Contents

1 Problem Setup

A ball is thrown with an initial velocity and angle, considering the following forces:

- Gravity: Acts downward with constant acceleration $g \approx 9.81\,\mathrm{m/s^2}$.
- Air resistance (drag): Acts opposite to the direction of motion, affecting both the horizontal and vertical components of the ball's velocity.

2 Breaking Down the Initial Conditions

The initial conditions consist of:

2.1 Initial velocity components:

- Horizontal component:
$$v_{0x} = v_0 \cos(\theta)$$
- Vertical component:
$$v_{0y} = v_0 \sin(\theta)$$

2.2 Initial position:

- Horizontal position:
$$x_0$$
- Vertical position:
$$y_0$$

3 Forces Acting on the Ball

The forces acting on the ball include:

3.1 Gravity (F_g):

- Acts downward with a constant acceleration g, contributing to vertical motion only.

3.2 Drag force ($F_{\mathbf{drag}}$):

- Acts opposite to the direction of motion and is proportional to the square of the velocity. This force is decomposed into horizontal and vertical components:

$$F_{\text{drag}} = \frac{1}{2} C_d \rho A v^2$$

Where:

1. C_d: Drag coefficient,
2. ρ: Air density,
3. A: Cross-sectional area of the ball ($A = \pi r^2$, with r as the radius of the ball),
4. v: Total velocity of the ball,

$$v = \sqrt{v_x^2 + v_y^2}$$

The drag force components are:

- Horizontal drag:

$$F_{\text{drag},x} = \frac{1}{2} C_d \rho A v v_x$$

- Vertical drag:

$$F_{\text{drag},y} = \frac{1}{2} C_d \rho A v v_y$$

4 Equations of Motion with Air Resistance

The motion of the ball is governed by the following equations of motion:

4.1 Horizontal direction (x):

$$m \frac{dv_x}{dt} = -\frac{1}{2} C_d \rho A v v_x$$

$$\frac{dv_x}{dt} = -\frac{C_d \rho A}{2m} v v_x$$

4.2 Vertical direction (y):

$$m\frac{dv_y}{dt} = -mg - \frac{1}{2}C_d\rho A v v_y$$

$$\frac{dv_y}{dt} = -g - \frac{C_d\rho A}{2m} v v_y$$

Where:

- m is the mass of the ball,
- g is the acceleration due to gravity,
- v_x and v_y are the horizontal and vertical velocity components at any time t,
- v is the total velocity,

$$v = \sqrt{v_x^2 + v_y^2}$$

5 Numerical Integration (Euler's Method)

The differential equations are solved numerically using **Euler's method** for time-stepping, where the velocities and positions are updated iteratively over small time steps Δt.

5.1 Initial Conditions

At $t = 0$:

- $$v_x(0) = v_0 \cos(\theta)$$
- $$v_y(0) = v_0 \sin(\theta)$$
- $$x(0) = x_0$$
- $$y(0) = y_0$$

5.2 Update Rules for Velocity and Position

1. **Velocity update**: For each time step Δt, update the velocity components v_x and v_y as follows:

$$v_x(t + \Delta t) = v_x(t) + \frac{dv_x}{dt} \cdot \Delta t$$

$$v_y(t + \Delta t) = v_y(t) + \frac{dv_y}{dt} \cdot \Delta t$$

Where the derivatives of the velocities are given by:

$$\frac{dv_x}{dt} = -\frac{C_d \rho A}{2m} v v_x$$

$$\frac{dv_y}{dt} = -g - \frac{C_d \rho A}{2m} v v_y$$

1. **Position update**: After updating the velocities, update the positions x and y based on the new velocities:

$$x(t + \Delta t) = x(t) + v_x(t) \cdot \Delta t$$

$$y(t + \Delta t) = y(t) + v_y(t) \cdot \Delta t$$

5.3 Repeat for each time step

The process is repeated for subsequent time steps until balls collide. At each step, the velocity and position are updated using the above formulas.

6 Complete Algorithm

1. **Initialize parameters**:
   - Set initial values for v_0, θ, m, r, C_d, ρ, A, and g.
   - Set initial positions: x_0, y_0.
   - Compute initial velocity components:

$$v_{0x} = v_0 \cos(\theta)$$

,

$$v_{0y} = v_0 \sin(\theta)$$

2. **Define time step** Δt (e.g., 0.01 seconds).
3. **Set initial time** $t = 0$.
4. **At each time step**:
   - Compute total velocity $v = \sqrt{v_x^2 + v_y^2}$.
   - Update velocity components using:

$$v_x(t + \Delta t) = v_x(t) + \frac{dv_x}{dt} \cdot \Delta t$$

$$v_y(t + \Delta t) = v_y(t) + \frac{dv_y}{dt} \cdot \Delta t$$

   - Update position using:

$$x(t + \Delta t) = x(t) + v_x(t) \cdot \Delta t$$

$$y(t + \Delta t) = y(t) + v_y(t) \cdot \Delta t$$

   - Increment time t by Δt.
5. **Continue until** the balls collide.

7 Implementation

7.1 Constants and Parameters

This section defines the constants used in the simulation.

- **Physical Constants:**
  - Gravity (‘g‘): 9.81 m/s^2
  - Air density (‘rho‘): 1.225 kg/m^3
  - Drag coefficient (‘Cd‘): 0.47
  - Cross-sectional area (‘A‘): 0.01 m^2
  - Mass (‘m‘): 0.045 kg
  - Radius (‘r‘): 0.15 m
- **Simulation Parameters:**
  - Time step (‘$delta_t$‘): 0.01 seconds

```
g = 9.81
rho = 1.225
Cd = 0.47
A = 0.01
m = 0.045
r = 0.15
delta_t = 0.01
```

7.2 Target Detection Using Edge Detection

Detect targets in an image using Sobel edge detection.

- **Explanation of Sobel Edge Detection:** The Sobel edge detection operator is a simple convolutional filter used to detect edges in an image. It calculates the gradient of the image intensity in both horizontal (‘x‘) and vertical (‘y‘) directions. The operator highlights regions of high spatial frequency, which correspond to edges in the image.

  The Sobel operator uses two kernels (filters) to compute the gradients:

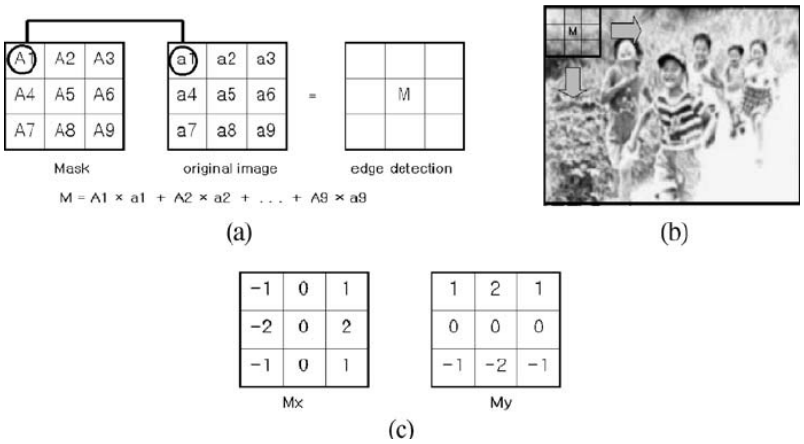
  1. **Sobel X (Horizontal gradient):**

$$\text{Sobel}_X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

  2. **Sobel Y (Vertical gradient):**

$$\text{Sobel}_Y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

  These filters are applied to the image in a sliding window fashion, where each pixel is updated based on its neighbors. The result is the gradient magnitude at each pixel, which indicates the strength of the edge. After calculating the gradients, the image is thresholded to keep only significant edges.



- **Steps:**
  - Convert to grayscale.
  - Apply Sobel filters for edges.
  - Extract contours and centroids for target detection.

```
def apply_sobel_edge_detection(image_path):
    image = cv2.imread(image_path)
    gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    sobel_x, sobel_y = get_sobel_filters()
    grad_x, grad_y = apply_filters(gray_image, sobel_x, sobel_y)
    grad_magnitude = np.sqrt(grad_x ** 2 + grad_y ** 2)
    edges = detect_edges(grad_magnitude)
    contours = extract_contours(edges)
    return get_targets(contours)
```

7.3 Simulating the Projectile's Trajectory

Simulate the motion of a projectile with gravity and drag forces.

- **Steps:**
  - Decompose initial velocity into components.
  - Calculate position and velocity at each time step.
  - Check for collisions with targets.

```
def simulate_trajectory(v_0, theta, x_0=2, y_0=2, targets=None):
    theta_rad = np.radians(theta)
    v_0x, v_0y = get_velocity_components(v_0, theta_rad)
    x, y, vx, vy = initialize_trajectory(x_0, y_0, v_0x, v_0y)
    while y >= 0:
        vx, vy, x, y = update_trajectory(vx, vy, x, y)
        if check_target_collision(x, y, targets):
            return True, x, y
    return False, x, y
```

7.4 Finding the Optimal Launch Parameters

Iteratively adjust angle and velocity to hit all targets.

- **Steps:**
  - Simulate different trajectories.
  - Adjust launch parameters based on success.

```
def shooting_method(targets):
    angle, velocity = 20, 20
    hit_targets = []
    attempts = 0
    max_attempts = 100
    while not all_targets_hit(hit_targets, targets) and attempts < max_attempts:
        success, x, y = simulate_trajectory(velocity, angle)
        if success:
            hit_targets.append((x, y))
        adjust_parameters(angle, velocity)
        attempts += 1
    return hit_targets
```

7.5 Visualizing the Trajectory

Generate a plot of the trajectory and targets.

- **Steps:**
  - Plot detected targets and the trajectory.
  - Highlight starting point and paths.

```
def generate_trajectory_plot(image_path, x_0=2, y_0=2):
    targets = apply_sobel_edge_detection(image_path)
    plot_trajectory(targets, x_0, y_0)
```

7.6 Creating an Animated Visualization

Create an animation of the projectile's motion.

- **Steps:**
  - Animate projectile movement frame by frame.
  - Save as a video file.

```
def generate_trajectory_animation(image_path, x_0=2, y_0=2,
                                  video_filename="trajectory_motion.mp4"):
    targets = apply_sobel_edge_detection(image_path)
    animate_trajectory(targets, video_filename)
```

8 Test Case: Trajectory Plot Generation and Target Detection

This test case involves generating a projectile trajectory plot based on edge-detected targets in an image. The image is processed using Sobel edge detection to identify potential targets, and the trajectory of the projectile is simulated to check if it hits the identified targets.

8.1 Function: generate trajectory plot

This function performs the following steps:

1. **Image Processing:**
   - Reads the input image.
   - Applies Sobel edge detection to find potential targets.
   - Scales the detected targets based on the image dimensions.
2. **Trajectory Simulation:**
   - Uses the shooting method function to simulate the projectile's motion, iterating over different launch angles and velocities to hit all targets.

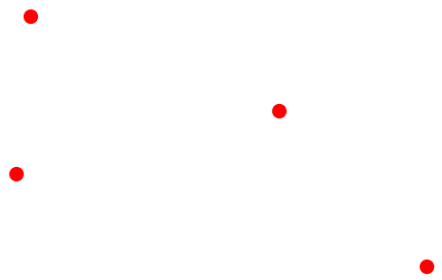
3. **Plot Generation:**
    - Plots the trajectory of the projectile.
    - Marks the targets and starting point.
    - Saves the plot.

8.2 Expected Outcome:

- The function should generate a plot showing the projectile's trajectory with the detected targets.
- If a target is hit during the simulation, the function should correctly display the projectile's path, hitting the target.

8.3 Test Case Inputs:

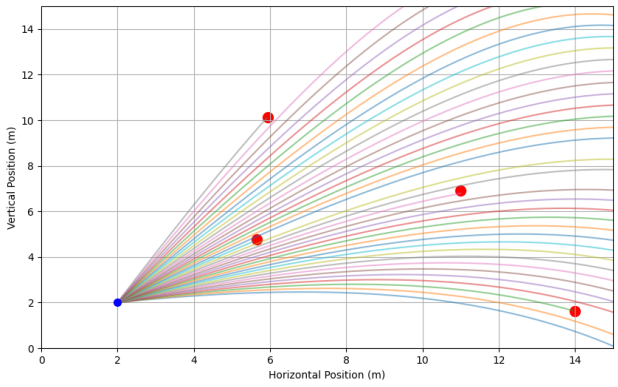
- **Image:** Input image used for target detection via Sobel edge detection.
- **Initial Conditions:** Starting position of the projectile (x_0=2, y_0=2), plot dimensions (plot-width=15, plot-height=15).
    - Image



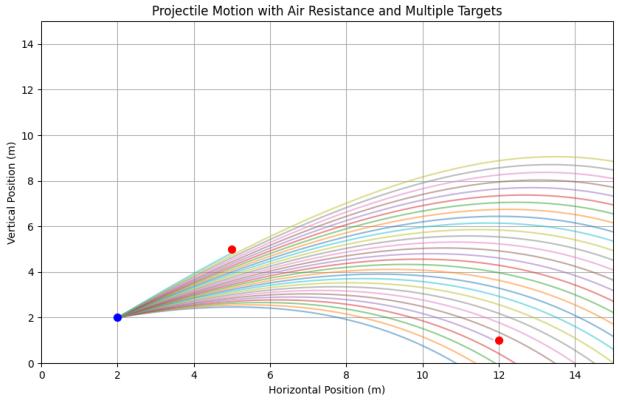
    - Manual
        * targets = [(12, 1), (5, 5)]

8.4 Test Case

- The generated trajectory showing the projectile's path and the detected targets.



—



—

8.5 Results:

- If all targets are successfully hit, the test case passes.
- If any targets are missed, the function needs adjustments to the simulation parameters or edge detection accuracy.

```
Ball hit the target at (13.993548387096773, 1.6233766233766234) with angle 13.0°
and velocity 21.0 m/
```

```
Ball hit the target at (10.993548387096775, 6.915584415584416) with angle 34.0°
and velocity 28.0 m/

Ball hit the target at (5.651612903225806, 4.7727272727272725) with angle 38.5°
and velocity 29.5 m/

Ball hit the target at (5.941935483870967, 10.12987012987013) with angle 65.5°
and velocity 38.5 m/

All targets hit after 38 attempts!
```

9 Observations

The following observations can be made regarding the coverage of projectile trajectories:

1. **Uniqueness of Trajectories**: Every combination of initial angle and velocity, within their respective ranges, will result in a unique trajectory.

2. **Exploration of Trajectories**: By systematically reducing the launch angle (ranging from 0° to 90°) and increasing the initial velocity (covering all relevant values), the method will eventually explore the entire space of possible trajectories.

3. **Time Complexity**: While the method of exploring the space of projectile trajectories by varying angle and velocity is effective, it suffers from a significant time complexity drawback. Specifically, the number of possible angle-velocity combinations grows quickly with finer sampling increments. This results in a computationally expensive process as the number of iterations increases.

   The time complexity can be approximated as $O(n \cdot m)$, where n represents the number of angle values and m represents the number of velocity values. In practical terms, this means that for higher precision (smaller increments in angle and velocity), the method will require a large number of simulations to cover the entire space, which could lead to excessive computation time, especially when aiming to explore a wide range of conditions.

9.1 Mathematical Intuition

The motion of the projectile under the influence of air resistance can be described by the following equations:

- $x(t) = v_0 \cdot \cos(\theta) \cdot t$
- $y(t) = v_0 \cdot \sin(\theta) \cdot t - \frac{1}{2}gt^2$

These equations are influenced by both the initial angle θ and the initial velocity v_0. By iterating through a range of angle and velocity values, a wide variety of trajectories can be generated. As the angle is reduced and the velocity increased, a broad spectrum of possible paths is sampled, ensuring that for any target within the projectile's reach, there will be a specific combination of angle and velocity that will hit the target.

- Initial conditions
  - angle = 16
  - velocity = 12
  - $\max_{\text{attempts}} = 1000$