Intercept a moving ball

- → 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

Solution (General):

First, we need to filter the video and extract where the ball is moving, its radius and color. To do this we use edge detection for each frame. After detecting the edges, we extract the circle's center and radius and save the coordinates this circle's center has been for each frame. After that, we must predict the ball's moving trajectory, which was not shown in the video. To achieve this, we calculate the velocity that worked on the circle for the last position and integrate trajectory based on this. Next, we show the animation of the moving ball (from the start to this trajectory movement itself). Then, we choose a random point which does not intersect or involve these ball movement (This is a start position from which we will throw a ball). Next, we start solving throwing problems. For this, we first select some predicted trajectory point (at where we want to hit a moving ball). As we know the starting point (selected random point) and end point (selected trajectory point), we create (solve) the trajectory for the moving using shooting method and other equations to guess the velocities and output the desired result. After solving and creating the trajectory, we start the second animation. Here the main ball movement is shown as it was, but additionally, a new ball is thrown which hits this ball, therefore **solving the task**.

Solution (Detailed):

- ### Video Processing

A function is used to analyze each video frame to identify a ball's presence. The frame is first converted to grayscale to simplify the image data while retaining essential visual details. This makes edge detection more efficient. Edge detection is applied to identify potential boundaries of objects within the frame. As we were told, here we could use the built-in edge detection, therefore I use cv2.canny and cv2.contour to identify the circle edges (Again, if necessary, we could use edge detection and filtering from the First Project itself). For each contour, a minimum enclosing circle is fitted to estimate the position and size of the ball. If the detected circle has a radius above a threshold (to filter out noise or irrelevant small objects), it is considered a valid detection. Once a valid circle is found, the function extracts the ball's position (center coordinates), its radius, and its center color (used later for animation). During the initial frames, the program estimates the background color too. All detected information, including ball coordinates, video dimensions, background color, radius, and ball color, is compiled for further use.

- ### Trajectory Prediction and Integration

The last moving position of a ball is taken, and velocity is calculated by backward Euler: (more precise for velocities)

$$v_x = \frac{x_n - x_{n-3}}{\Delta t}, \quad v_y = \frac{y_n - y_{n-3}}{\Delta t}$$

After that we reach to **trajectory integration steps**:

- **Equations of Motion**: The motion of a projectile, affected by both gravity and drag, is described using a set of differential equations. Equation formulas are taken from slides:

Ball Motion

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

- **4th order Runge-Kuta Method:**

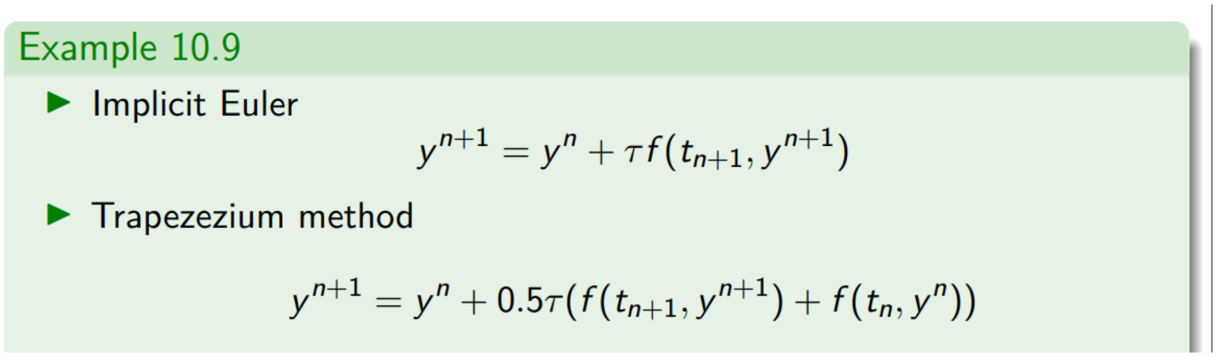
$$\begin{aligned}
k_1 &= f(t_n, y_n), \\
k_2 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right), \\
k_3 &= f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right), \\
k_4 &= f(t_n + h, y_n + hk_3).
\end{aligned}$$

$$\text{New state} = \text{current state} + \frac{dt}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$$

For comparison I have added others too:

simple Euler method (also known as the forward Euler method)

$$y_{n+1} = y_n + hf(t_n, y_n)$$


Example 10.9

- Implicit Euler
$$y^{n+1} = y^n + \tau f(t_{n+1}, y^{n+1})$$
- Trapezezium method
$$y^{n+1} = y^n + 0.5\tau(f(t_{n+1}, y^{n+1}) + f(t_n, y^n))$$

- **Trajectory Integration:**
  This function estimates the future trajectory of an object by analyzing its recent motion and setting up the initial conditions for integration. It uses backward finite differences to compute velocities and relies on calculating the final position of the projectile using Runge-Kuta and Motion equations. As for initial velocity, the velocities we calculated above are used to determine ball movement.

- **Append the Trajectory:**

  After creating the trajectory, the new coordinates of a moving ball are added to our coordinate system to display the animation.

- **Point selection**

Here by the code, we just try 1000 times to select some random point with these conditions: First, the point must not be in the moving ball itself or be around its edges at any point of the time. To ensure this, we just say that Euclidean distance between centers of the circles (random point and frame coordinate circle center) must be higher than radius sum of these 2 circles for each of them. Additionally, for visualization purposes I also add some distance between selected random point and try not to select edge of the video by Margin

- **Target Point**

After determining the start point of the Throwing Ball, we must determine where it should hit the moving ball. To achieve this, we select a random point in a newly generated trajectory (we could select any point of a moving ball, but the predicted trajectory point is visually more appealing).

- ## Shooting Method for Initial Velocity:

The goal is to hit a specific target (Target_Point) at a known position. To determine the initial velocity required to reach that target, the shooting method is used. The approach involves:

- Making an initial guess for the required velocity to reach the target. (Our initial guesses are always 0 for precise calculations)
- Integrating the trajectory as discussed above (for time parameter, it is the same as the time the main ball requires to reach the target point)
- Comparing the computed final position with the target position
- Adjusting the initial velocity to minimize the difference between the computed final position and the target
- The process is repeated iteratively until the trajectory reaches the desired target point within a predefined tolerance

For root finding, I have implemented 2 cases (scipy built-in root finding method and manual root finding with Newton-Raphson with finite differences) as in previous project:

1. Residual:

$$\mathbf{r}_k = \mathbf{F}(\mathbf{x}_k)$$

2. Residual Norm:

$$\|\mathbf{r}_k\| = \sqrt{\sum_{i=1}^{n} r_{k,i}^2}$$

3. Jacobian Approximation (Finite Differences):

$$\mathbf{J}_k[:, j] = \frac{\mathbf{F}(\mathbf{x}_k + \epsilon \mathbf{e}_j) - \mathbf{r}_k}{\epsilon}$$

4. Newton Step:

$$\mathbf{J}_k \Delta \mathbf{x}_k = -\mathbf{r}_k$$

5. Update:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta \mathbf{x}_k$$

• Animation

After calculating everything above, we start the animation. This consists of the two parts:

1. Animation of The Original Video (With Predicted Trajectory):

To do this, we create our own ball with given parameters (radius, color...) to match the video in FuncAnimation function and update each frame step by step:

a. Initialize the ball position
b. Update the position of a moving circle along the current trajectory
c. Display the predicted trajectory itself

This animation helps us analyze what our code does and makes it easier to compare to the Original Video

2. Animation With Ball Interception:

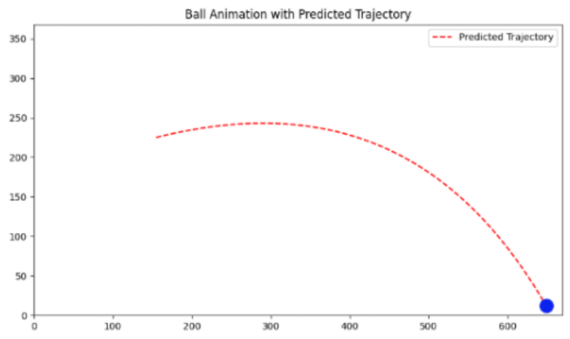
Here we animate using FuncAnimation again but with different steps:

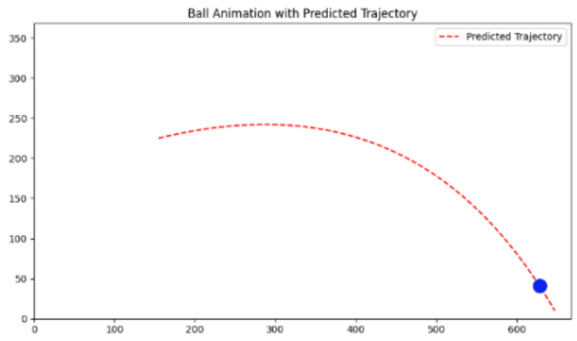
a. Initialize both main ball and thrown ball positions with their specified colors and radiuses (I consider that thrown ball has same radius as main one)
b. Scale factor the frames to match the animation (as animation is done in FuncAnimation, intervals such as fps (in here we consider 60) must be adjusted for animation to be precise and correct
c. Update the position of a moving point along the current trajectory
d. Check for a collision between the moving ball and thrown ball in the animation space. (If Euclidean distance is smaller than radiuses, then it means that the following circles hit each other).
e. Stop the animation and remove the circles for smooth **task ending**

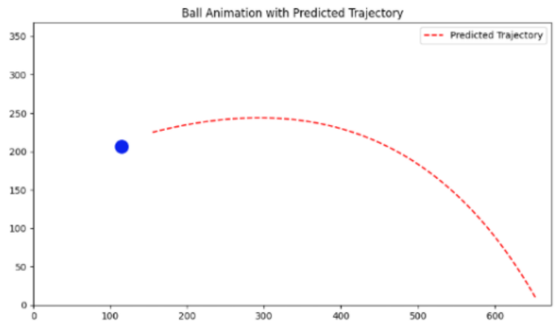
(to open second animation, you must wait at least 5 seconds before closing the first one)

Test Cases

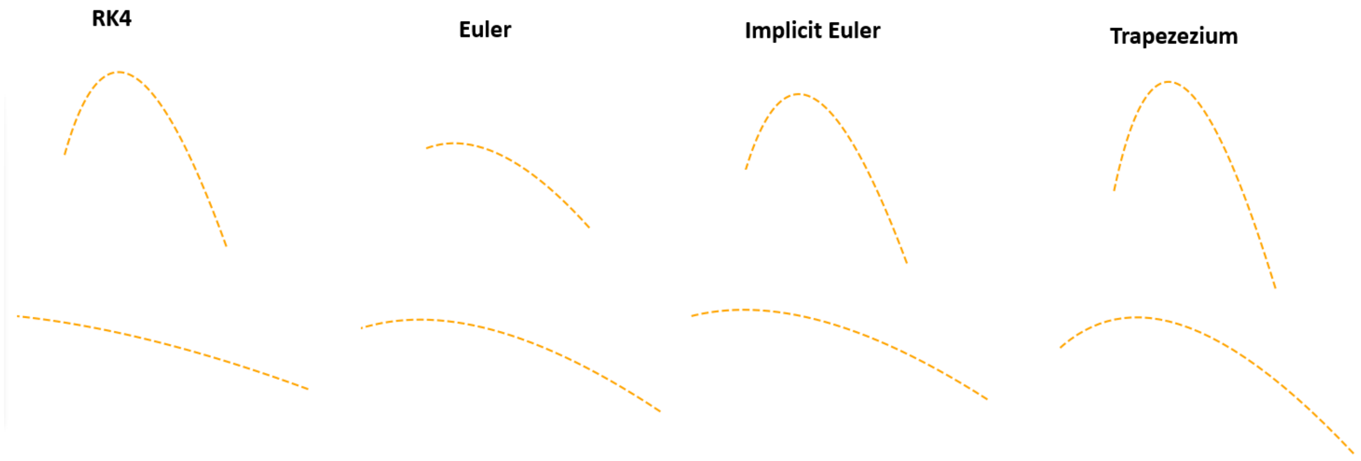
As we were instructed, we would have plain backgrounds with a single moving ball inside. Therefore, these are the test cases I used for this project are simple, I will firstly just compare different ODE solutions for comparison:

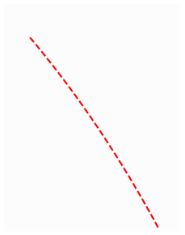


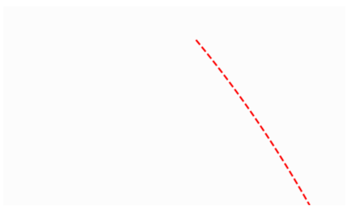
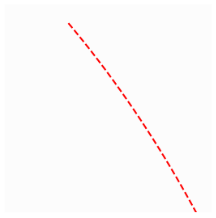




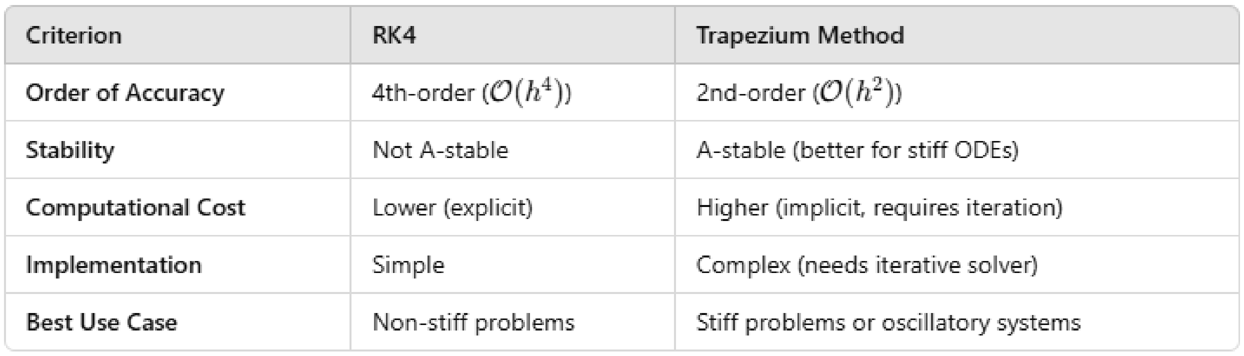
Trajectories for Thrown balls for different Solutions





After testing every case I can say that RK4 and Trapezium Methods were the closest to the real solution in most cases, so we must choose between these 2 cases which solution will be better: (Table of analysis)

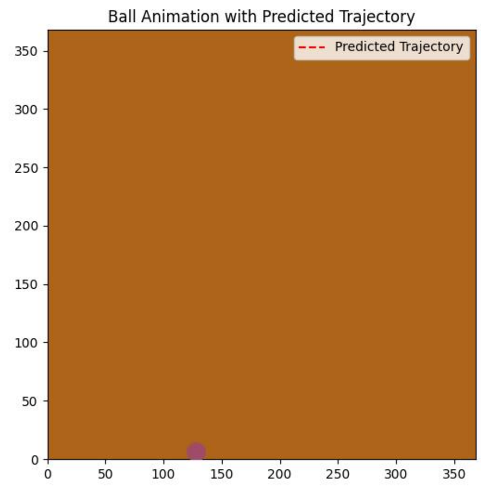


Criterion	RK4	Trapezium Method
Order of Accuracy	4th-order ($\mathcal{O}(h^4)$)	2nd-order ($\mathcal{O}(h^2)$)
Stability	Not A-stable	A-stable (better for stiff ODEs)
Computational Cost	Lower (explicit)	Higher (implicit, requires iteration)
Implementation	Simple	Complex (needs iterative solver)
Best Use Case	Non-stiff problems	Stiff problems or oscillatory systems

In test cases where calculation is easier, RK4 outperforms the Trapezium method, because the implementation of it is quite simple and does not require scipy.root solutions, therefore making it preferable in these cases. However, in cases such where coordinate size is too large, and velocities are hard to calculate, etc… Trapezium Method thrives. In overall, all these methods are fast for solving the videos where circle movement and extraction is not complex.

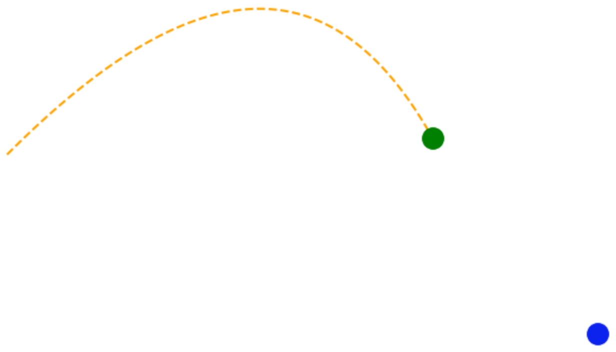
Inconsistencies:

Again, in this code, the same as in the previous project, videos with complex background fail to work:

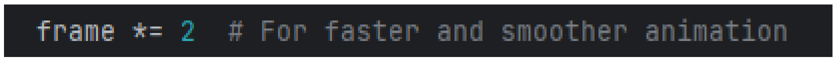


As you can see, the code fails.

Inconsistency #2:



Sometimes, the thrown ball does not hit the main ball. This issue occurs because the last position of the thrown ball is not registered (you see we scale down the animation of thrown ball to match the main ball in the code, and such values like:



```
frame *= 2  # For faster and smoother animation
```

It really affects the hitbox (the last position of trajectory is skipped and that also causes inaccuracy), but removing this code will generate the correct output (but animation will be slow). This issue occurs very rarely. (Mostly when interval in FuncAnimation is modified).

Modifying parameters result:

mass – changing its values outputs the velocities as they should be (for example if ball is lighter, then animation is fast, and if its heavy, its slow, but you can also see how gravity works well when ball goes down)

Time – changing t_0 and t_{end} values affect the ball movement speed. The less time value, the faster the animation, the more - the slower. These two parameters affect the hit process because 2 balls are moving at the same time, so their parameters to the time must be same for the solution to be correct.

Drag Coefficient – As expected, drag coefficient (Air resistance) works contrary to the mass (higher value tends to make ball float when it is going down)

Time step – works same as a time, slows, or makes animation faster (but again, it must stay same for calculations, or code will fail)

Tolerance for Calculations – tolerance does not matter that much, but if it is bigger than 10^{-1}, shooting method does not converge

Video – everything in a video affects calculations undoubtedly.

ODE solution method choices – affects speed and output of calculations (curves)

Interval in FuncAnimation – as mentioned above, FPS plays huge role in animation consistency, therefore modifying this leads to issues. For correctness, it would be better if it stayed as it is (60 fps)

Stability Result:

As described above, we can say that changing parameters affect the output, so let us discuss each step:

- **The trajectory integration method** does not change the result that much (of course if we do not give unrealistic parameters as mass to be negative, etc...), thus, making trajectory calculations **mostly stable.** And based on the ODE solution we choose, even though output is mostly correct, it does not change much. Additionally, based on the chosen method, integration might be **A-stable or not.**
- **Circle calculations, Video processing and Extraction**, as described above, there are many issues, but in most cases, output is correct, making it **half-stable**. It is mostly based on a video.
- **Animation** in every case where circles are correctly identified, and trajectories are calculated, and parameters create the desired trajectory - is always correct, making it **stable** (here I imply animation itself when everything other is stable)

Conclusion

Based on everything written above, we can conclude that code works well if the edge detection and ball extraction are done correctly and performs accordingly to the parameters we give to methods, thus making the animation correct and enjoyable to watch.