Numerical Programming

Intercept a Moving Ball

Lado 🦇 Turmanidze

January 20, 2025

Problem Statement

Intercept a Moving Ball ⚽

- ❄ Input: A part of a video of a moving ball.
- ❄ Task: Throw a ball and intercept the moving ball.
- ❄ Output: An animation corresponding to the task description.
- ❄ Test: A description of the test case.
- ❄ Methodology: Should contain the problem formulation, including equations with initial and boundary conditions, the method of solution, and the algorithm.

Tasks

- ❄ Formulate the algorithm and explain your approach in written form.
- ❄ Describe the properties of numerical methods in written form.
- ❄ Develop test cases and demonstrate the validity of your results.
- ❄ Upload all necessary files, including:
  1. Presentation file
  2. Code
  3. Test data and their description
- ❄ Using the shooting method and the ball motion equation is compulsory.

Note:
The code uses *pyray* for visualization, but if you open up IDE of your choice and let it install this library(in case you have not done it yet), it will probably install a wrong version. For the correct version, you have to install *raylib*, either with *pip3* or, in case of PyCharm, with **Python Packages** window. Also, pixels per frame units will be used for g and φ.

Libraries and User-Defined Classes

Some import will make sense as you follow the document, but most are probably obvious. As for class definitions, since we have to constantly deal with two coordinates: $(x, y), (v_x, v_y), (dv_x, dv_y)$ I have implemented a *Vector* class that has two elements, and have overloaded math operators for convenience.

Ball class is straightforward, except probably for the *deepcopy()* method, which simply ensures values of p and v are copied, instead of referencing them.

```
from typing import cast
import numpy as np
import cv2
from numpy._core.multiarray import ndarray
from sklearn.cluster import DBSCAN
from pyray import *
import copy

H: float = 0.001  # constant for numerical differentiation


class Vector:
    def __init__(self, x: float, y: float):
        self.x, self.y = x, y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, other: float):
        return Vector(self.x * other, self.y * other)

    def __rmul__(self, other: float):
        return self * other

    def __truediv__(self, other: float):
        return Vector(self.x / other, self.y / other)


class Ball:
    def __init__(self, p: Vector, v: Vector, phi, g):
        self.p, self.v = copy.deepcopy(p), copy.deepcopy(v)
        self.phi, self.g = phi, g
```

Background Subtraction

```python
def read_data_points(file_path: str) -> tuple[list[Vector], float, tuple[int, int]]:
    cap = cv2.VideoCapture(file_path)
    dims = None
    count = 0
    acc = None
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        dims = frame.shape
        frame = np.array(frame).astype(np.float32)
        if acc is None:
            acc = frame
        else:
            x = [acc, frame]
            acc = np.sum(np.array(x), axis=0)
        count += 1

    cap.release()

    bg = cast(ndarray, acc) / count
    cap = cv2.VideoCapture(file_path)
    count = 0
    all_centers = []
    radius_acc = 0
    radius_count = 0
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        frame = np.array(frame).astype(np.float32)
        diff = cv2.absdiff(frame, bg)
        bw_diff = cv2.cvtColor(diff, cv2.COLOR_RGB2GRAY)
        projectile_points = np.column_stack(np.where(bw_diff > 20)[::-1])
        scanner = DBSCAN(eps=5.0)
        if len(projectile_points) == 0:
            continue
        labels = scanner.fit_predict(projectile_points)
        for label in set(labels):
            if label == -1:
                continue
            cluster = projectile_points[labels == label]
            center = np.mean(cluster, axis=0)
            radius_acc += np.sqrt(((cluster - center) ** 2).sum(axis=1).max())
            radius_count += 1
            all_centers.append(Vector(center[0], center[1]))

        count += 1

    return all_centers, radius_acc / radius_count, cast(tuple[int, int], dims)
```

The algorithm method above read frames from the video and accumulates their pixel values to compute an average frame. This average frame is used as the background. The background is computed as:

$$bg = \frac{1}{N} \sum_{i=1}^{N} F_i$$

where N is the total number of frames, and F_i is the i-th frame represented as a matrix of pixel values.

For each frame, the algorithm computes the absolute difference between the frame and the background: $\text{diff} = |F - bg|$, highlighting the regions in the frame that differ significantly from the background, isolating moving objects or changes.

The difference frame diff is converted to grayscale, and pixels with intensity values greater than a threshold (20 in this case) are extracted:

$$bw\_diff = \text{Threshold}(diff, 20)$$

The resulting binary mask highlights areas with significant changes.

Then, *sklearn.cluster.DBSCAN* clusters those points:

$$\text{center} = \frac{1}{M} \sum_{j=1}^{M} C_j$$

$$\text{radius} = \max_j \|C_j - \text{center}\|$$

where M is the number of pixels in a cluster, and C_j represents the j-th pixel in the cluster.

Finally, the function returns:

1. The centers of all detected clusters.
2. The average radius of clusters across all frames.
3. The dimensions of the video frames.

Ball Motion ODEs

The motion of a ball can be described by the following ordinary differential equations:

$$\frac{dx}{dt} = v_x, \quad \frac{dy}{dt} = v_y, \tag{1}$$

$$\frac{dv_x}{dt} = -\frac{k}{m} v_x \sqrt{v_x^2 + v_y^2}, \tag{2}$$

$$\frac{dv_y}{dt} = -g - \frac{k}{m} v_y \sqrt{v_x^2 + v_y^2}. \tag{3}$$

The initial value conditions are given by:

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

This system of differential equations and initial value conditions must be transformed into a boundary value problem, which should then be solved using the shooting method. In the current initial value conditions, $x(0) = x_0$ and $y(0) = y_0$ represent the position of the shooter from which the ball will be launched. What I need to do is add $x(b_x) = x_r$ and $y(b_y) = y_r$, which correspond to the position of the target I am aiming at. Even though there may be many targets in the input image, I am shooting at them one by one, so I do not need to solve the BVP for all balls simultaneously.

Following methods will be of great help:

```
def acceleration(v: Vector, phi: float, g: float) -> Vector:
    mag = (v.x * v.x + v.y * v.y) ** 0.5
    return Vector(-v.x * phi * mag, g - v.y * phi * mag)


def rk4_step(b: Ball, dt: float):
    k1_p = b.v
    k1_v = acceleration(b.v, b.phi, b.g)
    k2_p = b.v + k1_v * (dt / 2)
    k2_v = acceleration(b.v + k1_v * (dt / 2), b.phi, b.g)
    k3_p = b.v + k2_v * (dt / 2)
    k3_v = acceleration(b.v + k2_v * (dt / 2), b.phi, b.g)
    k4_p = b.v + k3_v * dt
    k4_v = acceleration(b.v + k3_v * dt, b.phi, b.g)
    b.p += (k1_p + 2 * k2_p + 2 * k3_p + k4_p) * (dt / 6)
    b.v += (k1_v + 2 * k2_v + 2 * k3_v + k4_v) * (dt / 6)


def heun_rk2_step(b: Ball, dt: float):
    k1_p = b.v
    k1_v = acceleration(b.v, b.phi, b.g)
    k2_p = b.v + k1_v * dt
    k2_v = acceleration(b.v + k1_v * dt, b.phi, b.g)
    b.p += (k1_p + k2_p) * (dt / 2)
    b.v += (k1_v + k2_v) * (dt / 2)
```

As the tests showed, for this particular task, both methods seem to produce visually the same results, which brings us to their reliability:

Precision and A-Stability

RK4 is $O(\triangle t^4)$, while Heun's RK2 is $O(\triangle t^2)$, but as testing will show, there is no visual difference, which might be because of the "low" accuracy needed for this task as well as size of the balls influencing the precision. According to [1], for $\triangle t \leq \min(\frac{2m}{k}, \frac{2.785m}{k}) = \frac{2m}{k}$ both Heun's RK2 and RK4 are conditionally A-stable. $dt = 1$ does more than a fine job of respecting this upper bound.

Shooting Method

Now comes the main method, THE Shooting Method:

```
def shooting_method(b0: Ball, target: Vector, dt: float, steps: int
    ) -> Vector:
    v = Vector(b0.v.x, b0.v.y)

    for sakura in range(10):
        b = Ball(b0.p, v, b0.phi, b0.g)
        bx = Ball(b0.p, v + Vector(H, 0), b0.phi, b0.g)
        by = Ball(b0.p, v + Vector(0, H), b0.phi, b0.g)

        for useless2 in range(steps):
            rk4_step(b, dt)
            rk4_step(bx, dt)
            rk4_step(by, dt)

        error = target - b.p
        j_1 = (bx.p - b.p) / H
        j_2 = (by.p - b.p) / H
        det = j_1.x * j_2.y - j_2.x * j_1.y
        if abs(det) < 1e-4:
            break
        v += Vector(
            (j_2.y * error.x - j_2.x * error.y) / det,
            (-j_1.y * error.x + j_1.x * error.y) / det
        )

    return v

```

shooting_method computes the optimal initial velocities (v_{x_0}, v_{y_0}) for a ball to hit a specified target. I used Newton-Raphson method, instead of bisection method, in two dimensions to adjust the initial velocities, minimizing the difference between the ball's simulated final position and the target.

The goal is to solve for initial velocities (v_{x_0}, v_{y_0}) such that the ball's trajectory $(x(t), y(t))$ reaches the target at (x_r, y_r).

$$\begin{cases} F_1(v_{x_0}, v_{y_0}) = x(v_{x_0}, v_{y_0}) - x_r = 0, \\ F_2(v_{x_0}, v_{y_0}) = y(v_{x_0}, v_{y_0}) - y_r = 0. \end{cases}$$

Using Newton-Raphson iteration for systems of equations, the update rule for the initial velocities is:

$$\begin{bmatrix} v_{x_0} \\ v_{y_0} \end{bmatrix} := \begin{bmatrix} v_{x_0} \\ v_{y_0} \end{bmatrix} - J^{-1} \begin{bmatrix} F_1 \\ F_2 \end{bmatrix},$$

where J is the Jacobian matrix of partial derivatives:

$$J = \begin{bmatrix} \dfrac{\partial F_1}{\partial v_{x_0}} & \dfrac{\partial F_1}{\partial v_{y_0}} \\ \dfrac{\partial F_2}{\partial v_{x_0}} & \dfrac{\partial F_2}{\partial v_{y_0}} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial x}{\partial v_{x_0}} & \dfrac{\partial x}{\partial v_{y0}} \\ \dfrac{\partial y}{\partial v_{x_0}} & \dfrac{\partial y}{\partial v_{y0}} \end{bmatrix}. \tag{4}$$

Which I will approximate using finite differences:

$$\frac{\partial x}{\partial v_{x_0}} \approx \frac{x_1 - x}{h}, \qquad \frac{\partial x}{\partial v_{y_0}} \approx \frac{x_2 - x}{h},$$
$$\frac{\partial y}{\partial v_{x_0}} \approx \frac{y_1 - y}{h}, \qquad \frac{\partial y}{\partial v_{y_0}} \approx \frac{y_2 - y}{h},$$

where x_1, y_1 and x_2, y_2 are the positions after slightly increasing v_{x_0} and v_{y_0} by h.

Given $\det(J) = j_{11}j_{22} - j_{12}j_{21}$, the Newton-Raphson update for the velocities is:

$$\Delta v_{x_0} = \frac{j_{22} \cdot e_1 - j_{12} \cdot e_2}{\det(J)},$$
$$\Delta v_{y_0} = \frac{-j_{21} \cdot e_1 + j_{11} \cdot e_2}{\det(J)},$$

where the errors are:

$$e_1 = x_r - x,$$
$$e_2 = y_r - y.$$

New velocities will be calculated as:

$$v_{x_0} := v_{x_0} + \Delta v_{x_0},$$
$$v_{y_0} := v_{y_0} + \Delta v_{y_0}.$$

This iterative process continues until the errors are sufficiently small (criterion in code: $\det(J) < 10^{-5}$) or the maximum number of iterations is reached.

The Initial Ball

To get the trajectory of the ball, we want to approximate the moving ball's position, velocity, g and $\frac{k}{m} =: \varphi$. We have to somehow calculate initial g and φ, since without knowing exact properties of the ball, we can't extract these 2 parameters from the video. For method of calculation, I simply choose average of v and dv, since g and φ can be calculated using:

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

Since ball is always supposed to follow ball motion ordinary differential equations, v_x should never equal 0, so we do not get zero division. I use central finite differences for calculating velocity and acceleration at each point in time.

```python
def get_g_and_phi(p: list[Vector]) -> tuple[float, float]:
    n = len(p)

    v = [Vector(0, 0) for _ in range(n)]
    for i in range(1, n - 1):
        v[i] = (p[i + 1] - p[i - 1]) / 2.0
    v[0] = v[1]
    v[n - 1] = v[n - 2]

    a = [Vector(0, 0) for _ in range(n)]
    for i in range(1, n - 1):
        a[i] = (v[i + 1] - v[i - 1]) / 2.0
    a[0] = a[1]
    a[n - 1] = a[n - 2]

    avg_v = v[1] + v[n - 2]
    avg_a = Vector(0, 0)
    for i in range(2, n - 2):
        avg_v += v[i]
        avg_a += a[i]
    avg_v /= (n - 2)
    avg_a /= (n - 4)

    phi = max((-1.0 / (avg_v.x * (avg_v.x ** 2 + avg_v.y ** 2) ** 0.5) * avg_a.x), 0.0)
    g = avg_a.x * avg_v.y / avg_v.x + avg_a.y

    return g, phi


def get_initial_ball(p: list[Vector]) -> Ball:
    position = p[0]
    g, phi = get_g_and_phi(p)
    velocity = shooting_method(Ball(position, Vector(0, 0), phi, g), p[-1], 1, len(p) - 1)
    return Ball(position, velocity, phi, g)
```

get_initial_ball method will return the (approximated) ball in the very first frame, where it gets detected.

Main method

```python
if __name__ == "__main__":
    points, radius, dims = read_data_points("report/video 1.mp4")
    scale = min(1.0 / dims[0], 1.0 / dims[1])
    points = [point * scale for point in points]

    ball = get_initial_ball(points)
    ball_copy = copy.deepcopy(ball)
    for i in range(200):
        rk4_step(ball_copy, 1)

    bullet_v = shooting_method(Ball(Vector(0.5, 1.0), Vector(0, 0), ball.phi, ball.g), ball_copy.p, 1, 200)
    bullet = Ball(Vector(0.5, 1.0), bullet_v, ball.phi, ball.g)
```

```
13
14      init_window(640, 480, "Intercept A Moving Ball")
15      set_target_fps(100)
16
17      while not window_should_close():
18          begin_drawing()
19          clear_background(WHITE)
20          draw_circle(int(ball.p.x * 480), int(ball.p.y * 480),
        radius, RED)
21          draw_circle(int(bullet.p.x * 480), int(bullet.p.y * 480),
        10, PURPLE)
22
23          for p in points:
24              draw_circle(int(p.x * 480), int(p.y * 480), 2, BLUE)
25          rk4_step(bullet, 1)
26          rk4_step(ball, 1)
27
28          dist = np.sqrt((bullet.p.x - ball.p.x) ** 2 + (bullet.p.y -
         ball.p.y) ** 2)
29          if dist < 1e-5:
30              print("<----- Missile hit the target! ----->")
31              break
32
33          end_drawing()
34      close_window()
35
```

This is where we put all the methods to use, read from the video, plot the trajectory of the ball and shoot it.

It begins by loading and processing a video file to detect moving objects. The function *read_data_points* reads the video, calculates a background image, and then uses background subtraction to find moving objects (balls, really) in the video. It clusters these moving points using *sklearn.cluster.DBSCAN* to group them together and calculates the average radius of the detected objects.

Next, the code scales the detected points based on the video dimensions to fit them into a smaller space for rendering. It then calculates the initial conditions for a ball's motion using a custom method, *get_initial_ball*. This method estimates the gravitational force g and air resistance φ based on the ball's path. Using the *shooting_method*, it computes the initial velocity of a ball required to hit the last detected point from the first point in the list. This is done by simulating the ball's motion and adjusting the velocity until it hits the target.

A new 'bullet' is then created with a starting position and velocity aimed at the target, based on the previously calculated velocity.

The script then sets up a window using the *pyray* library(which, as I explained at the very beginning of the document, is actually *raylib*) to display the simulation. It continuously draws the background and the moving ball and bullet. The ball and bullet are updated with the *rk4_step* method, which calculates their positions based on the forces acting on them.

The code calculates the distance between the bullet and the ball. If the distance becomes very small (indicating a hit), it prints a message and ends the

simulation.

The window updates 100 times per second (FPS) and continues running until the user closes it or the target is hit.

Figure 1: Original video's frame

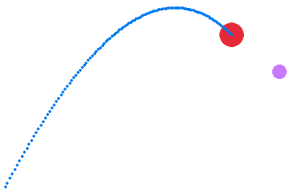

Figure 2: A frame of simulation

When does the code fail?

The code fails when the background is complicated, or when the ball is moving so fast that it is extremely hard to detect. In the plot below, one can see how fast moving ball and complex background can introduce noise in the image and make the trajectory of the ball differ significantly from the usual parabolic shape:

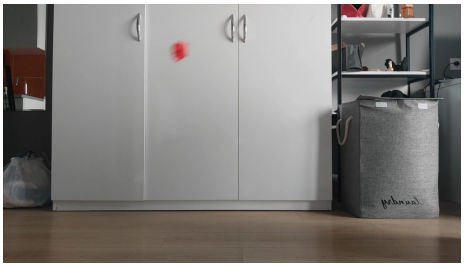

Figure 3: A frame of video that does not work

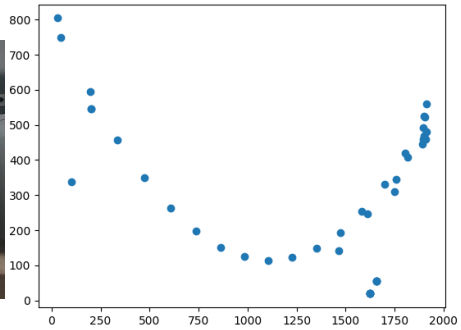

Figure 4: Plot of the points returned by *read_data_points*

References

[1] Lado Turmanidze, *Stability Wars: A-Stability in ODEs*, 2025.