Ball Motion Analysis

Mariami Garuchava

December 24, 2024

1 Overview

This document describes the steps involved in simulating the motion of a ball in a video, considering both gravitational and air drag forces. The motion is simulated using the Runge-Kutta method to solve the differential equations of motion. The process involves detecting the ball's diameter in video frames, computing its mass and velocity, and estimating the drag coefficient.

2 Steps in the Code

2.1 Edge Detection

In this step, the ball is detected in each frame using the Canny edge detection method. The Canny edge detection algorithm works by identifying areas of rapid intensity change, which helps locate the edges of objects in an image. The ball's edges are identified and then used to compute the ball's diameter. The steps are as follows:

1. Convert the frame to grayscale to simplify the image. 2. Apply Gaussian blur to reduce noise and improve edge detection. 3. Use the Canny edge detection algorithm to detect the edges. 4. Find the largest contour, which corresponds to the ball, and fit a circle around it.

The formula for the Canny edge detection is:

$$\text{edges} = \text{Canny}(\text{blurred frame}, \text{low threshold}, \text{high threshold})$$

This allows us to detect the ball's edges in the image.

```
def detect_ball_diameter(frame, edge_param):
    gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    blurred_frame = cv2.GaussianBlur(gray_frame, (5, 5), 0)
    edges = cv2.Canny(blurred_frame, edge_param['low_threshold'], edge_param['high
```

2.2 Scaling Factor Calculation

The scaling factor is crucial because it converts measurements in pixels (from the video) into real-world units (meters). This is done by comparing the size of a known reference object in the real world to its pixel size in the video. The scaling factor can be computed as:

$$\text{scaling factor} = \frac{\text{reference object real size}}{\text{reference object pixel size}}$$

This scaling factor is then used to convert pixel-based measurements (such as the ball's diameter) to real-world measurements (such as meters).

```
def estimate_scaling_factor(reference_object_real_size, reference_object_pixel_siz
    scaling_factor = reference_object_real_size / reference_object_pixel_size
    return scaling_factor
```

2.3 Ball Mass Computation

To compute the ball's mass, we use the formula for the volume of a sphere, which is given by:

$$V = \frac{4}{3}\pi r^3$$

where r is the radius of the ball in real-world units (meters). The ball's mass is then calculated by multiplying the volume by the material density (ρ) of the ball (assumed to be rubber, with a density of 1200 kg/m^3):

$$\text{mass} = \rho \times V$$

where: - $\rho = 1200\,\text{kg/m}^3$ (density of rubber) - r is the radius of the ball in meters.

```
def compute_ball_mass(diameter_in_pixels, scaling_factor, density):
    radius_real_world = (diameter_in_pixels / 2) * scaling_factor
```

```
    volume = (4 / 3) * np.pi * (radius_real_world ** 3)
    mass = density * volume
    return mass
```

2.4 Velocity Calculation

The velocity of the ball is calculated by measuring the displacement between two consecutive positions in the frames. The displacement in pixels is first calculated as:

$$\Delta x = x_i - x_{i-1}, \quad \Delta y = y_i - y_{i-1}$$

where x_i and y_i are the coordinates of the ball at frame i.

The displacement in meters is then calculated by multiplying the displacement in pixels by the scaling factor:

$$\text{displacement in meters} = \sqrt{(\Delta x)^2 + (\Delta y)^2} \times \text{scaling factor}$$

Finally, the velocity is calculated by dividing the displacement by the time interval dt, which is the time between frames (in seconds):

$$v = \frac{\text{displacement in meters}}{dt}$$

The average velocity is then computed over all frame intervals.

```
def compute_velocity(positions, dt, scaling_factor):
    velocities = []
    for i in range(1, len(positions)):
        delta_x = positions[i][0] - positions[i - 1][0]
        delta_y = positions[i][1] - positions[i - 1][1]
        displacement_meters = np.sqrt(delta_x ** 2 + delta_y ** 2) * scaling_facto
        velocity = displacement_meters / dt
        velocities.append(velocity)
    return np.mean(velocities) if velocities else 0
```

2.5 Drag Coefficient Estimation

The drag coefficient C_d quantifies the resistance of the ball moving through air. It can be estimated using the following formula derived from the force balance between drag and gravity:

$$F_d = \frac{1}{2}\rho A C_d v^2$$

where: - F_d is the drag force. - ρ is the air density (1.225 kg/m^3). - A is the cross-sectional area of the ball. - C_d is the drag coefficient. - v is the velocity of the ball.

Assuming the drag force is equal to the gravitational force acting on the ball ($F_d = mg$), we can solve for the drag coefficient as:

$$C_d = \frac{2mg}{\rho A v^2}$$

where: - m is the mass of the ball. - g is the acceleration due to gravity (9.81 m/s^2).

```
def estimate_drag_coefficient(velocity, ball_diameter, air_density=1.225):
    radius = ball_diameter / 2
    area = np.pi * radius ** 2
    mass = compute_ball_mass(ball_diameter, 1, 1200)  # 1200 is the density for ru
    drag_force = mass * 9.81
    if velocity > 0:
        drag_coefficient = (2 * drag_force) / (air_density * area * velocity ** 2)
    else:
        drag_coefficient = 0
    return drag_coefficient
```

2.6 Numerical Simulation (Runge-Kutta Method)

The motion of the ball is simulated using the Runge-Kutta method. This method allows us to solve the differential equations of motion numerically. The equations of motion, including the effects of gravity and air drag, are:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ a_x \\ a_y \end{bmatrix}$$

where: - x, y are the position coordinates of the ball. - v_x, v_y are the velocity components. - a_x, a_y are the accelerations due to drag and gravity.

The accelerations due to drag and gravity are calculated as:

$$a_x = \frac{F_{d_x}}{m}, \quad a_y = \frac{F_{d_y}}{m} - g$$

where F_{d_x}, F_{d_y} are the drag forces in the x and y directions, and g is the acceleration due to gravity.

The Runge-Kutta method approximates the new state of the system using the weighted average of slopes:

$$\text{state}_{n+1} = \text{state}_n + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where k_1, k_2, k_3, k_4 are the slopes calculated at different stages within the time step dt.

```
def motion_equations(t, state, drag_coefficient, air_density, radius, mass, g=9.81
    x, y, vx, vy = state
    speed = np.sqrt(vx ** 2 + vy ** 2)
    drag_force_x = -0.5 * air_density * A * drag_coefficient * speed * vx
    drag_force_y = -0.5 * air_density * A * drag_coefficient * speed * vy
    ax = drag_force_x / mass
    ay = (-g + drag_force_y / mass)
    return [vx, vy, ax, ay]
```

This method allows for the accurate simulation of the ball's trajectory, including the effects of both gravity and air drag.

3 Results

3.1 Plot of the Ball's Trajectory

Below is the plot showing the trajectory of the ball as it moves under the influence of gravity and air drag.

3.2 Terminal Output for Ball Mass, Velocity, and Drag Coefficient

Here is the terminal output showing the estimated ball mass, average velocity, and drag coefficient for both videos.

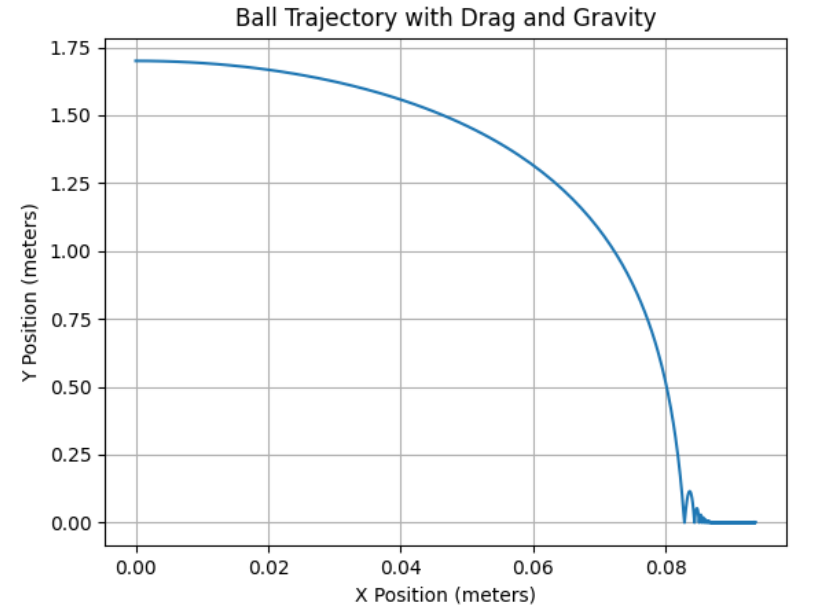

Figure 1:

```
Processing Good Lighting Video:
Scaling Factor: 0.0033
Estimated Ball Mass: 0.83629 kg
Average Ball Velocity: 14.63 m/s
Estimated Drag Coefficient: 64.7187

Processing Bad Lighting Video:
Scaling Factor: 0.0033
Estimated Ball Mass: 0.83629 kg
Average Ball Velocity: 0.16 m/s
Estimated Drag Coefficient: 457369.6356
Initial Velocity for the good lighting video: 0.25 m/s
```

Figure 2:

4 Testing

4.1 Why the Code Works Well for the First Video

The code performs well on the first video, which is recorded in good lighting and against a plain white background, due to several factors:

- **Clear Contrast:** The ball is clearly distinguishable from the background in the video, allowing for accurate edge detection. The Canny edge detection algorithm works best when there is a high contrast between the object and its surroundings.
- **Consistent Lighting:** The uniform lighting in the first video ensures that the edges of the ball are well-defined, without significant noise from shadows or reflections. This consistency allows the algorithm to detect the ball's edges reliably.
- **Stable Background:** The plain white background in the first video ensures minimal interference from complex patterns, which allows the algorithm to focus solely on the ball and its motion.

As a result, the ball's diameter is accurately calculated, and the velocity and drag coefficient estimates are reasonable.

4.2 Why the Code Calculates Poorly for the Second Video

On the other hand, the second video, which is recorded in very bad lighting, leads to inaccurate calculations for several reasons:

- **Poor Contrast:** The ball and the background are not well-separated due to the poor lighting. This makes it difficult for the edge detection algorithm to accurately identify the edges of the ball. Shadows, overexposure, and reflections may cause the edges to become blurred or incomplete, leading to incorrect diameter measurements.
- **Inconsistent Lighting:** The uneven lighting in the second video introduces noise in the image, such as areas of the ball appearing darker or brighter than others. These inconsistencies confuse the edge detection algorithm, making it harder to detect the ball's shape correctly.
- **unrealistic results:** Both videos were recorded by dropping the same ball on the same floor from the same height, so the results should have been close, but the outputs differ significantly from each other.

Due to incorrect velocity and drag coefficient calculations for the second video, I only plotted the trajectory for the first video.