Computational Project 2
(Ballin')

Lado Turmanidze

December 24, 2024

1 Problem Statement

- ❄ Determine the velocity, mass, and drag coefficient of a ball by analyzing its motion in a video from point A to point B.
- ❄ Develop suitable tests.
- ❄ Use ball motion ODEs and numerical methods.

1.1 Tasks

- ♣ Formulate algorithm, explain your approach.
- ♦ Describe properties of numerical methods.
- ♥ Select or generate one video for which your approach works fine and another for which it produces incorrect results.
- ♠ Understanding limitations of the methods and its application domain is an important skill. Explain why your method works in one case and why it does not work in another case.

1.2 For which problems does this algorithm work?

The code provided here works for 2D models, where ball is thrown up in the air, follows parabolic trajectory and stops at some point, or goes "out" of the screen. Video should only contain ball and the background. Colors of the ball and the background should be chosen in such a way, that it is possible to use edge detection and contours to identify and extract centroid of the ball.

1.3 ODEs

The motion of the ball is described by:

$$\frac{dx}{dt} = v_x$$
$$\frac{dy}{dt} = v_y$$
$$\frac{dv_x}{dt} = -\frac{k}{m} v_x \sqrt{v_x^2 + v_y^2}$$
$$\frac{dv_y}{dt} = -g - \frac{k}{m} v_y \sqrt{v_x^2 + v_y^2}$$

with initial values conditions: $x(0) = x_0$, $y(0) = y_0$, $v_x(0) = v_{x_0}$, $v_y(0) = v_{y_0}$ where:

- ↔ v_x: The horizontal component of the ball's velocity.
- ↕ v_y: The vertical component of the ball's velocity.
- ↓ g: The acceleration due to gravity $\approx 9.81 \frac{m}{s^2}$ on Earth's surface.
- ➟ k: The drag coefficient, incorporates air resistance in the model.
- ❍ m: The mass of the ball.

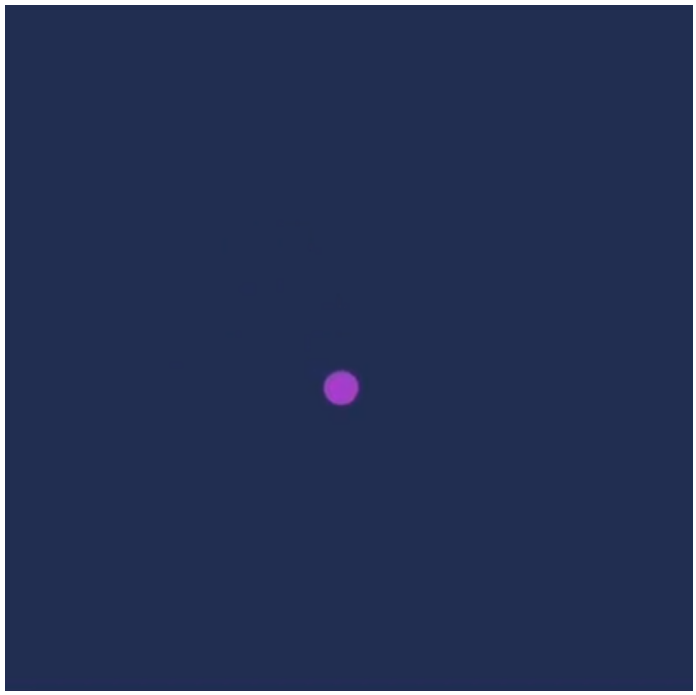

Figure 1: Single frame from the test video

2 Algorithm & Implementation

I will implement methods step by step and explain their use:

```
def detect_ball_and_velocity(frame1: np.ndarray, frame2: np.ndarray
                                , p2m: float = 1.0, f_time: float
                                 = 1 / 30.0):
  gray_scaled_1 = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY)
  gray_scaled_2 = cv2.cvtColor(frame2, cv2.COLOR_BGR2GRAY)
  blurred_1 = cv2.GaussianBlur(gray_scaled_1, (3, 3), 0)
  blurred_2 = cv2.GaussianBlur(gray_scaled_2, (3, 3), 0)

  edges_1 = cv2.Canny(blurred_1, 50, 150)
  edges_2 = cv2.Canny(blurred_2, 50, 150)
  contours_1, _ = cv2.findContours(edges_1, cv2.RETR_EXTERNAL, cv2.
                                CHAIN_APPROX_SIMPLE)
  contours_2, _ = cv2.findContours(edges_2, cv2.RETR_EXTERNAL, cv2.
                                CHAIN_APPROX_SIMPLE)

  def find_ball_position(contours):
    for contour in contours:
      area = cv2.contourArea(contour)
      perimeter = cv2.arcLength(contour, True)

      if perimeter == 0:
        continue
      circularity = 4 * np.pi * area / (perimeter * perimeter)

      if 0.7 < circularity < 1.3:
        M = cv2.moments(contour)
        if M["m00"] != 0:
          cx = int(M["m10"] / M["m00"])
          cy = int(M["m01"] / M["m00"])
          return (cx, cy)
    return None

  pos_1 = find_ball_position(contours_1)
  pos_2 = find_ball_position(contours_2)

  if pos_1 is None or pos_2 is None:
    return None, (np.inf, np.inf)

  v_x = (pos_2[0] - pos_1[0]) * p2m / f_time
  v_y = (pos_2[1] - pos_1[1]) * p2m / f_time

  return pos_2, (v_x, v_y)
```

detect_ball_and_velocity, is designed to detect the position of a ball in two frames of video and calculate its velocity based on the change in position between the frames.

The method first grayscales both frames, uses Gaussian blur on both of them and does Canny edge detection. It also finds contours with following parameters for *cv2.findContours*:

a. **cv2.RETR_EXTERNAL**: retrieves only the extreme outer contours. It ignores all the contours that are inside other contours.

b. **cv2.CHAIN_APPROX_SIMPLE**: compresses horizontal, vertical, and diagonal segments and leaves only their end points. This reduces the number of points in the contour, which can save memory and improve performance.

With this contours, we aim to find ball position. For this purpose, we add inner method *find_ball_position*. For each contour, we calculate area and perimeter. If the perimeter is 0 (which can occur for very small or degenerate contours), the function skips to the next iteration of the loop. For circularity, we use formula circularity $= \frac{4\pi \times \text{area}}{\text{perimeter}^2}$. A circularity value close to 1 indicates that the shape is close to a perfect circle. Then, we check if circularity value is between 0.7 and 1.3, which work as lower and upper thresholds. If this condition is satisfied, we calculate centroid. If the contour is approximately circular, the function calculates the moments of the contour using *cv2.moments*. Moments are statistical properties that can be used to compute the centroid (center of mass) of the contour. The condition *M["m00"] != 0* checks if the zeroth moment is not zero to avoid division by zero.

$$\text{M["m00"]} = \sum_{x,y} I(x,y) \text{ where } I(x,y) \text{ is the pixel value at coordinates } (x,y)$$

$$\text{M["m10"]} = \sum_{x,y} x \cdot I(x,y), \quad \text{M["m01"]} = \sum_{x,y} y \cdot I(x,y)$$

We use *find_ball_position* to find center of circle for both frames, and if either on of them is absent, we return *None* and infinity values for position and velocity. Else, we calculate velocities in both directions by subtracting corresponding positions, multiplying by pixels to meters argument and dividing by the time between frames, effectively getting velocity for that direction. Finally, we return position in the second frame and velocity.

Here is the visualization function, which I won't be explaining:

```
def visualize_detection(frame, ball_pos, velocity, k_over_m=None, g
                                    =None):
  output = frame.copy()
  if ball_pos is None or np.isinf(ball_pos[0]) or np.isinf(ball_pos
                                    [1]):
    return output

  ball_pos = tuple(map(int, ball_pos))
  cv2.circle(output, ball_pos, 10, (0, 255, 0), 2)
  scale = 20
  end_point = (int(ball_pos[0] + velocity[0] * scale), int(ball_pos
                                    [1] + velocity[1] * scale))
  cv2.arrowedLine(output, ball_pos, end_point, (0, 0, 255), 2)
```

```
    speed = np.sqrt(velocity[0] ** 2 + velocity[1] ** 2)
    cv2.putText(output, f'Speed: {speed:.2f} m/s', (10, 30), cv2.
                                        FONT_HERSHEY_SIMPLEX, 1, (0, 0,
                                        255), 2)

    if k_over_m is not None:
      cv2.putText(output, f'k/m: {k_over_m:.4f}', (10, 70), cv2.
                                        FONT_HERSHEY_SIMPLEX, 1, (0, 0,
                                        255), 2)
    if g is not None:
      cv2.putText(output, f'g: {g:.4f}', (10, 110), cv2.
                                        FONT_HERSHEY_SIMPLEX, 1, (0, 0,
                                        255), 2)

    return output
```

Now, what we need is to find $\frac{dv_x}{dt}$ and $\frac{dv_y}{dt}$. We implement central finite difference:

```
def central_FD(v_prev: float, v_next: float, dt: float) -> float:
    return (v_next - v_prev) / (2 * dt)
```

We need to solve the following:

$$\begin{bmatrix} -v_x\sqrt{v_x^2+v_y^2} & 0 \\ -v_y\sqrt{v_x^2+v_y^2} & -1 \end{bmatrix} \begin{bmatrix} \frac{k}{m} \\ g \end{bmatrix} = \begin{bmatrix} \frac{dv_x}{dt} \\ \frac{dv_y}{dt} \end{bmatrix}$$

Since determinant of this 2×2 matrix is $v_x\sqrt{v_x^2+v_y^2}$, we get inverse of the above matrix:

$$\begin{bmatrix} -\frac{1}{v_x\sqrt{v_x^2+v_y^2}} & 0 \\ \frac{v_y}{v_x} & -1 \end{bmatrix}$$

So we have to simply multiply this matrix by vector of finite differences to get $\begin{bmatrix} \frac{k}{m} \\ g \end{bmatrix}$:

$$\begin{bmatrix} -\frac{1}{v_x\sqrt{v_x^2+v_y^2}} & 0 \\ \frac{v_y}{v_x} & -1 \end{bmatrix} \begin{bmatrix} \frac{dv_x}{dt} \\ \frac{dv_y}{dt} \end{bmatrix} = \begin{bmatrix} \frac{k}{m} \\ g \end{bmatrix}$$

We can hardcode this calculation in the following function:

```
def get_g_and_k_over_m(v_x, v_y, dv_x, dv_y):
    return (-1 / (v_x * np.sqrt(v_x ** 2 + v_y ** 2)) * dv_x,
      v_y / v_x * dv_x - dv_y)
```

Now comes the main code, where we assemble all of the above and get the results. It is important to note, that *output.mp4* is generated as visualization, but we will only print average of $v_x, v_y, \frac{dv_x}{dt}, \frac{dv_y}{dt}, \frac{k}{m}$ and g:

```
if __name__ == "__main__":
  PATH = 'Not Free Fall.mp4'
  cap = cv2.VideoCapture(PATH)

  if not cap.isOpened():
    print("Error: Could not open video file")
    sys.exit(1)

  ret, prev_frame = cap.read()
  if not ret:
    print("Error: Could not read first frame")
    cap.release()
    sys.exit(1)

  ret, current_frame = cap.read()
  if not ret:
    print("Error: Could not read second frame")
    cap.release()
    sys.exit(1)

  fourcc = cv2.VideoWriter_fourcc(*'mp4v')
  frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
  frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
  fps = int(cap.get(cv2.CAP_PROP_FPS))
  out = cv2.VideoWriter('output.mp4', fourcc, fps, (frame_width,
                                    frame_height))

  temp_dv_x = []
  temp_dv_y = []
  temp_k_over_m = []
  temp_g = []
  temp_v_x = []
  temp_v_y = []
  temp_ball_pos = []

  _, prev_velocity = detect_ball_and_velocity(prev_frame,
                                    current_frame)
  prev_frame = current_frame.copy()

  current_k_over_m = None
  current_g = None

  while True:
    ret, current_frame = cap.read()
    if not ret:
      break

    ball_pos, current_velocity = detect_ball_and_velocity(
                                    prev_frame, current_frame)

    if ball_pos is not None and not np.isinf(current_velocity[0])
                                    and not np.isinf(prev_velocity[0]
                                    ) and current_velocity[0] != 0:
      temp_ball_pos.append(ball_pos)
      temp_v_x.append(current_velocity[0])
      temp_v_y.append(current_velocity[1])
```

```
            d_x = central_FD(prev_velocity[0], current_velocity[0], 1)
            d_y = central_FD(prev_velocity[1], current_velocity[1], 1)
            temp_dv_x.append(d_x)
            temp_dv_y.append(d_y)

            current_k_over_m, current_g = get_k_over_m_and_g(
                                          current_velocity[0],
                                          current_velocity[1], d_x, d_y)
            temp_k_over_m.append(current_k_over_m)
            temp_g.append(current_g)
        else:
            current_k_over_m = None
            current_g = None

    output_frame = visualize_detection(current_frame, ball_pos,
                                       current_velocity,
                                       current_k_over_m, current_g)
    cv2.imshow('Ball Tracking', output_frame)
    out.write(output_frame)

    prev_frame = current_frame.copy()
    prev_velocity = current_velocity

cap.release()
out.release()
cv2.destroyAllWindows()

dv_x = np.array(temp_dv_x)
dv_y = np.array(temp_dv_y)
k_over_m = np.array(temp_k_over_m)
g = np.array(temp_g)
v_x = np.array(temp_v_x)
v_y = np.array(temp_v_y)
ball_pos = np.array(temp_ball_pos)

print(f"Number of valid frames: {len(dv_x)}")
print(f"Average v_x: {np.mean(v_x):.4f}")
print(f"Average v_y: {np.mean(v_y):.4f}")
print(f"Average dv_x: {np.mean(dv_x):.4f}")
print(f"Average dv_y: {np.mean(dv_y):.4f}")
print(f"Average k/m: {np.mean(k_over_m):.4f}")
print(f"Average g: {np.mean(g):.4f}")
```

The results look promising:

```
Number of valid frames: 83
Average v_x: 38.3133
Average v_y: 29.6386
Average dv_x: 8.6747
Average dv_y: 4.5181
Average k/m: -0.0016
Average g: 11.7470
```

Figure 2: Output of Entire Code

3 When does this code work and when does it fail?

As mentioned above, this code works if video satisfies 1.2. It fails, when there are multiple balls, or multiple different object in the video. I tested on video, where I added parabolic shape which is concave downward. The ball starts going up on the ramp, but before reaching the extremum point, it falls back.

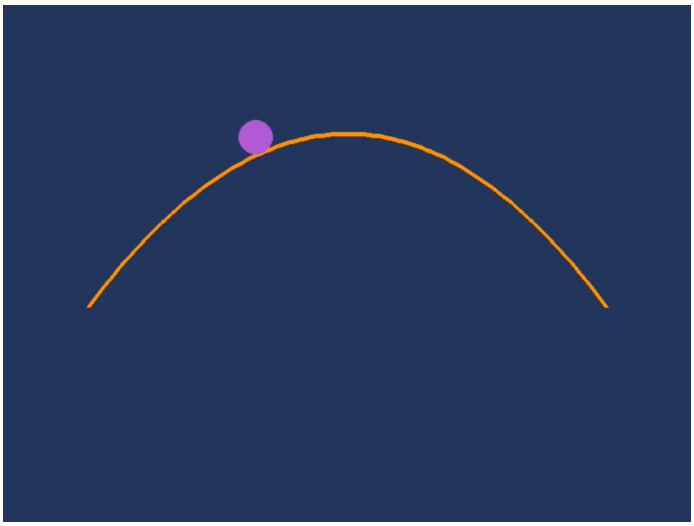

Figure 3: Frame, where ball is at the highest point it can reach

There are number of factors, which prevent this video outputting correct results. First and the most obvious factor is addition of ramp, which simply messes up the ball detection algorithm. Second factor is the ball going backwards, rolling back down the ramp, which messes up the calculation as then, the problem cannot be fully described with given ordinary differential equations.

```
Average v_x: nan
Average v_y: nan
Average dv_x: nan
Average dv_y: nan
Average k/m: nan
Average g: nan
/home/apeiron/KIU/NP/pcodes/.venv/lib/python3.10/site-packages/numpy/_core/fromnumeric.py:3904: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/home/apeiron/KIU/NP/pcodes/.venv/lib/python3.10/site-packages/numpy/_core/_methods.py:147: RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

Figure 4: Output "Fail Video.mp4"