Hit a ball to fixed target

→ Input: Image of randomly scattered balls.
→ Task: Throw ball and hit balls on the image one after another
→ 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 image and indicate where circles are placed. To do this we use edge detection. After detecting the edges, we must identify which edges belong to which circle, where the circle's center is and find out its corresponding radius and color to match the image. To solve this problem, we cluster the points of edges in such a way that each cluster corresponds to each circle. Then we need to identify the radius and center coordinates of the circles and choose color of a center point of circle. After this, we choose a random point which does not intersect or involve these circles (This is a start position from which we will throw a ball). Next, we start solving throwing problems. As we know the starting point (selected random point) and end point (the center of the circle), we create (solve) the trajectories for each ball using shooting method and other equations to guess the velocities and output the desired result. After solving and creating trajectories for each circle, we start the animation itself. Here we try to be as realistic as possible. For example, if a thrown ball hits the circle which is not our trajectory target, then this circle will be removed, and the ball will be thrown again from selected start point. This process goes on until all circles are hit, thereby finishing the task it was created for.

Solution (Detailed):

- ## Edge detection

The provided algorithm performs edge detection using the **Sobel operator** to compute gradients in the image. Here is a step-by-step explanation: Firstly, we convert our image into grayscale, represented as a 2D array of intensity values and give some threshold value to decide if a gradient magnitude qualifies as an edge. For this, we need **Gradient Calculation**: The **Sobel operator** is used to calculate the gradient: Measure the rate of intensity change in the horizontal and vertical direction. Gradients are calculated using convolution with predefined Sobel kernels provided by these formulas:

Discrete convolution, Sobel, Prewitt, Scharr

▶ Sobel matrix

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

After calculating with Sobel operator for both directions, we need to find out magnitude:

$$\text{Magnitude} = \sqrt{g_x^2 + g_y^2}$$

If the magnitude exceeds the threshold, the pixel is marked as an edge (value set to 255, white). Otherwise, it remains as a non-edge (value set to 0, black). **Output**: A binary edge-detected image where edges are white (255), and the rest are black (0).

- ## Clustering

Clustering is essential for grouping the edge points that are part of the same circle. For this code I have provided 3 implementations (1 built-in, 2 manual):

1. Depth First Search

Here we perform clustering of connected components from an edge-detected image. Connected Component Detection uses Depth First Search (DFS) to identify connected components in the binary edge-detected image. The image is traversed pixel-by-pixel, and each unvisited edge pixel serves as a seed to grow a connected component. DFS is based on neighborhood (top, bottom, left, right).

2. Density-Based Spatial Clustering

DBSCAN iterates over all the points and checks whether each point is already visited or not. If not visited, it finds the point's neighbors. If the neighbors are sufficient to form a cluster, it expands the cluster; otherwise, it marks the point as noise. function starts with a core point and attempts to expand the cluster by exploring its neighbors. If a point has enough neighbors (min_samples), it is considered a core point, and its neighbors are recursively added to the cluster. The get_neighbors function checks which points are within a given distance (defined by eps) of a specific point.

(I have provided 2 implementations of DBSCAN: 1 uses built-in method, the second one is manual (implemented by me))

In DBSCAN method, values like EPS and Min_Values must be modified manually (not dynamically)

- ## Filtering

Each detected component (Cluster) is then filtered based on a minimum size threshold to exclude noise or irrelevant small clusters. For each retained component we compute:

1. Centroid (center of circle): Computed as the mean position of all pixels in the component. (The centroid of a cluster is the "center" of all points in the cluster). General formula is:

$$C_x = \frac{\sum_{i=1}^{N} x_i}{N}, \quad C_y = \frac{\sum_{i=1}^{N} y_i}{N}$$

2. Radius: Estimated as the average distance of the component's pixels from its centroid. (The radius of a cluster represents the average distance of all points in the cluster from the centroid). General formula (d represents Euclidean distance):

$$R = \frac{\sum_{i=1}^{N} d((x_i, y_i), (C_x, C_y))}{N}$$

3. Color: Extracted as the RGB color of the pixel of the centroid.

- **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 circles itself or be around their edges. To ensure this, we just say that Euclidean distance between centroids 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 image by Margin

- **Trajectory Calculations**
  - **Equations of Motion**: The motion of a projectile, affected by both gravity and drag, is described using a set of differential equations. The equations account for the position and velocity in

both horizontal and vertical directions. Formula 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:**

$$k_1 = f(t_n, y_n),$$
$$k_2 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_1}{2}\right),$$
$$k_3 = f\left(t_n + \frac{h}{2}, y_n + h\frac{k_2}{2}\right),$$
$$k_4 = f(t_n + h, y_n + hk_3).$$
$$\text{New state} = \text{current state} + \frac{dt}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4)$$

- **Trajectory Simulation:** The projectile's trajectory is computed by integrating the equations of motion over a specified period. The initial conditions (such as initial position and velocity) are provided, and the simulation continues until the projectile hits the ground (i.e., when the vertical position y becomes zero). The trajectory at each timestep is stored for later analysis.

- **Shooting Method for Initial Velocity:** the goal is to hit a specific target (centroid) 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 using this initial velocity and calculating the final position of the projectile using Runge-Kuta and Motion equations
- 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 (Because scipy was used in lectures, I use that for better calculations in main code):

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

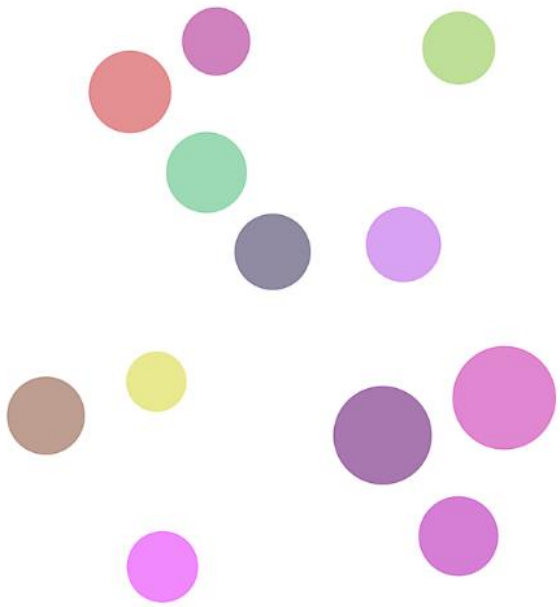
After calculating trajectories for each circle, we start The Throwing Animation. To do this, we create our own circles with given parameters to match the image in FuncAnimation function and update each frame step by step:

1. Initialize the next trajectory of the circle we throw ball at
2. Update the position of a moving point along the current trajectory
3. Check for a collision between the moving point and other targets in the animation space. (Again, if Euclidean distance is smaller than radiuses, then it means that the following circle is hit).
4. Remove the target upon collision and prepare to move to the next object. (If the hit circle is different from the trajectory circle, we try to hit, then trajectory does not change)
5. Animation continues until all circles are hit **(the task is completed)**

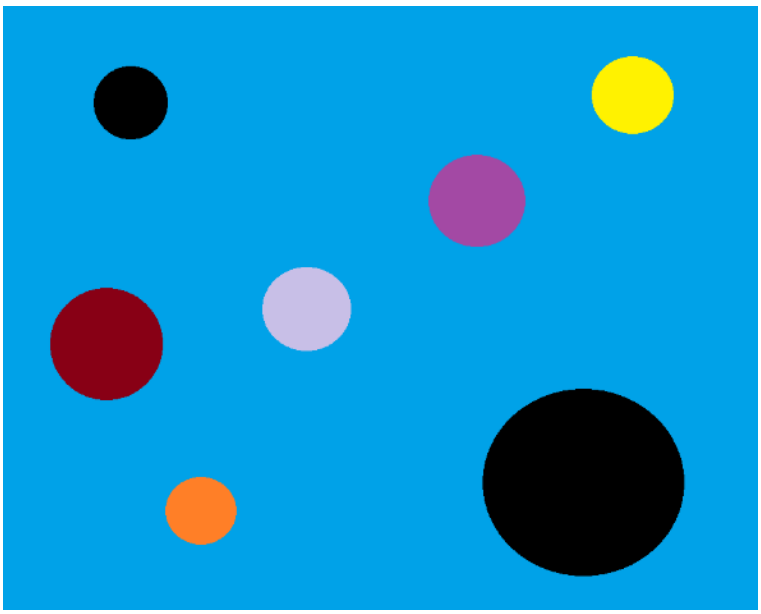
Test Cases

As we were instructed, we would have plain backgrounds with circles drawn inside. Therefore, these are the test cases I used for this project:

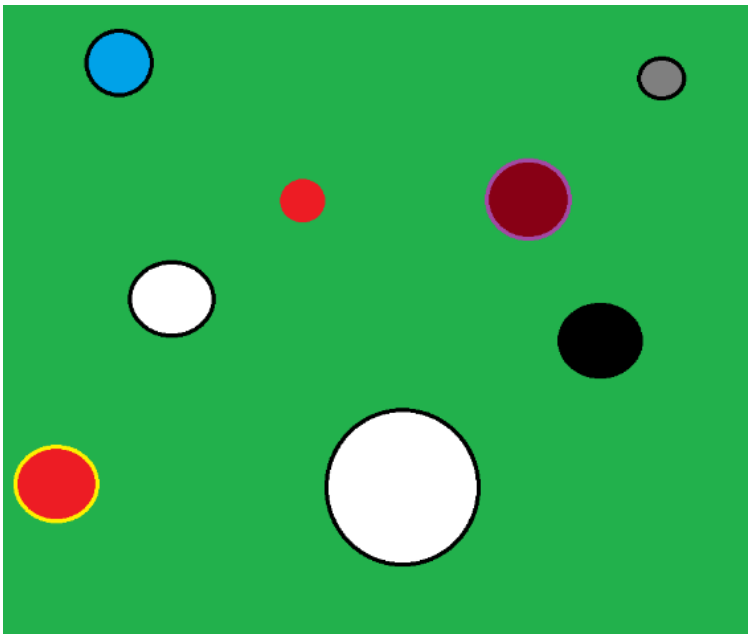
Test Case 1:



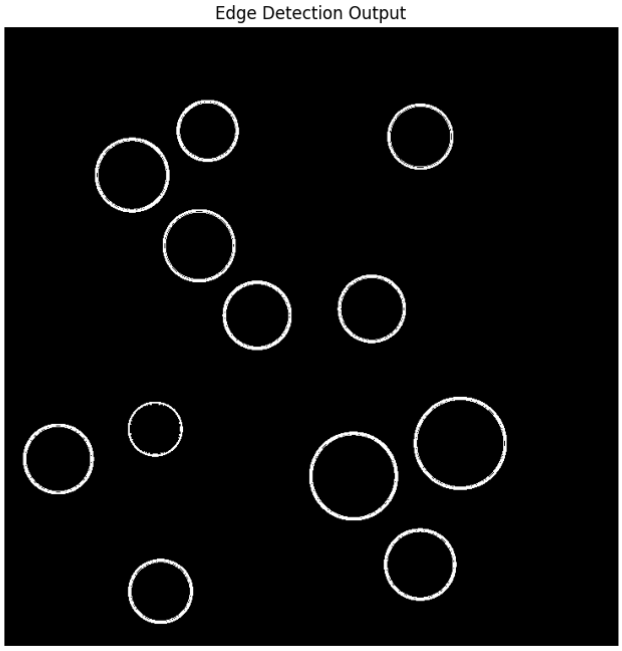
Test Case 2:

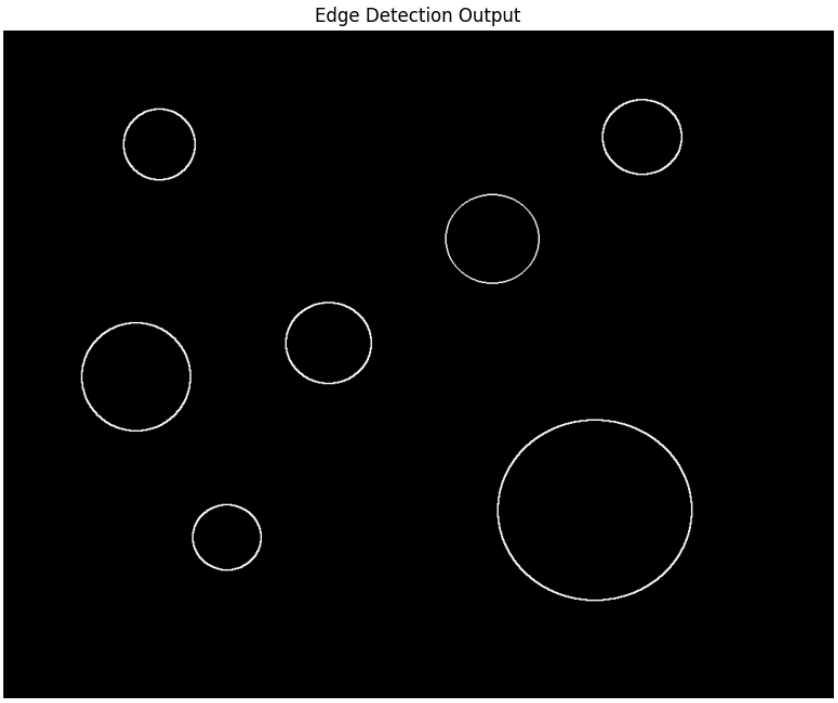


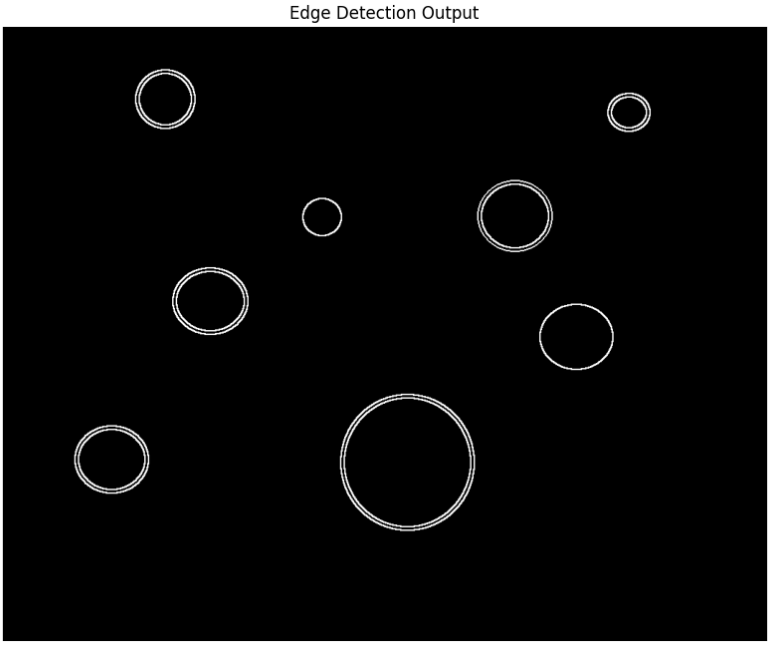
Test Case 3:



Edge detection outputs:







The main code works perfectly well for 2 test cases but fails on the third one. The issue will be discussed later, firstly we will output the time differences between the methods:

```
Execution time DFS: 1.675020 seconds
```

```
Execution time DBSCAN-Built-in: 1.753726 seconds
```

```
Execution time DFS: 2.878940 seconds
```

```
Execution time DBSCAN-Built-in: 2.859788 seconds
```

```
Execution time DFS: 2.960956 seconds
```

```
Execution time DBSCAN-Built-in: 2.999280 seconds
```

The manual (non-built-in DBSCAN) takes at least 1 minute to perform, so I did not include it here. As you can see from the outputs, the DFS method works faster than the built-in DBSCAN, but here is the issue: In test case 3, the output does not perform as intended: The outline is considered as second circle (second edge as shown in the third picture). The DFS method is unable to fix this, but with DBSCAN it's possible, we must just change eps and min_samples values manually (for this case 20, 20 worked), and the output will be correct.

```
Execution time for custom root finding: 11.627238 seconds
```

```
Execution time for scipy.root: 12.819651 seconds
```

```
Execution time for custom root finding: 9.187450 seconds
```

```
Execution time for scipy.root: 9.148329 seconds
```

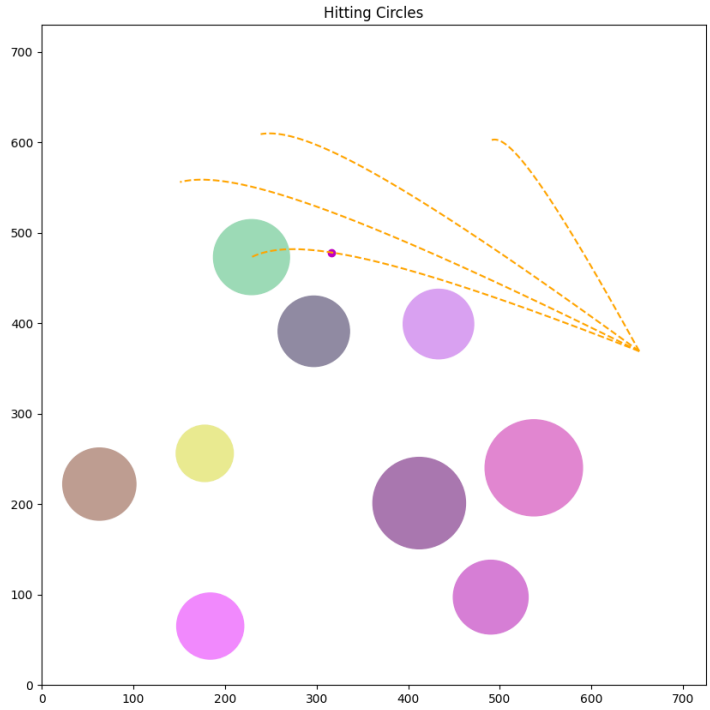
```
Execution time for custom root finding: 14.533366 seconds
```

```
Execution time for scipy.root: 16.629993 seconds
```

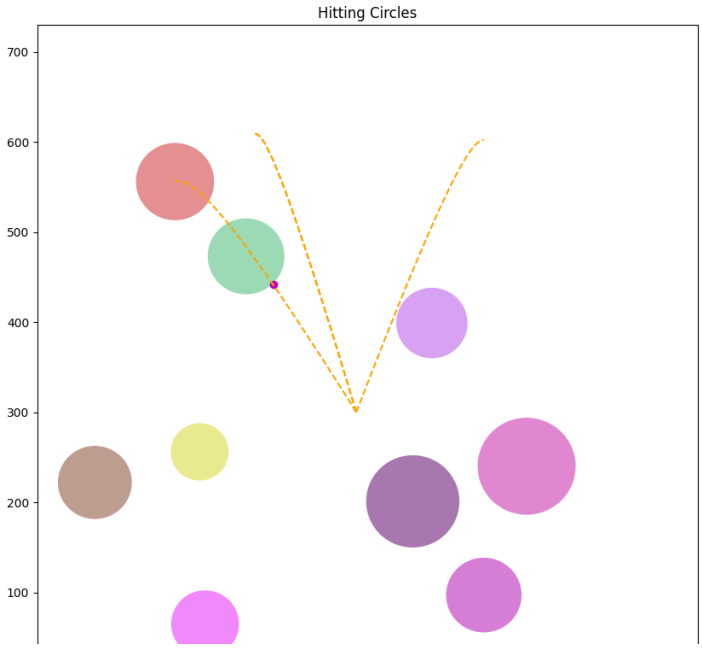
As you can see Newton-Raphson with finite differences implementation is faster than scipy.root, and the outputs are correct for both methods, but in main code I will still leave the scipy.root implementation because I think it will be more precise.

The output:

If the edge detection and clustering correctly calculate and discover circles, then the code works perfectly well on all root outputs: (videos like this)



But there are inconsistencies:

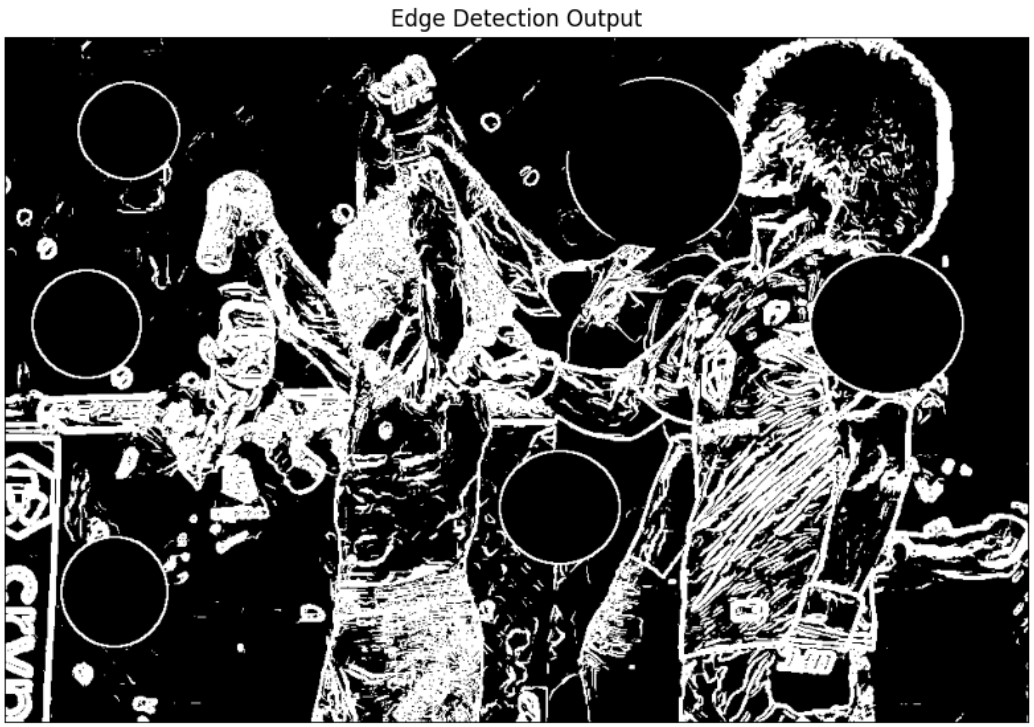


Sometimes the moving ball passes through the circle line which must register as a hit but does not. This is because in this code the hitting ball is considered as a moving point (animation just outputs a little circle). That is why the ball only hits the circle if it passes through it in trajectory line (for example green circle above). To address this issue, we can just add radius to the circle and collision detection algorithm and visually output the new moving circle.

Inconsistency #2:

Consider this test case:

All edge detection algorithms that I have failed to even work slightly correct on pictures like these. The output:



Even though edge detection detects circles, the background image plays the biggest role in edge detection, so it fails every time the background picture isn't simple plain color (white, blue, ...).

Additionally, there is inconsistency where background or circle color is assigned wrongly (For example considering half of a circle has red color, and another half has blue, (or basketball ball viewed as 2D), then color assigned will be one of the colors (which center of circle has)). Also, this code will never work on 3D balls.

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 (as we were not specified, I assume that all balls must hit at the same time (5 seconds for example) and calculations are done based on this in shooting method). The less time value, the faster the animation, the more - the slower.

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

Time step – works same as a time, slows or makes animation faster

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

Image – everything in an image affects calculations undoubtedly.

Clustering and root choices – affects speed and output of calculations

Stability Result:

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

- **Shooting method** does not change the result that much (of course if we don't give unrealistic parameters as mass to be negative, etc...), thus, making trajectory calculations **mostly stable.**
- **Circle calculations**, as described above, there are many issues, but in most cases, output is correct, making it **half-stable**.
- **Animation** in every case where circles are correctly identified, and trajectories are calculated, is always correct, making it **stable** (here I don't imply image colors etc... just the throwing animation itself)

Conclusion

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