Ball Detection and Trajectory Analysis

Numerical Programming Final Project

Nikoloz Rusishvili

January 20, 2025

Contents

1 Problem Statement

1.1 Overview

Given a partial video recording of a ball's projectile motion, this project aims to:

1. Detect and track the ball in the partial video footage
2. Reconstruct and predict the ball's complete trajectory from partial observations
3. Calculate and simulate an intercepting trajectory of a smaller ball
4. Generate a visualization of both the predicted and intercepting trajectories

1.2 Specific Objectives

1.2.1 Trajectory Reconstruction

- Extract ball positions from partial video footage using computer vision
- Estimate physical parameters (drag coefficient, initial velocity) from observed trajectory
- Use physical models to predict the complete trajectory beyond observed data
- Validate prediction accuracy against known physics principles

1.2.2 Interception Planning

- Calculate optimal initial position for the intercepting ball
- Determine required initial velocity for successful interception
- Ensure interception occurs at a specific time T
- Generate physically accurate simulation of both trajectories

1.3 Challenges

The project addresses several key technical challenges:

- Reliable ball detection under varying conditions
- Accurate parameter estimation from incomplete trajectory data
- Precise trajectory prediction accounting for air resistance
- Optimal interception path calculation using shooting method
- Real-time visualization of complex physical interactions

2 Mathematical Models

2.1 Physical System

The system models projectile motion with air resistance in a 2D plane.

2.2 Governing Equations

The motion is described by coupled differential equations:

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

where:

- (x, y) is the position
- (v_x, v_y) is the velocity
- k_m is the drag coefficient to mass ratio
- g is the gravitational acceleration

3 Numerical Methods

3.1 Integration Methods

3.1.1 Runge-Kutta 4th Order Method

The primary integration method uses RK4:

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{1}$$

3.1.2 Truncation Error Analysis

Local truncation error for RK4:

$$\text{LTE} = O(h^5) \tag{2}$$

Global truncation error:

$$\text{GTE} = O(h^4) \tag{3}$$

3.2 Comparison of Numerical Methods

In order to assess the accuracy and reliability of our integration methods, we compared the Runge-Kutta 4th order method (RK4) with Euler's method. Euler's method is a simpler, first-order method that approximates the solution by taking the previous value and adding the product of the step size and the derivative. However, it is well-known for its tendency to accumulate error over time, especially in problems involving nonlinear forces such as drag.

Through extensive testing and comparison, we found that RK4 significantly outperformed Euler's method in terms of accuracy. The error in Euler's method grew more rapidly as the simulation progressed, leading to less reliable results for both the trajectory and interception calculations. On the other hand, RK4 maintained a much more consistent and lower error margin throughout the simulation, making it the preferable choice for modeling projectile motion with air resistance.

We quantified the accuracy by calculating the global error for both methods across multiple test cases. RK4 consistently showed an error of order $O(h^4)$, while Euler's method exhibited a larger error of order $O(h^2)$, especially for longer time steps. This discrepancy underscores the importance of using a higher-order method like RK4 when dealing with complex physical systems that require precise numerical solutions.

3.3 Stability Analysis

The stability condition for the system is crucial to ensure accurate and reliable numerical results. In our model, stability is dictated by the drag term, which directly influences the maximum permissible step size in the numerical integration. The derived condition for stability is expressed as:

$$h \leq \frac{2.785m}{k} \tag{4}$$

Here:

- h is the step size used in the numerical method,
- m is the mass of the object,
- k is the drag coefficient.

This inequality ensures that the numerical integration method remains stable and does not introduce spurious oscillations or diverging behavior. The factor 2.785 originates from the specific form of the drag term and its interaction with the numerical scheme.

In practice, I can enforce this condition in code by calculating the maximum permissible step size h_{max} and taking the smaller of h_{max} and a desired step size h_{desired}. The corresponding Python snippet is as follows:

```python
# Parameters
m = 1.0   # Mass of the object (kg)
k = 0.1   # Drag coefficient (N s/m)
desired_h = 0.01  # Desired step size (s)

# Calculate maximum stable step size
h_max = (2.785 * m) / k  # Maximum stable step size

# Enforce stability condition
h = min(h_max, desired_h)  # Ensure step size respects stability limit

print(f"Step size used: {h:.5f} seconds")
```

Listing 1: Python Code for Enforcing Stability

Explanation of the Code

The code first defines the parameters:

- m: The mass of the object in kilograms.
- k: The drag coefficient, representing the resistance to motion.
- h_{desired}: The desired step size specified by the user or the algorithm.

It then computes h_{max}, the maximum stable step size, based on the derived formula. Finally, it selects the smaller of h_{max} and h_{desired} to ensure stability. This approach guarantees that the chosen step size does not violate the stability condition, maintaining the numerical accuracy and robustness of the simulation.

4 Shooting Method Implementation

4.1 Problem Formulation

The shooting method solves for initial velocities (v_{x0}, v_{y0}) to hit a target (x_t, y_t) at time T:

$$\mathbf{F}(v_{x0}, v_{y0}) = \begin{bmatrix} x(T) - x_t \\ y(T) - y_t \end{bmatrix} = \mathbf{0} \tag{5}$$

4.2 Newton Iteration

The solution uses Newton's method:

$$\begin{bmatrix} v_{x0} \\ v_{y0} \end{bmatrix}_{k+1} = \begin{bmatrix} v_{x0} \\ v_{y0} \end{bmatrix}_{k} - \mathbf{J}^{-1}\mathbf{F} \tag{6}$$

where **J** is computed using finite differences:

$$\mathbf{J} \approx \begin{bmatrix} \frac{F_x(v_{x0}+h)-F_x(v_{x0})}{h} & \frac{F_x(v_{y0}+h)-F_x(v_{y0})}{h} \\ \frac{F_y(v_{x0}+h)-F_y(v_{x0})}{h} & \frac{F_y(v_{y0}+h)-F_y(v_{y0})}{h} \end{bmatrix} \tag{7}$$

5 Algorithm

- Read the video and get the frames
- Detect the moving ball and extract its trajectory
- With gradient descent we guess initial velocities, since frame differentiation was not accurate
- With guessed initial velocities, we use shooting method to get k m ratio
- Now We have trajectory, we take any point and do the same from Task 1

6 Results Visualization

6.1 Trajectory Visualization

The system generates:

- Real-time ball tracking overlay
- Predicted trajectory curves
- Intercepting ball animation
- Extended trajectory prediction

6.2 Video Generation

Output video features:

- Original frame resolution and FPS
- Tracked ball position (red)
- Intercepting ball position (blue)
- Predicted trajectory overlay
- Time-multiplied extended simulation

7 Performance Analysis

7.1 Ball Detection Performance

- **Strengths:**
  - Robust to varying lighting conditions
  - Effective motion-based tracking
  - Adaptive search region
- **Limitations:**
  - Requires consistent ball visibility
  - Sensitive to rapid camera motion
  - May lose tracking during occlusions

7.2 Trajectory Analysis Performance

- **Strengths:**
  - Accurate physical modeling
  - Robust parameter optimization
  - Precise interception calculation
- **Limitations:**
  - Assumes constant drag coefficient
  - Limited to 2D trajectories
  - Requires good initial guesses

8 How To Run the Program.

- Download the Zip File From Teams
- Extract The Folders
- Enter Task 2 Folder inside Finals Folder
- Open main.py
- scroll down to the end, line 436
- here you can enter relative path for input video
- RUN!