}$$

Substitute $T = T_0 - Lh$ we have,

$$p = p_0 \left(1 - \frac{Lh}{h_0}\right)^{\frac{gM}{RL}}$$

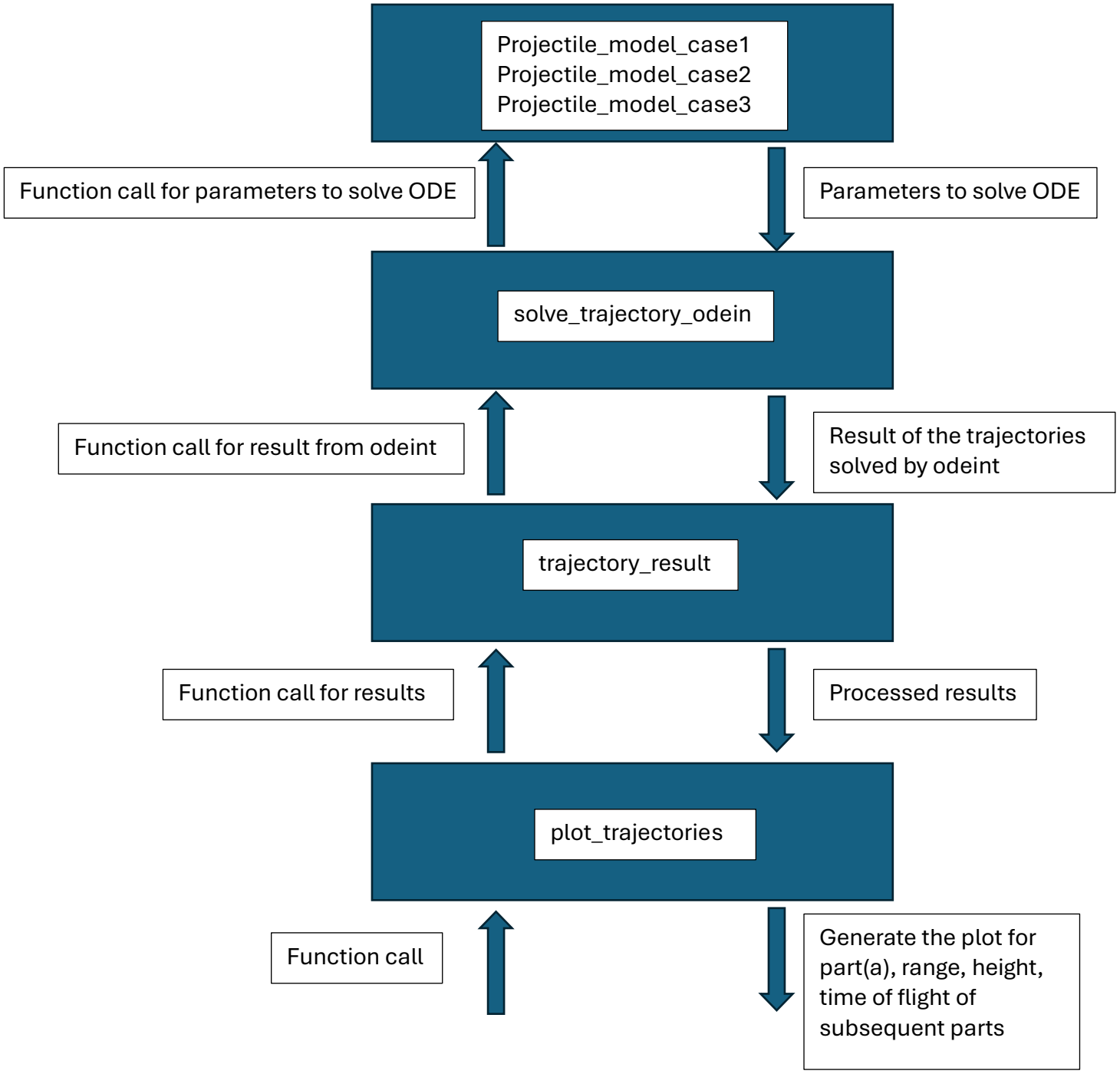
by substituting the appropriate values, we can obtain the quadratic drag parameter c formula stated in case III.

Algorithm

Part(a)

1. Set up constants g, b/m and c0/m for calculations.
2. Set up three functions to return dxdt, dtdt, dvxdt and dvydt for 3 cases respectively.
    a. For function of case 3, if-else condition is added to handle the case to calculate the changing air density due to change in altitude h.
3. Write a function called solve_trajectory_odeint to return the trajectory of with x and y coordinates, time and time of flight with odeint function.
    a. Convert the angles into radians for calculation.
    b. Initialize the conditions for the odeint.
    c. Extract the useful part of the solution returned by odeint as the coordinates of x and y.
    d. Use valid indices to store only the trajectory until it hits the ground.
    e. Time of flight is the last index valid_indices of y of the time grid
    f. Return the x, y, t, within the valid_indices, as well as the time of flight. Time of flight is for part (d) but we return here since part (d) request the data from part (a).
4. Write a function trajectory_results to process the result from solve_trajectory_odeint. Store the range, height and time_of_flight_array for the trajectories for part (b) ,part (c) and part (d) since both parts require the information from part (a).
5. Write a function to plot the trajectories. In here, the function in 4. and 3. are called, then the results will be plotted. Nested for loop is used since results of multiple angles should be on the same graph for different speed. The function will return range, height, and time of flight for other parts.
6. Set up array for angles for calculation.
7. Set up 6 different speeds for different cases, for case 1, we need to round the number to decimals=7 due to number representation problem.
8. Call the function plot_trajectories to show the result, meanwhile, store the range array, height array and time of flight array for part (b) and part (c) and part (d).

The relationship of the functions in part (a) is illustrated in the following diagram.



Part(b)(c)(d)

1. **Write a function call plot_data to show the result, the functions take arguments of an array, a array of speed, y-label, title, and angle arrays. Since part (b), (c) and (d) all have the same x variable, it only take the y-label as the input.**
2. **Split the range, height and time of flight arrays from part (a) into 6 parts corresponding to 6 different speeds.**
3. **Use tuples to assign all the cases for plotting.**
4. **Use a for loop to iterate all the cases and print the result on the same graph.**

Coding

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint

############################## part (a) ##############################

# Constants
g = 9.807  # gravitational acceleration (m/s^2)
b_over_m = 1.28e5  # drag coefficient per unit mass (1/s)\
c0_over_m = 1.46e-4  # drag coefficient per unit mass (1/s)

def projectile_model_case1(z, t):
    x, y, vx, vy = z
    v = np.sqrt(vx**2 + vy**2)
    dxdt = vx
    dydt = vy
    dvxdt = -b_over_m * vx
    dvydt = -g - b_over_m * vy
    return [dxdt, dydt, dvxdt, dvydt]

def projectile_model_case2(z, t):
    x, y, vx, vy = z
    v = np.sqrt(vx**2 + vy**2)  # Calculate the magnitude of velocity
    dxdt = vx
    dydt = vy
    dvxdt = -c0_over_m * vx * v
    dvydt = -c0_over_m * vy * v - g
    return [dxdt, dydt, dvxdt, dvydt]

def projectile_model_case3(z, t):
    x, y, vx, vy = z
    v = np.sqrt(vx**2 + vy**2)  # Calculate the magnitude of velocity

    # Constants
    c0_over_m = 1.46e-4  # drag coefficient per unit mass (1/s)
    T_0 = 288.2  # Reference temperature (K)
    L = 0.0065  # Temperature lapse rate (K/m)

    # Calculate altitude-dependent drag coefficient
    if y > 0:
```

```
        c_over_m = c0_over_m * (1 - (L * y) / T_0) ** 4.256
    else:
        c_over_m = c0_over_m

    dxdt = vx
    dydt = vy
    dvxdt = -c_over_m * vx * v
    dvydt = -c_over_m * vy * v - 9.807  # gravitational acceleration (m/s^2)

    return [dxdt, dydt, dvxdt, dvydt]

# this is the function that will be used to solve the ODEs
def solve_trajectory_odeint(v0, theta_deg, model):
    theta = np.radians(theta_deg)
    initial_conditions = [0, 0, v0 * np.cos(theta), v0 * np.sin(theta)]
    t = np.linspace(0, v0 * 5, 99999)
    sol = odeint(model, initial_conditions, t)
    x, y = sol[:, 0], sol[:, 1]
    valid_indices = y >= 0
    time_of_flight = t[valid_indices][-1]
    return x[valid_indices], y[valid_indices], t[valid_indices], time_of_flight

# this function will be used to get the results of the trajectory, also for part (b), (c) and (d)
def trajectory_results(v0s, angles, model):
    results = {}
    range_arr = []
    height_arr = []
    time_of_flight_arr = []

    for v0 in v0s:
        for theta in angles:
            key = (v0, theta)
            x, y, t, tf = solve_trajectory_odeint(v0, theta, model)
            range_val = max(x)
            height_val = max(y)
            results[key] = {'x': x, 'y': y, 't': t, 'tf': tf, 'range': range_val, 'height': height_val}
            range_arr.append(range_val)
            height_arr.append(height_val)
            time_of_flight_arr.append(tf)

    return results, range_arr, height_arr, time_of_flight_arr
```

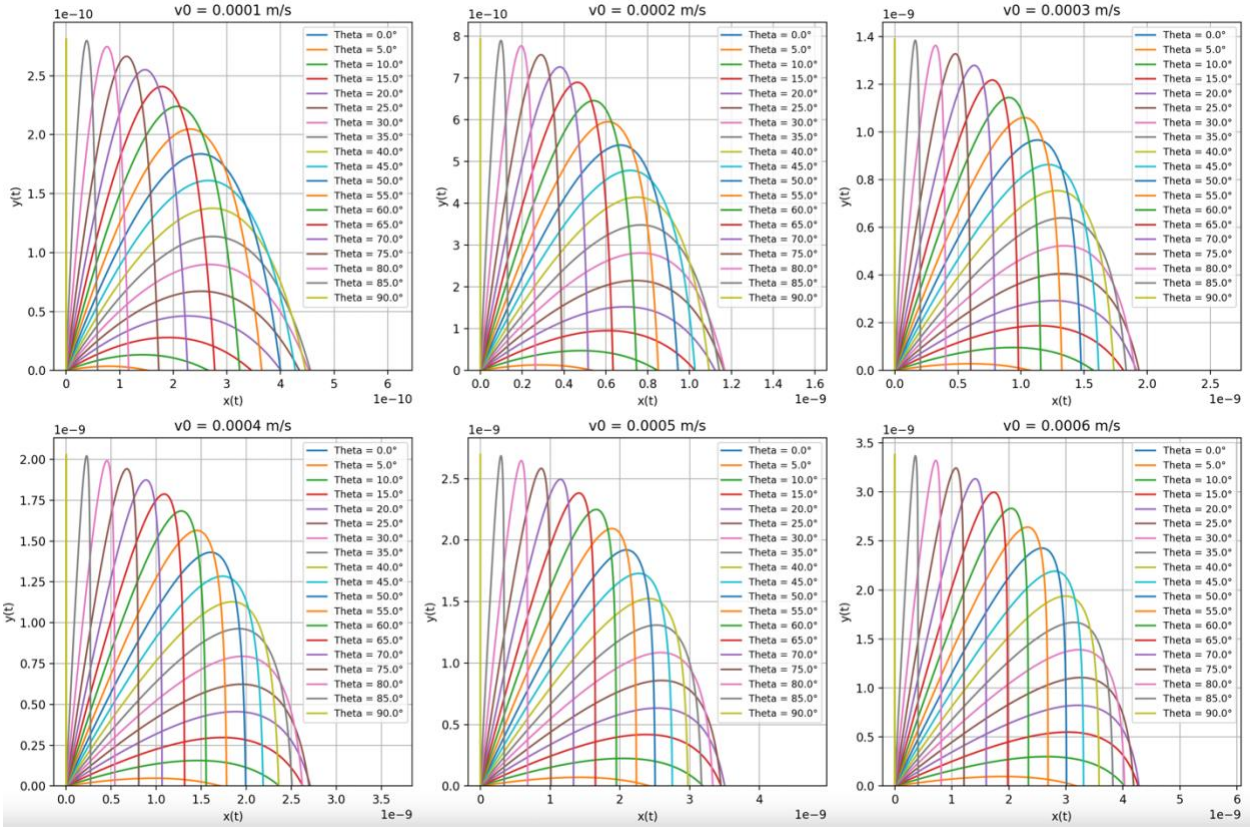
```python
# this function will be used to plot the trajectories
def plot_trajectories(v0s, angles, model):
    results, range_arr, height_arr, time_of_flight_arr = trajectory_results(v0s, angles, model)
    fig, axs = plt.subplots(2, 3, figsize=(15, 10))
    axs = axs.flatten()  # Flatten the array of axes for easier indexing

    idx = 0
    for i, v0 in enumerate(v0s):
        for theta in angles:
            key = (v0, theta)
            x = results[key]['x']
            y = results[key]['y']
            axs[i].plot(x, y, label=f'Theta = {theta}°')

        axs[i].set_title(f'v0 = {v0} m/s')
        axs[i].set_xlabel('x(t)')
        axs[i].set_ylabel('y(t)')
        axs[i].legend(fontsize='small', loc= 'best')
        axs[i].grid(True)
        axs[i].set_ylim(bottom=0)
        # Automatically extend x-axis limits with margin for better visualization
        current_x_limit = axs[i].get_xlim()
        axs[i].set_xlim(current_x_limit[0], current_x_limit[1] * 1.35)
    plt.tight_layout()
    plt.show()

    return range_arr, height_arr, time_of_flight_arr

# define the angles
angles = np.linspace(0, 90, 19)
num_of_speeds = 6

# plot the trajectories for case 1
v0s_case1 = np.linspace(10e-5, 6*10e-5, num_of_speeds)  # 6 different speeds
v0s_case1 = np.around(v0s_case1, decimals=7)
range_arr_case1, height_arr_case1, time_of_flight_arr_case1 = plot_trajectories(v0s_case1, angles, projectile_model_case1)

# plot the trajectories for case 2
v0s_case2 = np.linspace(500, 3*500, num_of_speeds)  # 6 different speeds
range_arr_case2, height_arr_case2, time_of_flight_arr_case2 = plot_trajectories(v0s_case2, angles, projectile_model_case2)
```

```
# plot the trajectories for case 3
v0s_case3 = np.linspace(500, 3*500, num_of_speeds)  # 6 different speeds
range_arr_case3 , height_arr_case3, time_of_flight_arr_case3 = plot_trajectories(v0s_case3, angles, projectile_model_case3)

############################## part (b)(c)(d) ##############################

# this is a function that will be used to plot the data for part (b) (c) and (d)
def plot_data(arrays, v0s, ylabel, title, angles):
    plt.figure(figsize=(10, 6))  # Set the size of the plot
    for i, array in enumerate(arrays):
        plt.plot(angles, array, label=f'{ylabel} for v0 = {v0s[i]} m/s')
    plt.title(title)
    plt.xlabel('Theta (degrees)')
    plt.ylabel(f'{ylabel}')
    plt.legend()
    plt.grid(True)
    plt.show()

#split the range arrays obtained from part (a) into 6 parts for each speed
range_arr_case1 = np.array_split(range_arr_case1, num_of_speeds)
range_arr_case2 = np.array_split(range_arr_case2, num_of_speeds)
range_arr_case3 = np.array_split(range_arr_case3, num_of_speeds)
height_arr_case1 = np.array_split(height_arr_case1, num_of_speeds)
height_arr_case2 = np.array_split(height_arr_case2, num_of_speeds)
height_arr_case3 = np.array_split(height_arr_case3, num_of_speeds)
time_of_flight_arr_case1 = np.array_split(time_of_flight_arr_case1, num_of_speeds)
time_of_flight_arr_case2 = np.array_split(time_of_flight_arr_case2, num_of_speeds)
time_of_flight_arr_case3 = np.array_split(time_of_flight_arr_case3, num_of_speeds)

#assign cases into the tuples and use for loop to plot them
cases = [
(range_arr_case1, v0s_case1, 'Range (m)', 'Case 1', angles),
(range_arr_case2, v0s_case2, 'Range (m)', 'Case 2', angles),
(range_arr_case3, v0s_case3, 'Range (m)', 'Case 3', angles),
(height_arr_case1, v0s_case1, 'Height (m)', 'Case 1', angles),
(height_arr_case2, v0s_case2, 'Height (m)', 'Case 2', angles),
(height_arr_case3, v0s_case3, 'Height (m)', 'Case 3', angles),
(time_of_flight_arr_case1, v0s_case1, 'Time of Flight (s)', 'Case 1', angles),
(time_of_flight_arr_case2, v0s_case2, 'Time of Flight (s)', 'Case 2', angles),
(time_of_flight_arr_case3, v0s_case3, 'Time of Flight (s)', 'Case 3', angles)]
```

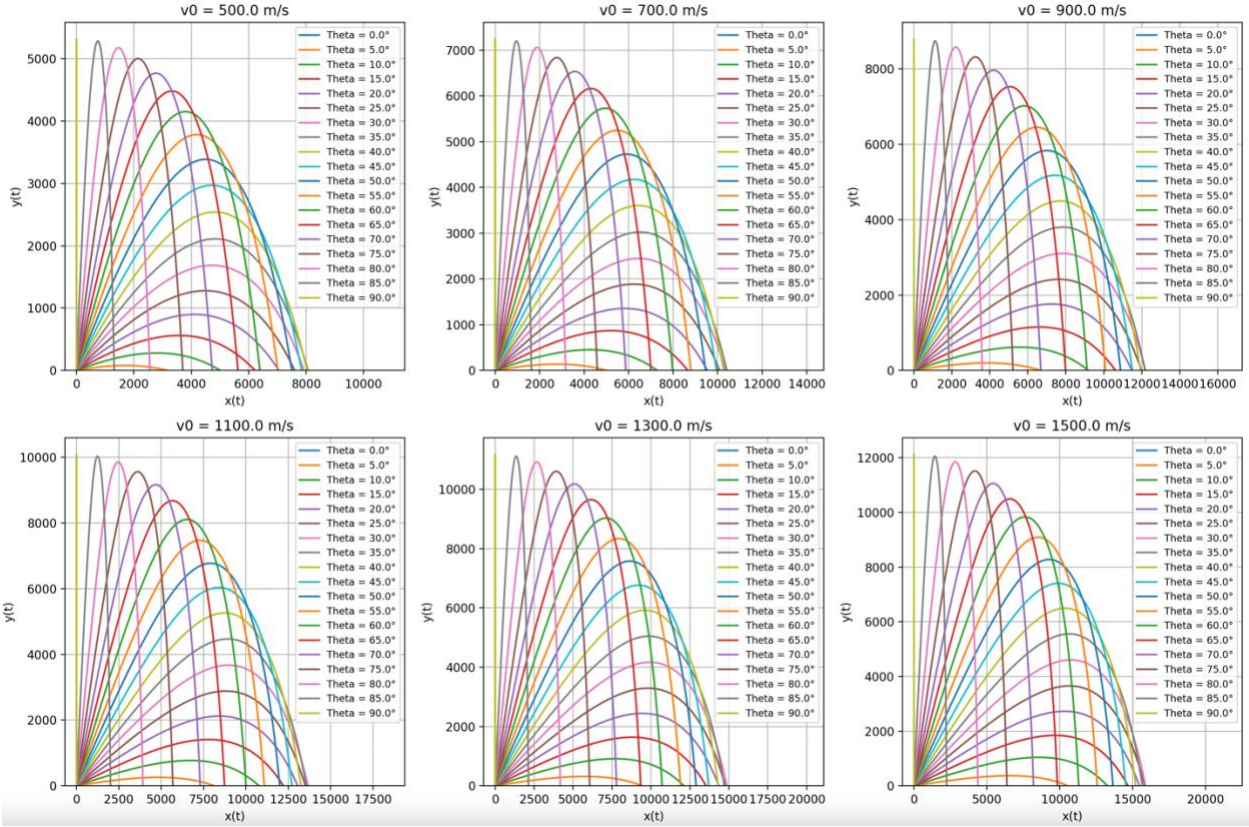
```
for data, v0s, ylabel, title_prefix, angles in cases:
    plot_data(data, v0s, ylabel, title_prefix, angles)
```

Results

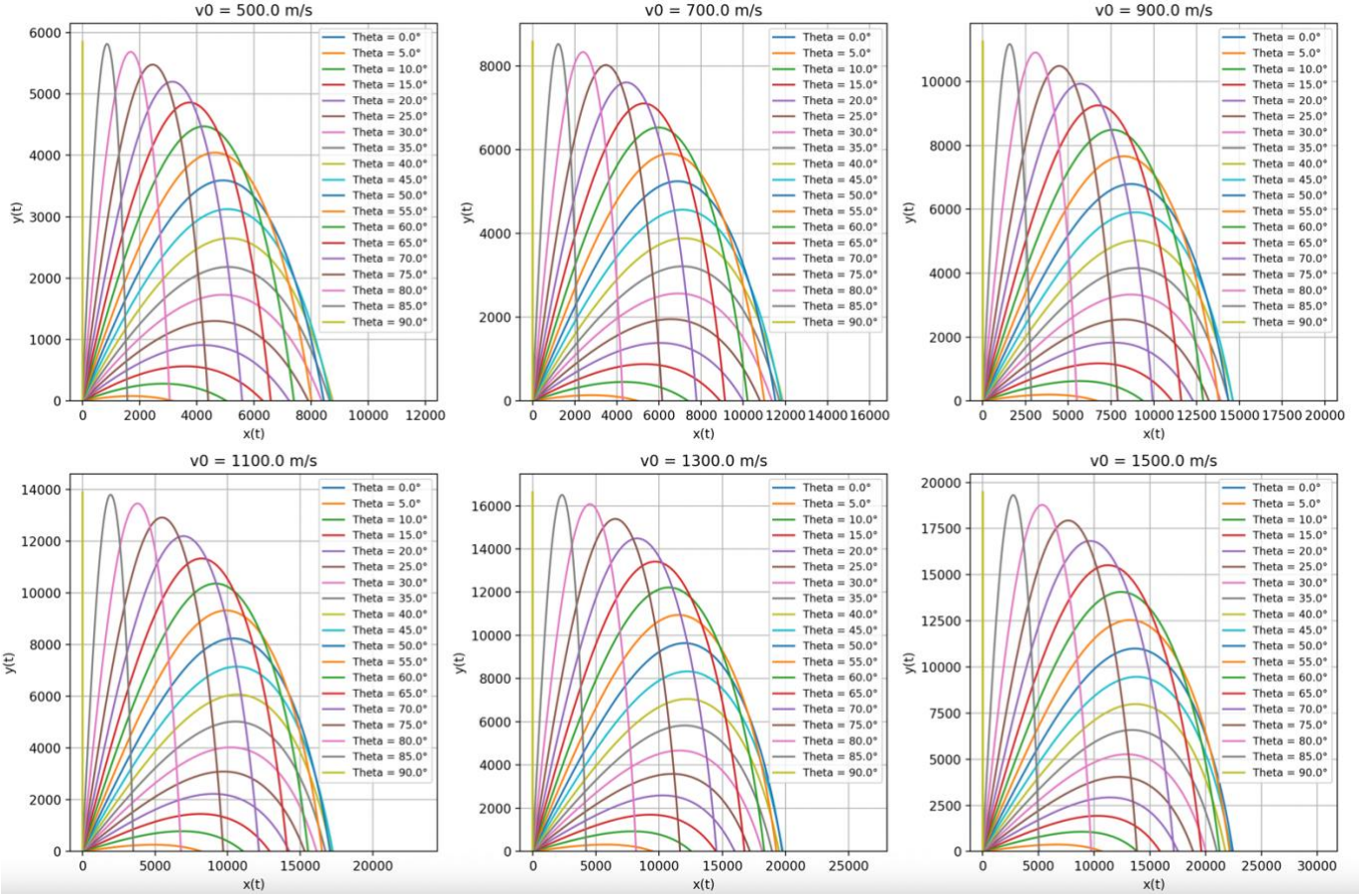
Trajectories of case 1 in 6 different initial speed and 18 angles



Trajectories of case 2 in 6 different initial speed and 18 angles



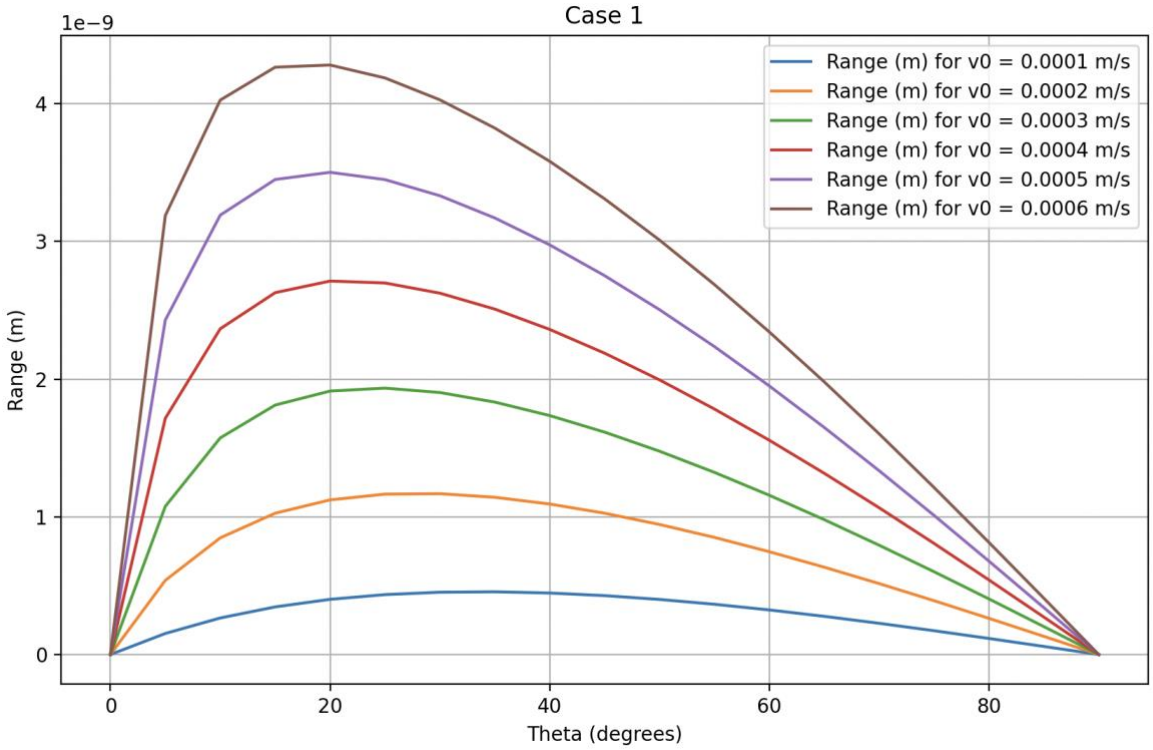
Trajectories of case 3 in 6 different initial speed and 18 angles



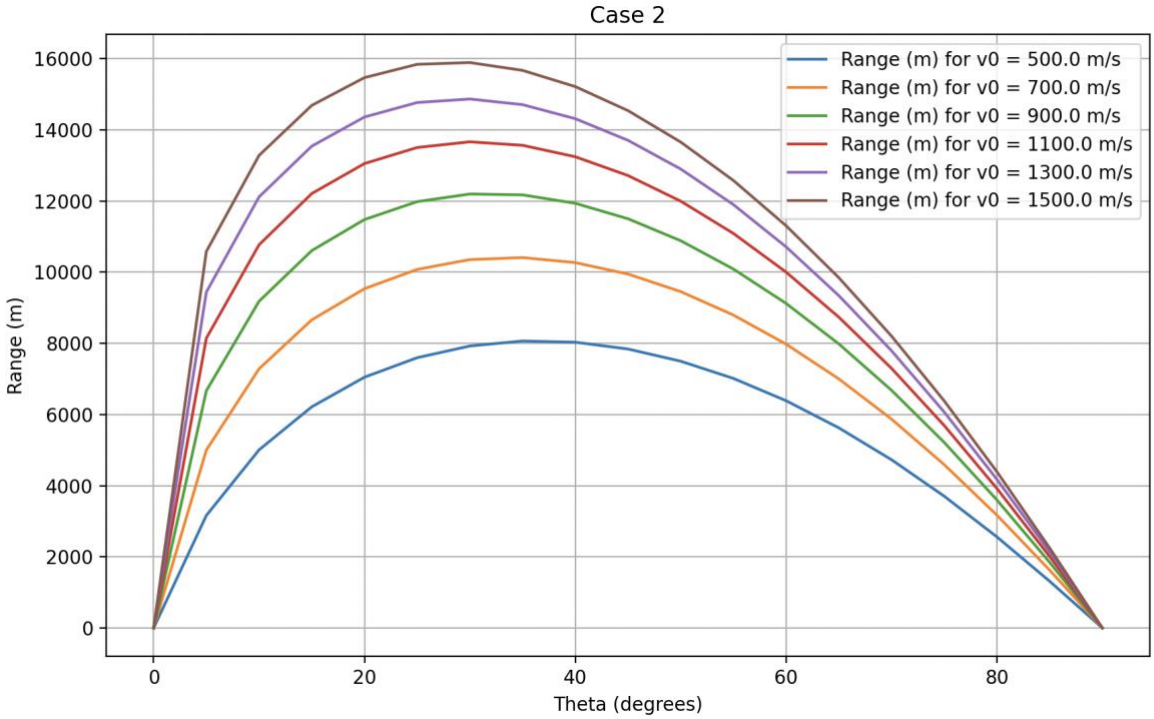
Comments on results:

All the trajectories are projectile shape. Generally, a larger speed will result in larger max height and range. However, it is obvious that the range max range are not achieved with 45 degrees. This will be further discussed in the next result.

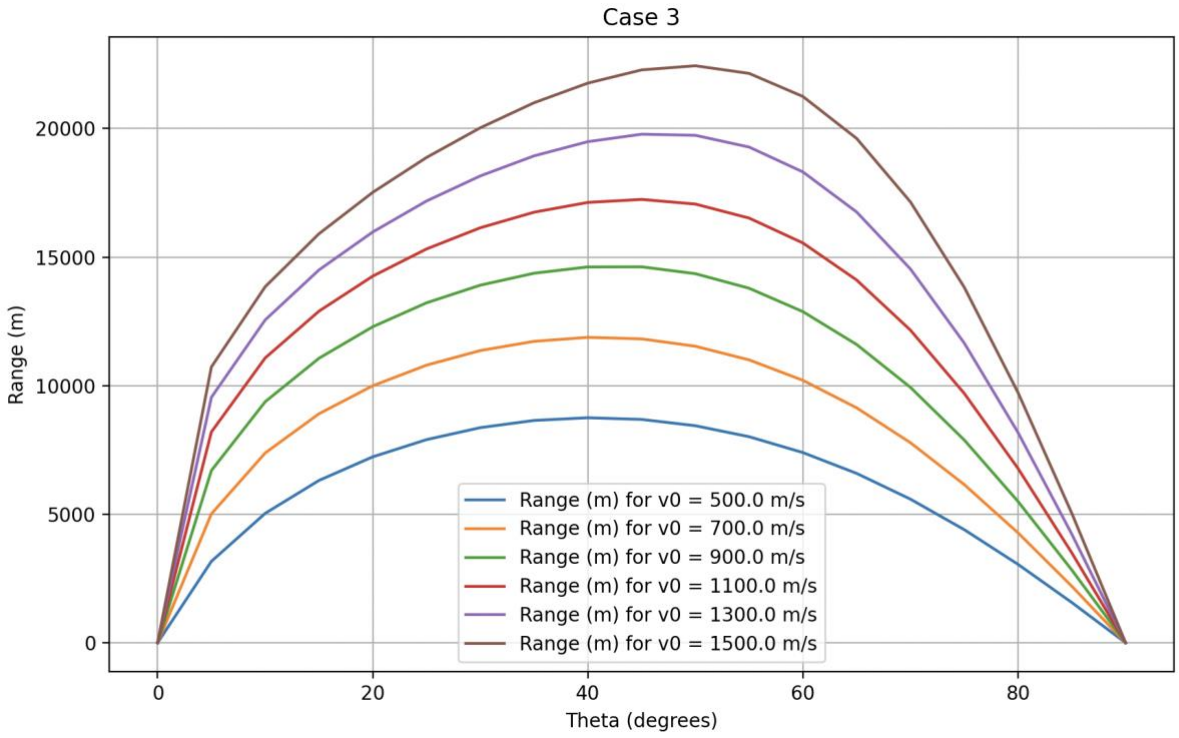
Ranges versus different launching angles in case 1 in 6 different initial speeds



Ranges versus different launching angles in case 2 in 6 different initial speeds



Ranges versus different launching angles in case 3 in 6 different initial speeds



Comments on the result:

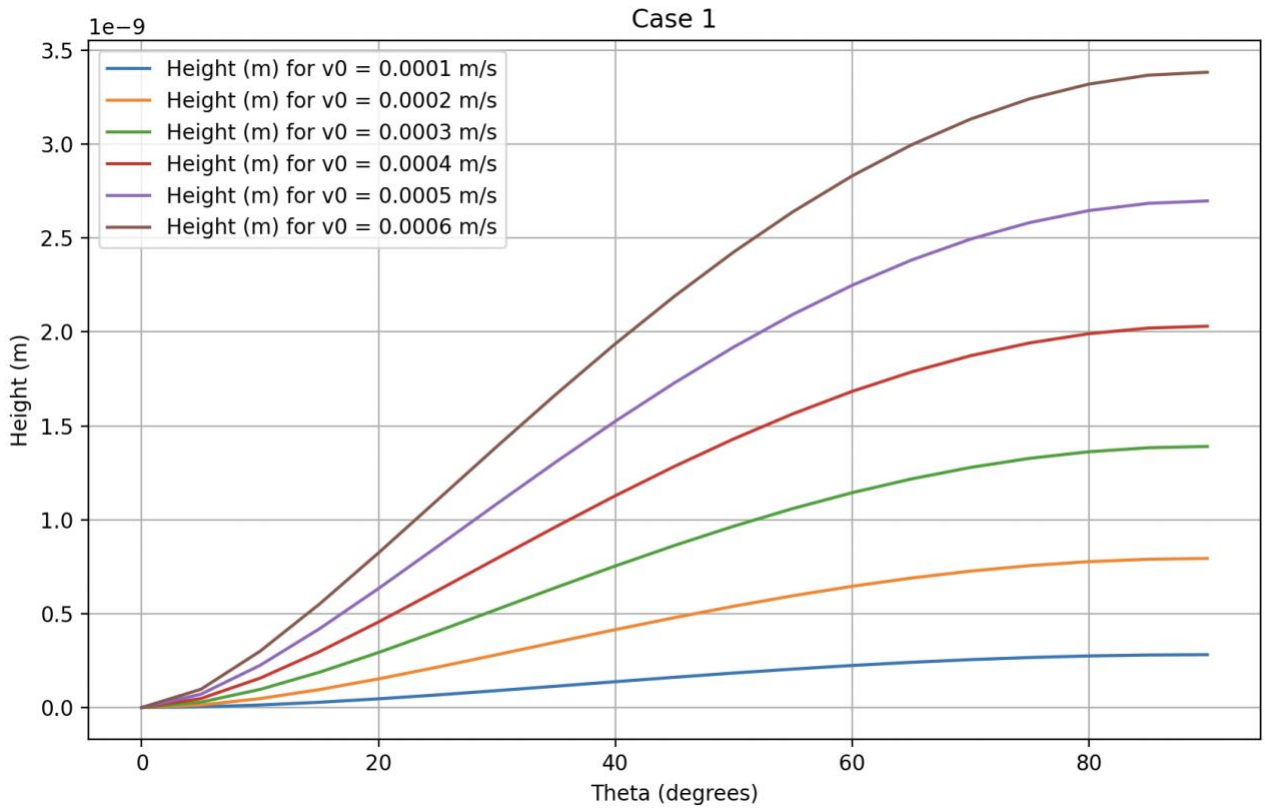
Without drag force, the relationship of the Theta and the range should be symmetric parabola. Since it is governed by the formula $R = \frac{(v_0)^2 sin(2\theta)}{g}$. When $\theta = 45^o$, $sin(2\theta) = 1$, we have the maximum range R of the trajectories.

However, due the effect of drag force and since drag force is related to the velocity as stated in the equations (1) and (5) of the theory part, the shape of the trajectories is not symmetric and will depends on the velocities.

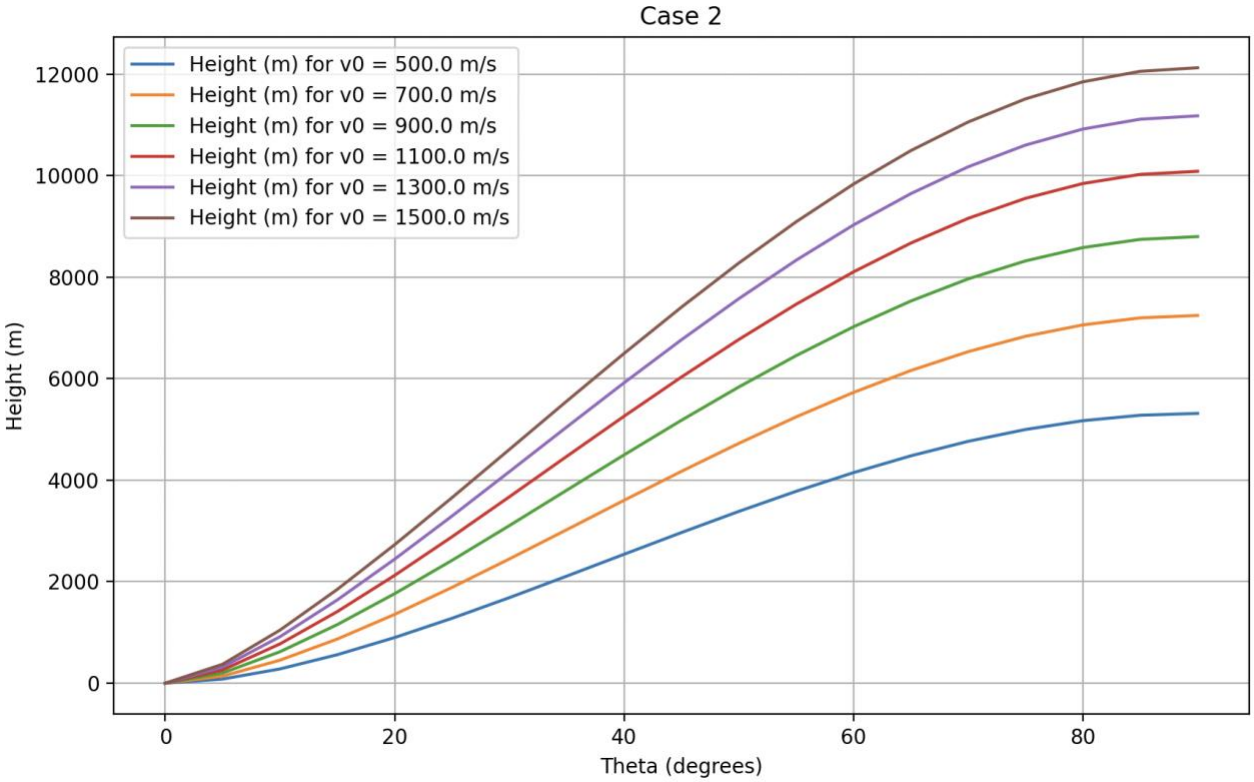
In general, the curve shows a more symmetric shape with lower speed. With a higher speed, since the effecting of drag force becomes more significant as drag force is either related to v or v^2 , it deviates more from the symmetric shape.

For case 1, generally, a longer range can be achieved with smaller angle. For case 2 we need a larger one but still smaller than 45 degrees. For case 3, we need a larger launching angle with more than 45 degrees to achieve longer range. For case 1 and case 2, since lower angles will give a larger v_x but smaller v_y, and since v_y govern the time the object spend in the air. Spending less time in the air will but greater speed will therefore increase the range of the object as less time for drag force to act on the object. For case 3, since the air density at higher altitude is lower and hence reducing the drag force. Larger range can be achieved with slightly larger angles as it will spend more time in higher altitude.

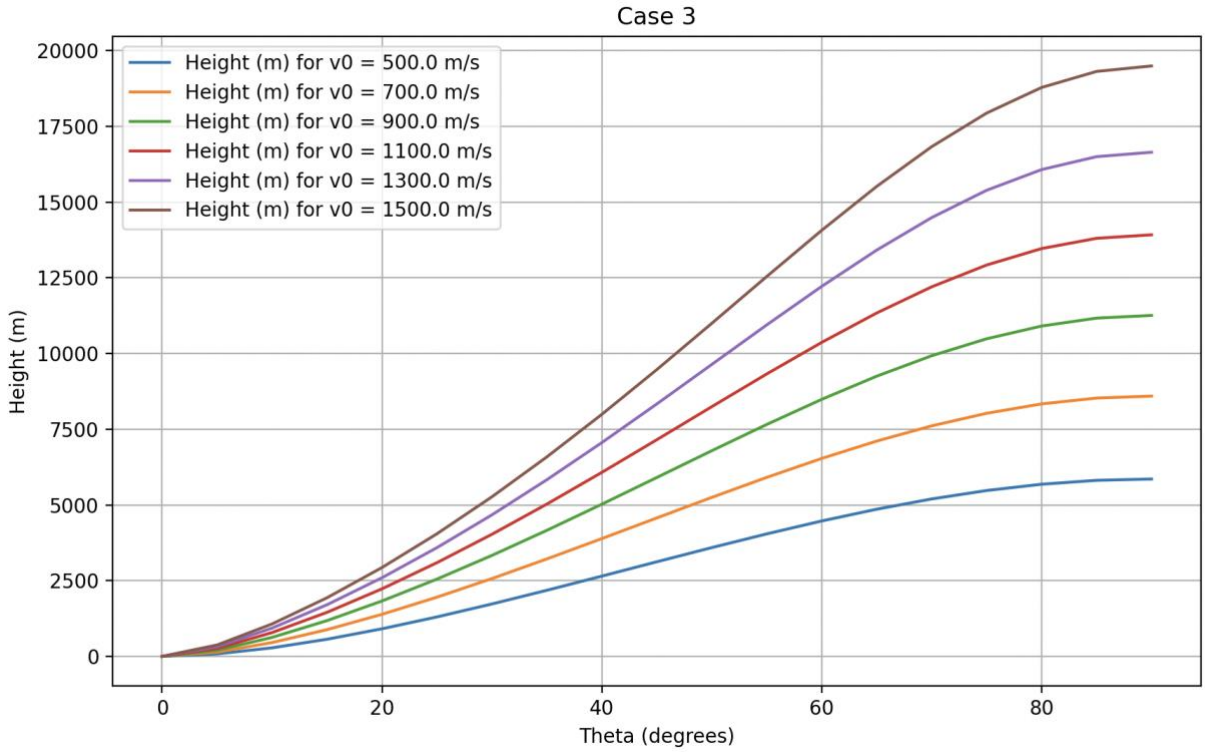
Max height versus different launching angles in case 1 in 6 different initial speeds



Max height versus different launching angles in case 2 in 6 different initial speeds



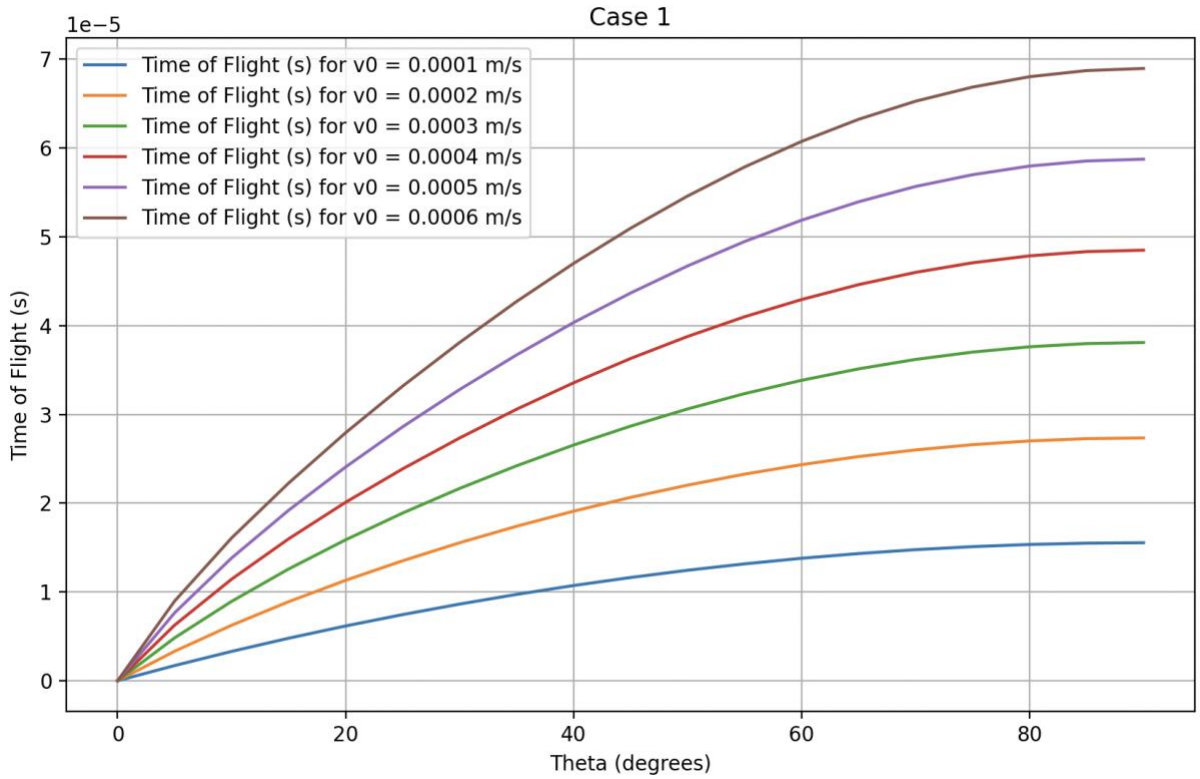
Max height versus different launching angles in case 3 in 6 different initial speeds



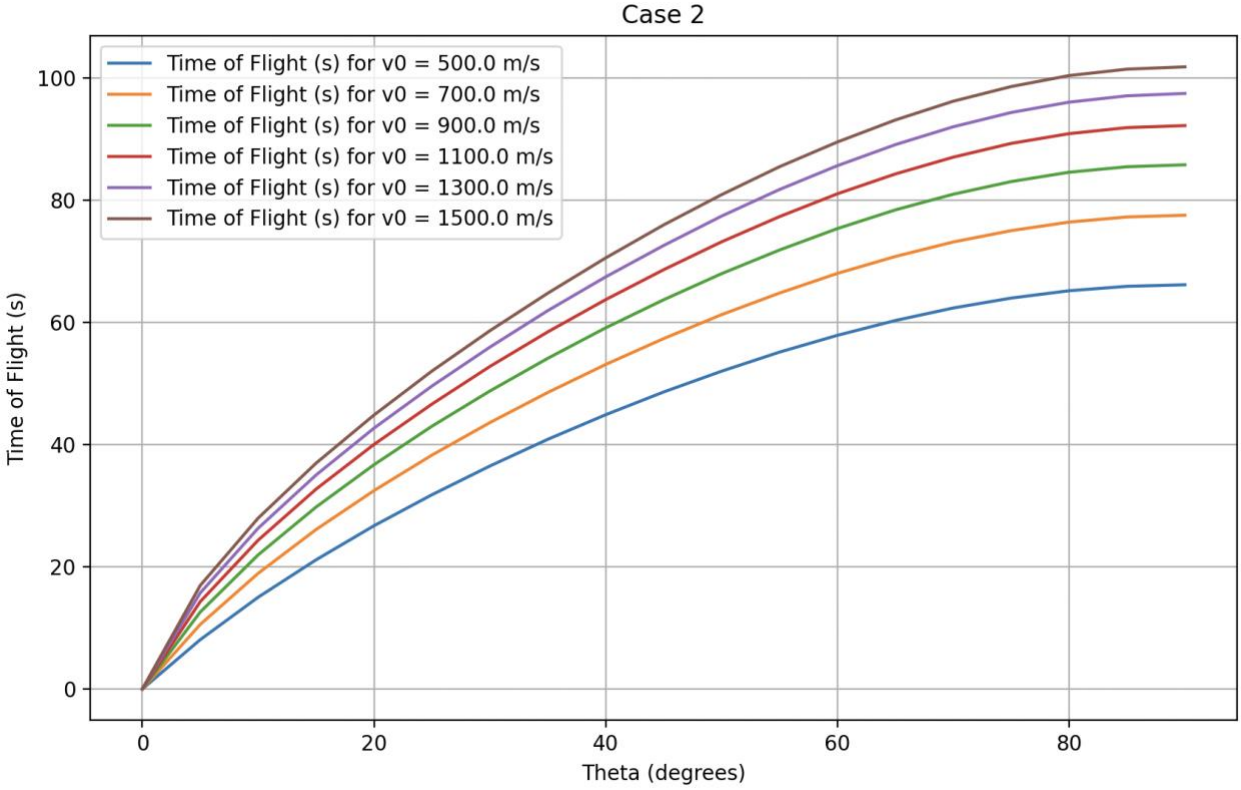
Comments on the result:

We can observe that the maximum height of the object increase monotonically with the launching angles in different speeds. $h_{max} = \frac{v_y^2}{2g}$ is the equation governing the where $v_y = vsin(\theta)$. Therefore, the shape of the ideal projectile motion should be like the shape pf the graph $y = ksin^2\theta$ where $k = \frac{v^2}{2g}$. As we can see from the result, due to the presence of drag force, the graph is fattened comparatively. The v_y component is reduced by drag force and therefore reduce the maximum height of the object. The reduction is more significant for case 2 and 3 since the drag force is increase faster as indicated in equation (5) in theory. But the effect of case 3 is less than case 2 since the drag force is weaker when the air density is lower in the higher altitude.

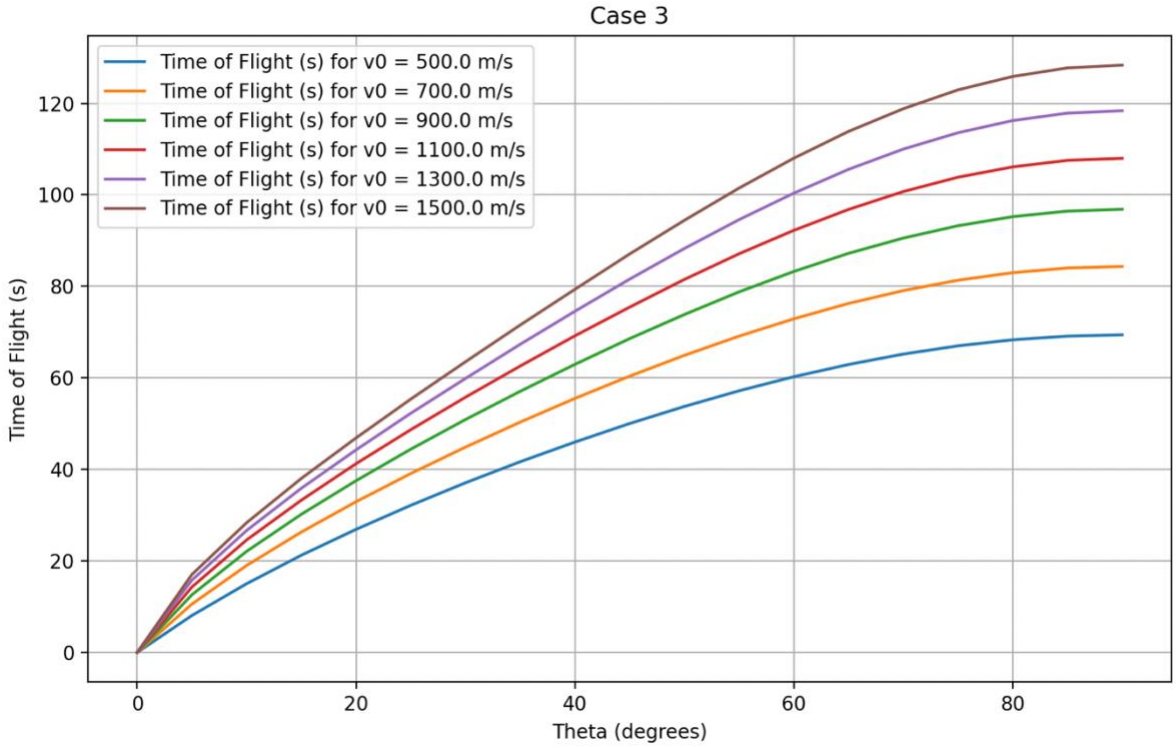
Time of flight versus different launching angles in case 1 in 6 different initial speeds



Time of flight versus different launching angles in case 2 in 6 different initial speeds



Time of flight versus different launching angles in case 3 in 6 different initial speeds



Comments on the result:

We can observe the time of flight increase monotonically with the launching angles in different speeds. The equations governing the time of flight of projectile motion without drag force is.

$$t = \frac{2v_0 sin\theta}{g}$$

The shape of the curve is therefore a sinusoidal shape. Due to the presence of drag force, all three results are flattened sinusoidal shape compared to the ideal case. Again, case 2 and 3 are more obvious since the drag force is increasing faster with 2 more obvious than 3. Since case 3 experience less drag force in higher altitude.

Discussion

1. **During the algorithm design and coding process. There is always constant tradeoff decision of compactness and modularity of the code. Shortening some code might reduce maintainability but will make it easier to read, the current version opted for compactness. However, if less linkage between function or break down the functions into more pieces might possibility enhance maintainability for debugging since the produce more intermediatory results and failures of certain parts of code might affect less other part of the overall system.**
2. **Another trade off was made for number of angles used to show the results. More angles will produce better results for part (b), (c) and (d) but will reduce the visibility of part (a) since putting a lot of trajectories make it hard to read. However, since the result of subsequent parts rely on part(a), therefore, after several trial and error, 18 angles were chosen to balance visibility of part (a) and smoothness of results in part (b), (c) and (d).**
3. **Whether or not to do part (b), (c) and (d) all together at once or separately has its pros and cons. Since part (b), (c) and (d) are higher repetitively, I chose to do it all in the same way with sharing one function called plot_data. However, if the questions amened slightly. A large refactoring of code might require. Therefore, implement part (b), (c) and (d) separately will have an advantage if the questions or output needs to adjust in the future.**
4. **This is a small scale project and therefore, OOP approaches were not utilized. However, If there are more subpart of the questions, OOP will certainly be better to enable modularity, reusability and scalability. Since the code will be less coupled and more reusable.**
5. **There is one little weakness of my program. In the solve_trajectroy_odeint. The time of flight was calculated by returning the last valid_indices of the trajectory, which is retrieved by -1. However, this approach will produce large error if the time points in t = np.linspace(0, v0*5, 99999) was chosen with less points. However, since this project requires a smooth visualization, large time points are needed, therefore, this is not a big problem.**

Reference:

West, M. (n.d.). *Dynamics*. Projectiles with air resistance. https://dynref.engr.illinois.edu/afp.html

Lente, G., Ősz, K. Barometric formulas: various derivations and comparisons to environmentally relevant observations. *ChemTexts* **6**, 13 (2020). https://doi.org/10.1007/s40828-020-0111-6

S. T. Thornton and J. B. Marion, Classical Dynamics (Thomson, 2004)