Project 2: Intercept a moving ball

Statement of the problem
Code explanation
- Video Processing
  - Code snippets explanation
- Ball motion ODE
- Ball motion
  - Parameters
  - Functionality
  - Mathematical
- Shooting method
  - Parameters:
  - Functionality:
  - Mathematical
- Ball throw trajectory for time T
  - Parameters:
  - Functionality:
  - Mathematical
- Shooting velocity for T time
  - Parameters:
  - Functionality:
  - Mathematical
- Simulation
  - Checking if velocity and position data exist
  - Predict future positions for the first object
  - Determine the target position at time T
  - Compute shooter's trajectory to intercept the target
  - **Visualization**
  - Animation initialization
  - Animation update function
  - Run the animation

Result
Tests
- Successful Test
- Unsuccessful Test
  - Limitations

Conclusion

Statement of the problem

- Input: part of a video of a moving ball
- Task: Throw a ball and intercept moving ball
- Output: Animation corresponding to the task description
- Test: Test case description
- Methodology: should contain problem formulation, including equation with initial and boundary condition, method of solution, algorithm

The code models the motion of a projectile and calculates the trajectory of a ball using a combination of numerical methods and physics principles. It also detects the ball's motion in a video file, computes its velocity, and simulates its trajectory while identifying the necessary parameters (e.g., gravity, drag coefficient). The problem involves tracking a moving object and predicting its trajectory for visualization.

Code explanation

Video Processing

This code tracks a single moving object in a video feed by detecting changes in the scene, estimating the object's position, and calculating its velocity in real time. It uses background subtraction, contour detection, and morphological operations to isolate and process the moving object.

▼ Code snippets explanation

1. Video Stream Initialization

- `while cap.isOpened()`: The loop continues as long as the video stream (`cap`) is open and `frame_count < 90`. We add `frame_count` to to later predict other frames.
- `cap.read()`: Captures the next frame from the video stream, If no frame is returned (`ret=False` or `frame=None`), the loop exits.

2. Frame Processing

- **Convert frame to grayscale**: `g = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)` converts the frame to grayscale to simplify processing.
- **Background subtraction**: `m = back_sub.apply(g)` uses a background subtraction algorithm (`back_sub`) to identify moving objects by removing the static background.
- **Thresholding**: `cv2.threshold` converts the background-subtracted image into a binary image (`th`), where moving areas are white (255) and the rest is black (0).
- **Morphological Operations**: `cv2.morphologyEx` applies opening and closing operations to reduce noise and smooth out the detected regions.

3. Object Detection

- **Find contours**: `cv2.findContours` detects the boundaries of white regions in the binary image, representing moving objects.
- **Process largest contour**: If there are contours and at least 30 frames have passed, the largest contour is selected using `max(contours, key=cv2.contourArea)`. **Circle fitting**: `cv2.minEnclosingCircle` finds the smallest circle that can enclose the largest contour. The circle's center `(x, y)` and radius `r` are determined. The circle is drawn on the frame if `r` (radius) is between 5 and 100.

4. Object Tracking

- **Track position**: The circle's center `(x, y)` is stored in the `positions` list.
- **Calculate velocity**: If there is a previous center (`prev_c`), calculate the difference in position `(dx, dy)` and divide by the frame time (`dtf = 1/fps`) to estimate velocity `(vx, vy)`. Velocity components are stored in `vels_x` and `vels_y`.

5. Display and Interaction

- **Display frame**: The processed frame, with the detected object marked by a circle, is displayed in a window titled "Single Moving Object Detection."
- **User interaction**: Pressing the `q` key breaks the loop and exits the program.

6. Cleanup

- Releases the video capture (`cap.release()`) and closes all OpenCV windows (`cv2.destroyAllWindows()`).

```
while cap.isOpened() and frame_count < 70:
    ret, frame = cap.read()
    if not ret or frame is None:
        break
    frame_count += 1
    g = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    m = back_sub.apply(g)
    _, th = cv2.threshold(m, 50, 255, cv2.THRESH_BINARY)
    k = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
    c1 = cv2.morphologyEx(th, cv2.MORPH_OPEN, k)
    c2 = cv2.morphologyEx(c1, cv2.MORPH_CLOSE, k)
    contours, _ = cv2.findContours(c2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    if contours and frame_count > 10:
        largest_contour = max(contours, key=cv2.contourArea)
        (x, y), r = cv2.minEnclosingCircle(largest_contour)
        if 5 < r < 100:
            c = (int(x), int(y))
            cv2.circle(frame, c, int(r), (0, 0, 255), 2)
            positions.append(c)
            if prev_c is not None:
                dtf = 1/fps
                dx, dy = c[0] - prev_c[0], c[1] - prev_c[1]
                vx, vy = dx/dtf, dy/dtf
                vels_x.append(vx)
                vels_y.append(vy)
            prev_c = c
    cv2.imshow('Single Moving Object Detection', frame)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```

Ball motion ODE

Ball Motion

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

$$x(0) = x_0, \quad y(0) = y_0, \quad v_x(0) = v_{x0}, \quad v_y(0) = v_{y0}$$

- 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 $\approx 9.81\ m/s^2$ on Earth's surface.
- k: The drag coefficient, incorporates air resistance in the model.
- m: The mass of the ball.

Ball motion

The `ball_motion` function models the motion of a projectile under the influence of gravity and air resistance, using an iterative numerical approach based on **Euler's method**.

Parameters

- `vx`, `vy`: Initial velocities of the ball in the x and y directions.
- `g0`, `k_m0`: Initial gravity (`g0`) and drag coefficient (`k_m0`).
- `x0`, `y0`: Initial position of the ball.
- `tx`, `ty`: Target position the ball should ideally reach.
- `dt`: Time step for the simulation.
- `ms`: Number of iterations (time steps) for simulation.

Functionality

1. Computes the acceleration (`ax`, `ay`) due to drag and gravity at each step.
2. Updates the velocities (`vx`, `vy`) and positions (`x`, `y`) using these accelerations and time step (`dt`).
3. Returns the differences (`tx - x`, `ty - y`) between the target position and the final simulated position.

```
def ball_motion(vx, vy, g0, k_m0, x0, y0, tx, ty, dt, ms=50):
    x, y = x0, y0
    g, k_m = g0, k_m0
    for _ in range(ms):
        ax = -k_m * vx * np.sqrt(vx*vx + vy*vy)
        ay = -g - k_m * vy * np.sqrt(vx*vx + vy*vy)
        vx += ax * dt
        vy += ay * dt
        x += vx * dt
        y += vy * dt
    return tx - x, ty - y
```

▼ Mathematical

Euler's method updates velocity and position iteratively over discrete time steps dt. At each iteration, velocity and position is updated:

v_x update:

$$v_x^{\text{new}} = v_x + a_x \cdot dt$$

v_y update:

$$v_y^{\text{new}} = v_y + a_y \cdot dt$$

x update:

$$x^{\text{new}} = x + v_x \cdot dt$$

y update:

$$y^{\text{new}} = y + v_y \cdot dt$$

Shooting method

The `shooting_method` function aims to iteratively estimate the values of gravitational acceleration (`g`) and air resistance coefficient (`k_m`) that best explain the observed motion of a ball. This is achieved using numerical methods, specifically Newton-Raphson optimization, applied to a system of equations.

Parameters:

- `vx0`, `vy0`: Initial velocities of the ball.
- `x0`, `y0`: Initial position of the ball.
- `tx`, `ty`: Target position of the ball.
- `dt`: Time step for simulation.
- `tol`: Tolerance for the error in reaching the target position.
- `it`: Maximum number of iterations for the optimization.

Functionality:

1. Simulates the ball's motion using `ball_motion` and computes the errors (`ex`, `ey`) between the simulated and target positions.
2. Approximates the Jacobian matrix (`J`) numerically by slightly perturbing `g` and `k_m` to observe changes in the errors.
3. Uses the Jacobian and the error vector (`F`) to solve a linear system (`J * dp = -F`) and update `g` and `k_m`.
4. Stops iterating when the error is within the specified tolerance or the maximum iterations are reached.
5. Returns the estimated values of `g` and `k_m`.

```
def shooting_method(vx0, vy0, x0, y0, tx, ty, dt, tol=0.01, it=100):
    g, k_m = 0.0, 0.0
    h = 1e-5
    for _ in range(it):
        ex, ey = ball_motion(vx0, vy0, g, k_m, x0, y0, tx, ty, dt)
        if abs(ex) < tol and abs(ey) < tol:
            return g, k_m
        exg, eyg = ball_motion(vx0, vy0, g + h, k_m, x0, y0, tx, ty, dt)
        exk, eyk = ball_motion(vx0, vy0, g, k_m + h, x0, y0, tx, ty, dt)
        dExdg, dEydg = (exg - ex)/h, (eyg - ey)/h
```

```
        dExdk, dEydk  = (exk - ex)/h, (eyk - ey)/h
        J = np.array([[dExdg, dExdk],[dEydg, dEydk]])
        F = np.array([ex, ey])
        try:
            dp = np.linalg.solve(J, -F)
        except:
            break
        g += dp[0]
        k_m += dp[1]
    return g, k_m
```

▼ Mathematical

The goal is to minimize the errors in the final x and y positions (e_x, e_y) of the ball, calculated using the motion model:

$$e_x(g, k_m) = t_x - x_f(g, k_m)$$

$$e_y(g, k_m) = t_y - y_f(g, k_m)$$

The Jacobian matrix, $\mathbf{J}$, represents how the errors change with respect to the parameters g and k_m:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial e_x}{\partial g} & \frac{\partial e_x}{\partial k_m} \\ \frac{\partial e_y}{\partial g} & \frac{\partial e_y}{\partial k_m} \end{bmatrix}$$

The partial derivatives are approximated numerically using finite differences with a small h:

$$\frac{\partial e_x}{\partial g} \approx \frac{e_x(g + h, k_m) - e_x(g, k_m)}{h}$$

$$\frac{\partial e_y}{\partial g} \approx \frac{e_y(g + h, k_m) - e_y(g, k_m)}{h}$$

$$\frac{\partial e_x}{\partial k_m} \approx \frac{e_x(g, k_m + h) - e_x(g, k_m)}{h}$$

$$\frac{\partial e_y}{\partial k_m} \approx \frac{e_y(g, k_m + h) - e_y(g, k_m)}{h}$$

Using the Jacobian matrix, the optimization problem is expressed as a system of linear equations:

$$\mathbf{J} \cdot \Delta \mathbf{p} = -\mathbf{F}$$

Where:

- $\Delta \mathbf{p} = [\Delta g, \Delta k_m]^T$ is the update vector for g and k_m
- $\mathbf{F} = [e_x, e_y]^T$ is the error vector.

The update vector $\Delta \mathbf{p}$ is computed by solving:

$$\Delta \mathbf{p} = -\mathbf{J}^{-1} \cdot \mathbf{F}$$

The parameters g and k_m are updated iteratively:

$$g \leftarrow g + \Delta g$$

$$k_m \leftarrow k_m + \Delta k_m$$

The process continues until the errors $|e_x|$ and $|e_y|$ are below the specified tolerance (tol), or the maximum number of iterations (it) is reached.

Ball throw trajectory for time T

The `ball_throw_trajectory` function calculates the trajectory of a projectile considering the effects of gravity and air resistance. It uses **Euler's method** for numerical integration to solve the equations of motion over a specified time interval t with time step dt.

Parameters:

- `vx0`, `vy0`: Initial velocities of the ball.
- `x0`, `y0`: Initial position of the ball.
- `t`: Total time for the simulation.
- `dt`: Time step for simulation.
- `g`, `k_m`: Gravity and drag coefficient.

Functionality:

1. Divides the total time `t` into discrete time steps (`steps`).
2. At each step, computes the accelerations, updates velocities, and positions as in `ball_motion`.
3. Returns the final position of the ball after time `t`.

```
def ball_throw_trajectory(vx0, vy0, x0, y0, t, dt, g, k_m):
    steps = int(round(t/dt))
    vx, vy = vx0, vy0
    x, y = x0, y0
    for _ in range(steps):
        ax = -k_m * vx * np.sqrt(vx*vx + vy*vy)
        ay = -g - k_m * vy * np.sqrt(vx*vx + vy*vy)
        vx += ax * dt
        vy += ay * dt
        x += vx * dt
        y += vy * dt
    return x, y
```

▼ Mathematical

Euler's method updates velocity and position iteratively over discrete time steps dt. At each iteration, velocity and position is updated:

v_x update:

$$v_x^{\text{new}} = v_x + a_x \cdot dt$$

v_y update:

$$v_y^{\text{new}} = v_y + a_y \cdot dt$$

x update:

$$x^{\text{new}} = x + v_x \cdot dt$$

y update:

$$y^{\text{new}} = y + v_y \cdot dt$$

Shooting velocity for T time

This function calculates the initial velocities (`vx0`, `vy0`) required for the ball to reach a specific target within a specified time `T`.

Parameters:

- `xs`, `ys`: Initial position of the ball.
- `xe`, `ye`: Target position of the ball.
- `T`: Time within which the ball should reach the target.
- `dt`: Time step for simulation.
- `g`, `k_m`: Gravity and drag coefficient.
- `vxg`, `vyg`: Initial guesses for velocities.
- `tol`: Tolerance for the error in reaching the target position.
- `mi`: Maximum number of iterations for optimization.

Functionality:

1. Simulates the ball's motion using `ball_throw_trajectory` and computes the errors (`fx`, `fy`) between the final position and the target.
2. Numerically approximates the Jacobian matrix (`J`) by slightly perturbing `vx0` and `vy0` to observe changes in the errors.
3. Solves a linear system (`J * dV = -F`) to update `vx0` and `vy0`.
4. Stops iterating when the error is within the specified tolerance or the maximum iterations are reached.
5. Returns the optimized initial velocities (`vx0`, `vy0`).

```
def shoot_v_for_T(xs, ys, xe, ye, T, dt, g, k_m, vxg=0.0, vyg=-0.0, tol=1e-2, mi=50):
    vx0, vy0 = vxg, vyg
    h = 1e-5
    for _ in range(mi):
        xf, yf = ball_throw_trajectory(vx0, vy0, xs, ys, T, dt, g, k_m)
        fx, fy = xe - xf, ye - yf
        if abs(fx)<tol and abs(fy)<tol:
            return vx0, vy0
        xfg, yfg = ball_throw_trajectory(vx0+h, vy0, xs, ys, T, dt, g, k_m)
        xfh, yfh = ball_throw_trajectory(vx0, vy0+h, xs, ys, T, dt, g, k_m)
        fxg, fyg = xe - xfg, ye - yfg
        fxh, fyh = xe - xfh, ye - yfh
        dfxdvx, dfydvx = (fxg - fx)/h, (fyg - fy)/h
        dfxdvy, dfydvy = (fxh - fx)/h, (fyh - fy)/h
        J = np.array([[dfxdvx, dfxdvy],[dfydvx, dfydvy]])
        F = np.array([fx, fy])
        try:
            dV = np.linalg.solve(J, -F)
        except:
            break
        vx0 += dV[0]
        vy0 += dV[1]
    return vx0, vy0
```

▼ Mathematical

Error Function: Let the current position of the projectile at time T be (x_f, y_f), computed using the `ball_throw_trajectory` function. Error between the target and the computed position is:

$$f_x = x_e - x_f, \quad f_y = y_e - y_f$$

The goal is to adjust (v_{x0}, v_{y0}) such that:

$$|f_x| < \text{tol} \quad \text{and} \quad |f_y| < \text{tol}$$

Jacobian Approximation: The Jacobian matrix $\mathbf{J}$ represents the sensitivity of the error (f_x, f_y) to small changes in the velocities (v_{x0}, v_{y0}):

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_x}{\partial v_{x0}} & \frac{\partial f_x}{\partial v_{y0}} \\ \frac{\partial f_y}{\partial v_{x0}} & \frac{\partial f_y}{\partial v_{y0}} \end{bmatrix}$$

The partial derivatives are approximated numerically using small h in the velocities:

For v_{x0} :

$$\frac{\partial f_x}{\partial v_{x0}} \approx \frac{f_x(v_{x0} + h, v_{y0}) - f_x(v_{x0}, v_{y0})}{h}$$

$$\frac{\partial f_y}{\partial v_{x0}} \approx \frac{f_y(v_{x0} + h, v_{y0}) - f_y(v_{x0}, v_{y0})}{h}$$

For v_{y0} :

$$\frac{\partial f_x}{\partial v_{y0}} \approx \frac{f_x(v_{x0}, v_{y0} + h) - f_x(v_{x0}, v_{y0})}{h}$$

$$\frac{\partial f_y}{\partial v_{y0}} \approx \frac{f_y(v_{x0}, v_{y0} + h) - f_y(v_{x0}, v_{y0})}{h}$$

Newton-Raphson Update: To reduce the error, solve the linear system:

$$\mathbf{J} \cdot \Delta \mathbf{v} = -\mathbf{F}$$

Where:

$$\mathbf{F} = \begin{bmatrix} f_x \\ f_y \end{bmatrix}, \quad \Delta \mathbf{v} = \begin{bmatrix} \Delta v_{x0} \\ \Delta v_{y0} \end{bmatrix}$$

Update the velocities:

$$v_{x0} \leftarrow v_{x0} + \Delta v_{x0}$$

$$v_{y0} \leftarrow v_{y0} + \Delta v_{y0}$$

Convergence Check: If the error $|f_x| < \text{tol} \quad \text{and} \quad |f_y| < \text{tol}$, return the current values of (v_{x0}, v_{y0}) .

Iteration Limit: Stop if the maximum number of iterations mi is reached.

Simulation

This code simulates and visualizes the trajectory of two objects: one moving based on initial observed conditions and a second "shooter" object trying to intercept the first at a specific target time T .

Checking if velocity and position data exist

If velocity (`vels_x`, `vels_y`) and position (`positions`) data exist:

- `x0, y0`: Initial position of the object.
- `tx, ty`: Final (last observed) position of the object.
- `vx0, vy0`: Initial velocities.
- Use the `shooting_method` to estimate: gravity (`g`), drag coefficient (`km`).

If data does not exist: set `g` and `km` to zero.

```
if vels_x and vels_y:
    dt = 1/fps
    x0, y0 = positions[0]
    tx, ty = positions[-1]
    vx0, vy0 = vels_x[0], vels_y[0]
    g, km = shooting_method(vx0, vy0, x0, y0, tx, ty, dt)
else:
    g, km = 0, 0
```

Predict future positions for the first object

If position and velocity data are available:

1. Start with the last known position (`px`, `py`) and velocities (`vxe`, `vye`).
2. Simulate 50 future positions using ball motion equations:
    - Compute accelerations (`ax`, `ay`) from drag and gravity.
    - Update velocities (`vxe`, `vye`) and positions (`px`, `py`) based on accelerations and time step `dt`.
    - Append the new positions to `ex_pos`.
3. Combine observed (`pos_arr`) and future (`ex_arr`) positions into `full_trajectory`.

If no data exists: set `full_trajectory` to an empty array.

```
if positions and vels_x and vels_y:
    vxe, vye = vels_x[-1], vels_y[-1]
    px, py = positions[-1]
    ex_pos = []
    for _ in range(50):
        ax = -km * vxe * np.sqrt(vxe*vxe + vye*vye)
        ay = -g - km * vye * np.sqrt(vxe*vxe + vye*vye)
        vxe += ax * dt
        vye += ay * dt
        px += vxe * dt
        py += vye * dt
        ex_pos.append((px, py))
    ex_arr = np.array(ex_pos)
    pos_arr = np.array(positions)
    full_trajectory = np.concatenate((pos_arr, ex_arr), axis=0)
else:
    full_trajectory = np.array([])
```

Determine the target position at time T

Using `T` (3.5 seconds):

- Calculate the index of the trajectory (`idxT`) corresponding to this time.
- If the index is within bounds, retrieve the position (`xt`, `yt`) at T.
- Otherwise, use the last point in the trajectory as the target.

```
T = 3.5
xt, yt = None, None
if len(full_trajectory) > 0:
    idxT = int(T/dt)
    if idxT < len(full_trajectory):
        xt, yt = full_trajectory[idxT]
    else:
        xt, yt = full_trajectory[-1]
```

Compute shooter's trajectory to intercept the target

If the target position (`xt`, `yt`) exists:

1. Define initial shooter parameters:
    - Starting position: `(900, 400)`.
    - Initial guess for velocities: `gvx, gvy = 0.0, 0.0`.
2. Use `shoot_v_for_T` to calculate the shooter's initial velocities (`ivx`, `ivy`) that allow it to hit the target at T.
3. Simulate the shooter's motion for T:
    - Compute accelerations and update velocities and positions similarly to the first object.
    - Append positions to `sp` and store them in `second_arr`.

```
second_arr = np.array([])
if xt is not None and yt is not None:
      shooter_x, shooter_y = 900, 400
    gvx, gvy = 0.0, 0.0
    ivx, ivy = shoot_v_for_T(shooter_x, shooter_y, xt, yt, T, dt, g, km, gvx, gvy)
    vx2, vy2 = ivx, ivy
    x2, y2 = shooter_x, shooter_y
    sp = []
    stp = int(round(T/dt))
    for _ in range(stp):
        ss2 = np.sqrt(vx2*vx2 + vy2*vy2)
        ax2 = -km * vx2 * ss2
        ay2 = -g - km * vy2 * ss2
        vx2 += ax2*dt
        vy2 += ay2*dt
        x2 += vx2*dt
        y2 += vy2*dt
        sp.append((x2, y2))
    second_arr = np.array(sp)
```

Visualization

The figure shows:

- **Red line (`line1`)**: Trajectory of the first object.
- **Green line (`line2`)**: Trajectory of the shooter.
- **Magenta cross (`pointT`)**: Target position.

Axes limits are set based on the full trajectory.

```
fig, ax = plt.subplots()
ax.invert_yaxis()
line1, = ax.plot([], [], 'b-o')
line2, = ax.plot([], [],  marker='o', color='purple')
```

```
pointT, = ax.plot([], [], marker = 'o', color='black', markersize=10)
if len(full_trajectory):
    ax.set_xlim(full_trajectory[:,0].min()-50, full_trajectory[:,0].max()+50)
    ax.set_ylim(full_trajectory[:,1].max()+50, full_trajectory[:,1].min()-50)
```

Animation initialization

Clears all trajectory lines and sets the target point.

```
def init():
    line1.set_data([], [])
    line2.set_data([], [])
    pointT.set_data(txl, tyl)
    return line1, line2, pointT
```

Animation update function

Updates:

- First object's trajectory (`cx`, `cy`) and shooter's trajectory (`cx2`, `cy2`).
- Redraws the updated trajectories.

```
def update(frame):
    if frame < len(full_trajectory):
        cx.append(full_trajectory[frame,0])
        cy.append(full_trajectory[frame,1])
    if frame < len(second_arr):
        cx2.append(second_arr[frame,0])
        cy2.append(second_arr[frame,1])
    else:
        plt.close(fig)
    line1.set_data(cx, cy)
    line2.set_data(cx2, cy2)
    return line1, line2, pointT
```

Run the animation

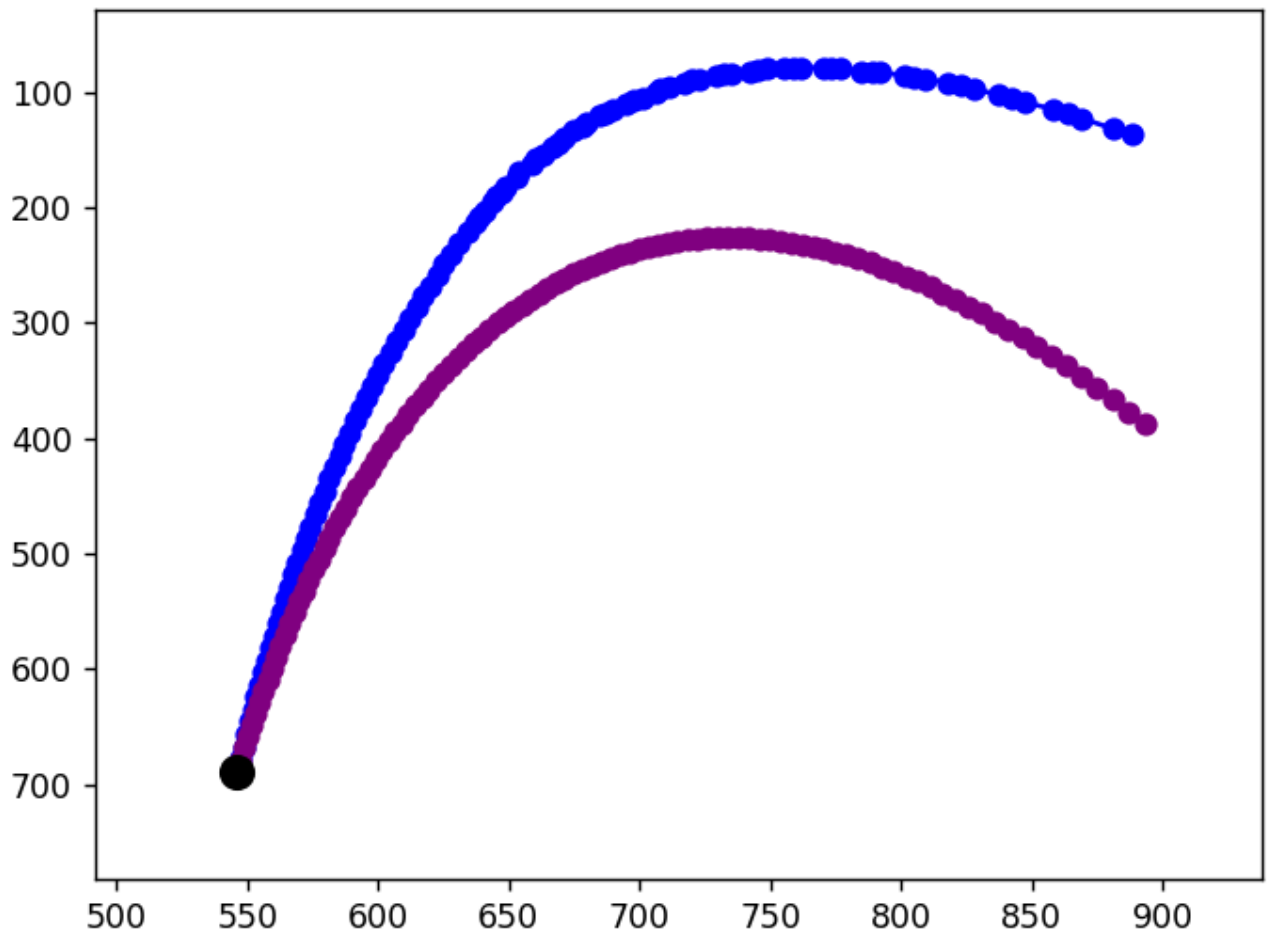
Uses `FuncAnimation` to animate the trajectories. `frames` determines the total number of animation frames, based on the longest trajectory.

```
ani = FuncAnimation(fig, update, frames=frames, init_func=init, interval=40, blit=True)
plt.show()
```

Result

This program simulates the motion of a moving object and predicts its future trajectory, calculates the initial velocity required for a "shooter" object to intercept the moving object at a specific target point and time, visualizes the trajectories of both objects and their interception in an animated plot.

Key physics concepts like gravity, drag, and numerical optimization are used to model realistic motion and solve the interception problem.



Tests

Successful Test

When the video is not too fast, the camera is not moving, background is not complicated, if it correctly catches clusters, predicting the trajectory and shooting the ball should be precise.

https://prod-files-secure.s3.us-west-2.amazonaws.com/41854c8f-3c76-420c-af1f-4b023f8e92f9/f9b74d4b-0ff2-4388-804b-e1bc7726e26a/p2.mp4

Unsuccessful Test

Here for example the video is too small it is 1 second long, and `cv2` can not cluster the ball correctly in frames, sometimes it can't catch it, although the background is not too complicated, it this is enough to make the code not work.

Limitations

- complicated background
- video too short
- moving camera
- ball not thrown correctly (it is thrown towards the camera)

https://prod-files-secure.s3.us-west-2.amazonaws.com/41854c8f-3c76-420c-af1f-4b023f8e92f9/495adc9e-b88e-4a97-aa52-da5b99c140cc/project2.mp4

Conclusion

This code provides a comprehensive simulation for tracking and predicting the motion of a ball and then intercepting it with another ball using video input, Ball motion ODE, and numerical methods. By leveraging background subtraction, contour detection, and trajectory estimation, it extracts the ball's initial position and velocity from a video feed. The shooting method and motion equations calculate the effects of gravity and air resistance on the ball's trajectory, enabling the prediction of its path. Furthermore, it implements an inverse problem solution to simulate the trajectory of a second projectile intended to intersect the predicted motion. The animation visually represents the observed and computed trajectories, demonstrating the integration of computer vision and physics modeling to solve dynamic motion problems.