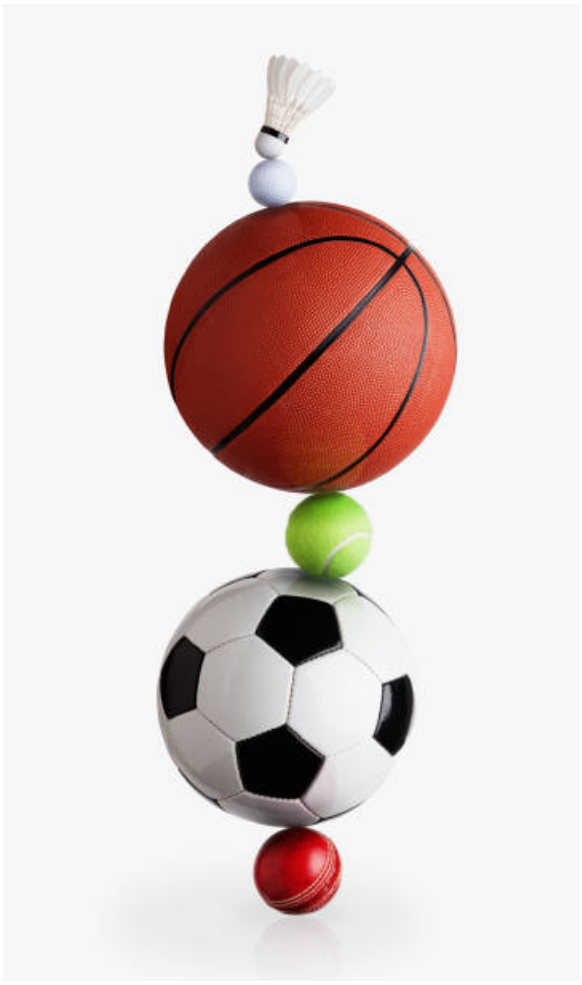
Project 1: Hit a ball to fixed target



Statement of the problem

- 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

The primary objective of the project is to develop a system that processes an input image to identify and cluster visually distinct targets, simulate projectile trajectories to hit these targets accurately, and dynamically visualize the results.

This problem involves multiple computational steps, including image processing (smoothing and edge detection), clustering of detected edges into distinct targets (DBSCAN), trajectory calculation using numerical methods (Euler and RK-2), and visualization of the shooting process. The project aims to integrate these components effectively, to shoot the targets from a single point.

Code explanation

Image Processing

▼ Convolution

This function performs the convolution operation, a fundamental technique in image processing. It takes an input image and a kernel (filter) and applies the filter over the image to produce a new image. Convolution is implemented using nested loops, where a region of the image is multiplied element-wise with the kernel and summed up to produce a single pixel value in the output image.

The padding mechanism ensures that the edges of the image are processed correctly by replicating the boundary values. This avoids issues like loss of data at the borders.

https://en.wikipedia.org/wiki/Convolution

```
def convolve(image, kernel):
    kernel_height, kernel_width = kernel.shape
    image_height, image_width = image.shape
    convolved_image = np.zeros((image_height, image_width), dtype=np.float32)
    padded_image = np.pad(image, ((1, 1), (1, 1)), mode='edge')
    for i in range(image_height):
        for j in range(image_width):
            region = padded_image[i : i + kernel_height, j : j + kernel_width]
            convolved_image[i, j] = np.sum(region * kernel)
    return convolved_image
```

▼ Gaussian blur

This function applies a Gaussian blur to the image, using a predefined 3×3 Gaussian kernel. The Gaussian blur smooths the image by reducing noise and detail. It is essential for edge detection, as it reduces the impact of noise on the Sobel operator.

The choice of a 3×3 kernel balances computational efficiency with effectiveness in noise reduction. Larger kernels would further smooth the image but increase computation time.

https://en.wikipedia.org/wiki/Gaussian_blur

```
def gaussian_blur(image):
    gaussian_kernel = np.array([
        [1/16, 2/16, 1/16],
        [2/16, 4/16, 2/16],
        [1/16, 2/16, 1/16]
    ], dtype=np.float32)
    return convolve(image, gaussian_kernel)
```

▼ Sobel edge detection

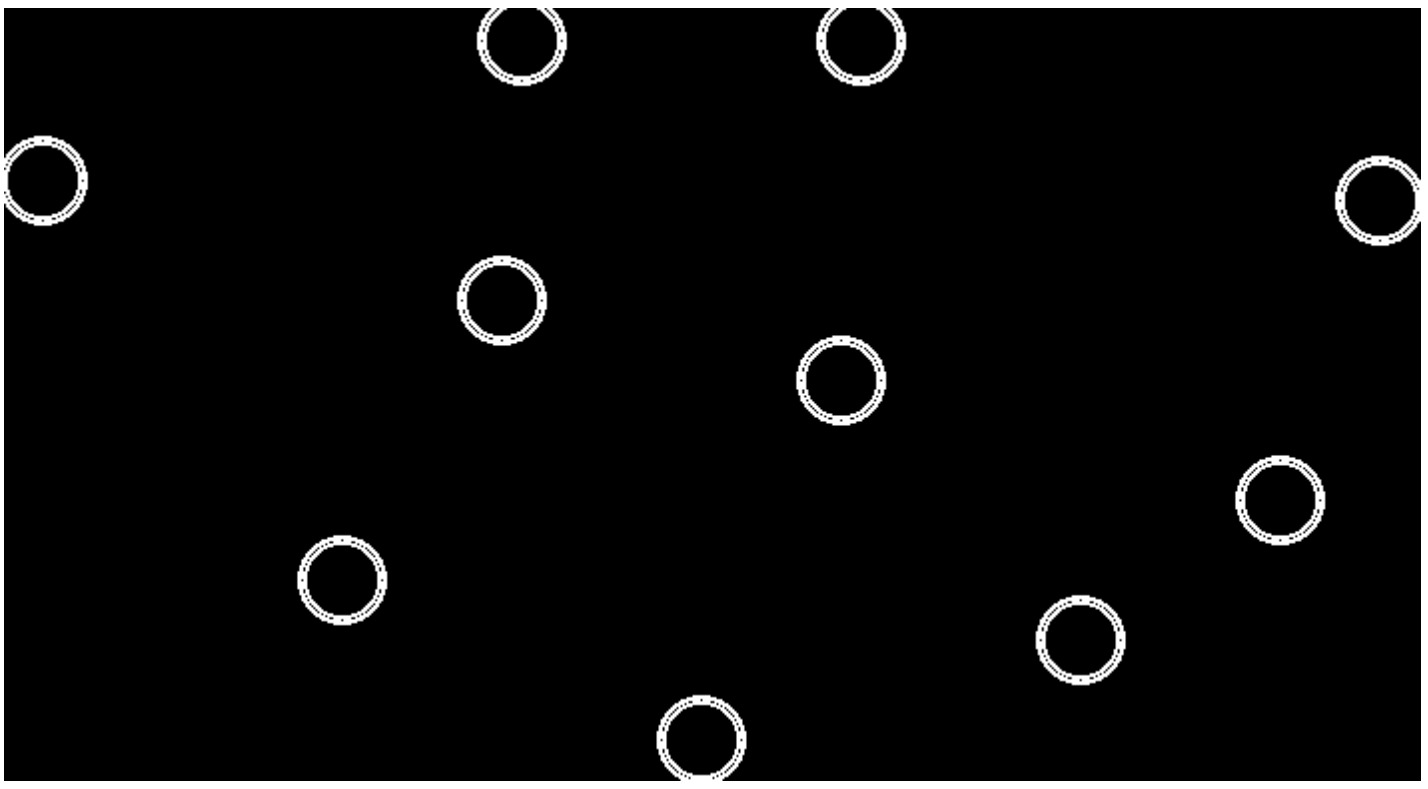
The Sobel operator is used to calculate the gradient magnitude of the image, highlighting areas of high intensity change, which correspond to edges. This function computes gradients in both the x and y directions using predefined Sobel kernels.

The resulting gradient magnitude is normalized to a range of 0-255 for better visualization and thresholded to produce a binary edge map. The threshold value of 50 determines which edges are considered significant, but this parameter could be fine-tuned for different applications.

https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm

```
def sobel_edge_detection(image, threshold=50, normalize=True, visualize=False):
    sobel_x = np.array([[-1, 0, 1],
                        [-2, 0, 2],
                        [-1, 0, 1]], dtype=np.float32)
    sobel_y = np.array([[1, 2, 1],
                        [0, 0, 0],
                        [-1, -2, -1]], dtype=np.float32)

    grad_x = convolve(image, sobel_x)
    grad_y = convolve(image, sobel_y)

    gradient_magnitude = np.sqrt(grad_x ** 2 + grad_y ** 2)
    if normalize:
        gradient_magnitude = (gradient_magnitude / gradient_magnitude.max()) * 255

    edge_image = (gradient_magnitude > threshold) * 255
    edge_image = edge_image.astype(np.uint8)

    return edge_image
```

Sobel image



▼ DBSCAN

The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm clusters the edge points detected in the previous step. DBSCAN is ideal for this application because it can identify clusters of arbitrary shape and is robust to noise.

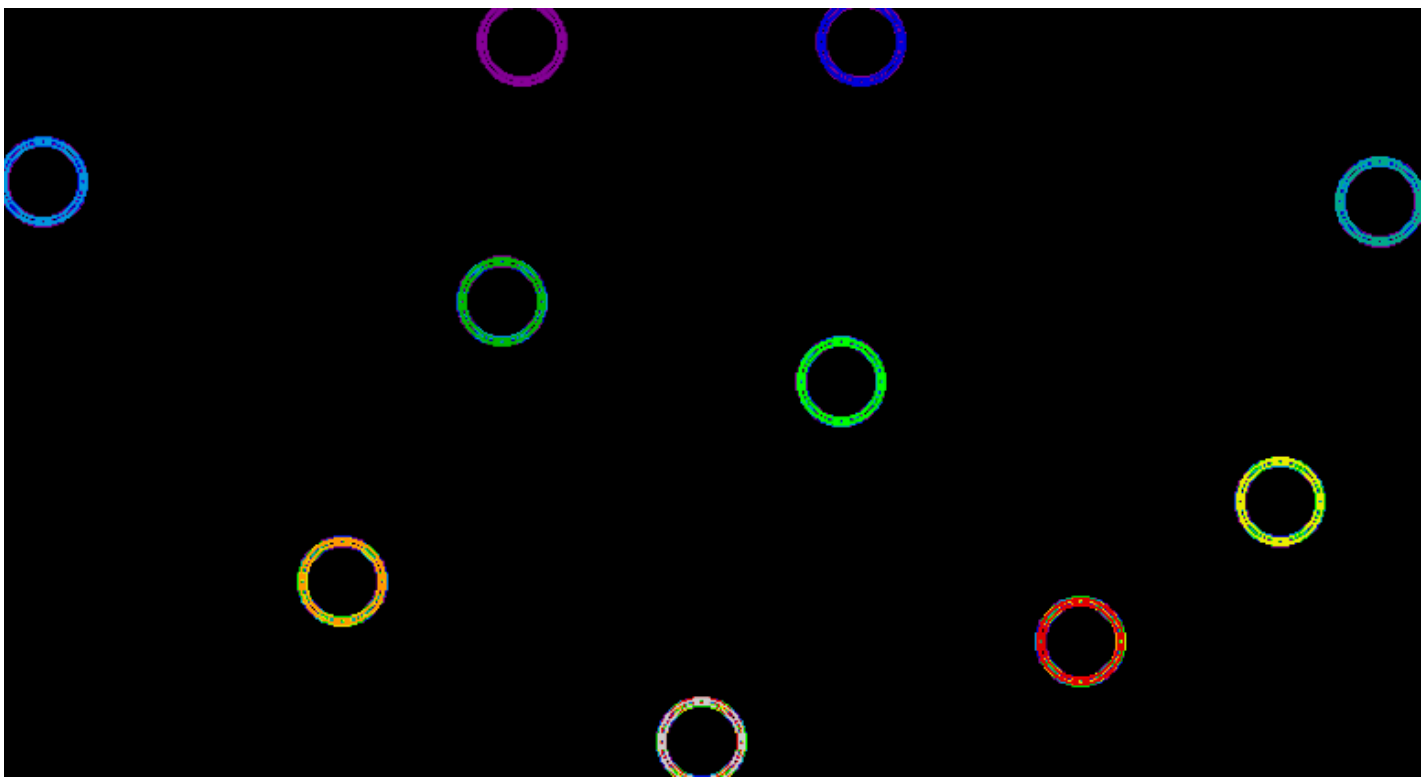
The algorithm identifies core points (points with enough neighbors within a radius eps), expands clusters from these core points, and marks points that do not belong to any cluster as noise.

Parameters eps (distance threshold) and min_samples (minimum neighbors for a core point) are crucial for the algorithm's performance. The default values work well for the given task but could be adjusted for different datasets.

https://en.wikipedia.org/wiki/DBSCAN

```
def dbscan(points, eps=2, min_samples=5):
    n_points, _ = points.shape
    labels = -2 * np.ones(n_points, dtype=int)
    cluster_id = 0
    def region_query(point_idx):
        point = points[point_idx]
        return np.where(np.linalg.norm(points - point, axis=1) < eps)[0]
    def expand_cluster(point_idx, neighbors):
        labels[point_idx] = cluster_id
        queue = deque(neighbors)
        while queue:
            neighbor_idx = queue.popleft()
            if labels[neighbor_idx] == -2:
                labels[neighbor_idx] = cluster_id
                neighbor_neighbors = region_query(neighbor_idx)
                if len(neighbor_neighbors) >= min_samples:
                    queue.extend(neighbor_neighbors)
            elif labels[neighbor_idx] == -1:
                labels[neighbor_idx] = cluster_id
    for point_idx in range(n_points):
        if labels[point_idx] != -2:
            continue
        neighbors = region_query(point_idx)
        if len(neighbors) < min_samples:
            labels[point_idx] = -1
        else:
            expand_cluster(point_idx, neighbors)
            cluster_id += 1
    return labels
```

Clustered image



▼ Centroids and radius

This function computes the smallest circle that can enclose a set of points (a cluster). The centroid of the points is used as the circle's center, and the radius is the maximum distance from the centroid to any point in the cluster.

This representation is useful for visualizing clusters and defining targets for the shooting simulation.

```
def calculate_bounding_circle(cluster_points):
    rows = cluster_points[:, 0]
    cols = cluster_points[:, 1]
    min_y, max_y = np.min(rows), np.max(rows)
    min_x, max_x = np.min(cols), np.max(cols)
    centroid_y = (min_y + max_y) // 2
    centroid_x = (min_x + max_x) // 2
    dists = np.sqrt((rows - centroid_y)**2 + (cols - centroid_x)**2)
    r = np.max(dists)
    return (centroid_x, centroid_y, r)
```

▼ Sort

Targets (bounding circles) are sorted based on their distance from the shooter’s position. This ensures that the shooter aims at the nearest target first, following a logical sequence.

The Euclidean distance is calculated for sorting, and the sorted targets are returned as a list.

```
def sort_targets(circles, shooter_point):
    sx, sy = shooter_point
    def distance(circle):
        cx, cy, _ = circle
        return math.hypot(cx - sx, cy - sy)
    return sorted(circles, key=distance)
```

Ball motion ODE

Ball Motion

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

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

- 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 dynamics

This function used Ball motion ODE and returns array of accelerations

```
def ball_dynamics(v, k_m=0.005, g=200.0):
    vx, vy = v
    speed = np.sqrt(vx*vx + vy*vy)
    ax = -k_m * vx * speed
    ay = g - k_m * vy * speed
    return np.array([ax, ay])
```

Numerical Methods

▼ Euler Method

Euler's method is used to numerically integrate the equations of motion. It updates the position and velocity of the projectile based on the current values and the computed accelerations.

While simple and computationally efficient, Euler's method introduces truncation errors proportional to the step size. A smaller step size would improve accuracy but increase computation time.

https://en.wikipedia.org/wiki/Euler_method

```
def euler_method_step(p, v, f, dt=0.01):
    p_new = p + v * dt
    v_new = v + f(v) * dt
    return p_new, v_new
```

A-Stability

The stability of a numerical integration method determines how errors evolve as time steps are increased or repeated. For the Euler method, which is conditionally stable, the time step Δt and system properties play a significant role in determining the stability. To analyze the method's stability, we examine its behavior when applied to simple linear differential equations of the form:

$$\frac{dy}{dt} = \lambda y,$$

where λ is a constant characterizing the system, and y_0 is the initial condition. The solution to this equation is: $y(t) = y_0 e^{\lambda t}$.

In Euler's method, the derivative is approximated as: $y_{n+1} = y_n + \lambda y_n \Delta t,$

leading to the recurrence relation: $y_{n+1} = (1 + \lambda \Delta t) y_n.$

This recurrence relationship is central to stability. The method is stable if the magnitude of the ratio:

$$\left| \frac{y_{n+1}}{y_n} \right| < 1,$$

as otherwise, the error grows exponentially, causing instability. By induction, the recurrence can be rewritten as: $y_n = (1 + \lambda \Delta t)^n y_0$.

Since the exact solution exhibits exponential decay when $\lambda < 0$, a stable numerical method should mimic this behavior. For Euler's method, stability requires the so-called growth factor to satisfy:

$$|1 + \lambda \Delta t| < 1.$$

For real $\lambda < 0$, this simplifies to: $-2 < \lambda \Delta t < 0 \quad \Longrightarrow \quad \Delta t < \frac{-2}{\lambda}$.

▼ Runge-Kutta (RK2)

The Runge-Kutta second-order method (RK2) is introduced as an improvement over Euler's method. It calculates two approximations for the velocity and position over a time step and averages them for greater accuracy:

RK2 reduces truncation errors compared to Euler's method, making it more stable for larger time steps.

https://en.wikipedia.org/wiki/Runge–Kutta_methods#Second-order_methods_with_two_stages

```
def runge_kutta_2_step(p, v, f, dt=0.01):
    k1_v = f(v)
    k1_p = v
    k2_v = f(v + k1_v * dt)
    k2_p = v + k1_v * dt
    v_new = v + (k1_v + k2_v) * dt / 2
    p_new = p + (k1_p + k2_p) * dt / 2
    return p_new, v_new
```

A-Stability

The RK2 method, known for its improved stability compared to the Euler method, still depends heavily on the time step Δt and the specific properties of the system being analyzed. Similar to the Euler method, we can analyze the stability of RK2 by applying it to a simple linear differential equation of the form:

$$\frac{dy}{dt} = \lambda y.$$

Using RK2, the derivative is approximated in two stages at each step n. The updates proceed as follows:

1. Compute an intermediate value (or slope estimate): $k_1 = \lambda y_n, \quad k_2 = \lambda \left(y_n + k_1 \Delta t \right).$
2. Update the solution using the average slope: $y_{n+1} = y_n + \frac{\Delta t}{2} \left(k_1 + k_2 \right).$

By substituting k_1 and k_2 into the update equation, the recurrence relation becomes:

$$y_{n+1} = y_n \left(1 + \lambda \Delta t + \frac{(\lambda \Delta t)^2}{2} \right).$$

This gives us the growth factor G as: $G = 1 + \lambda \Delta t + \frac{(\lambda \Delta t)^2}{2}$.

Introducing $x = \lambda \Delta t$, we rewrite the condition for stability as: $\left|1 + x + \frac{x^2}{2}\right| < 1.$

Solving this inequality graphically or analytically reveals that:

$$-2 < x < 0 \quad \Longrightarrow \quad \lambda \Delta t > -2.$$

Substituting λ with k_m, we observe that the RK2 method provides the same constraint on the upper bound of Δt as the Euler method for this particular problem.

Euler

- precision $O(\Delta t)$ - error proportional to the square of time step.
- Cost $O(N)$ - N is the number of steps.

RK2

- precision $O(\Delta t^2)$.
- Cost $O(2N)$ - N is the number of steps.

Although RK2 has higher cost it is more accurate.

http://ljs.academicdirect.org/A32/010_037.htm

Shooting method

`shooting_v` function calculates the initial velocity required to hit a target. It uses an iterative approach, adjusting the velocity based on the difference between the projectile's final position and the target.

The Jacobian matrix is computed to estimate how changes in velocity affect the projectile's position, enabling the iterative correction of errors.

```
def shooting_v(position, target, steps, h=0.01):
    position = position.astype(np.float64)
    target   = target.astype(np.float64)
    velocity = (target - position) / (steps * h)
    for _ in range(5):
        p  = position.copy()
        v  = velocity.copy()
        p1 = position.copy()
        v1 = velocity + np.array([h, 0.0])
        p2 = position.copy()
        v2 = velocity + np.array([0.0, h])
        for __ in range(steps):
            # p,  v  = euler_method_step(p,  v,  ball_dynamics)
            # p1, v1 = euler_method_step(p1, v1, ball_dynamics)
            # p2, v2 = euler_method_step(p2, v2, ball_dynamics)
            p,  v  = runge_kutta_2_step(p,  v,  ball_dynamics)
            p1, v1 = runge_kutta_2_step(p1, v1, ball_dynamics)
            p2, v2 = runge_kutta_2_step(p2, v2, ball_dynamics)
        dv1 = (p1 - p) / h
        dv2 = (p2 - p) / h
        jacobian = np.array([
            [dv1[0], dv2[0]],
            [dv1[1], dv2[1]]
        ])
        error = p - target
        velocity -= np.linalg.inv(jacobian) @ error
    return velocity
```

▼ Mathematical

Second-Order ODE to First-Order System

The motion of the projectile is governed by a second-order ordinary differential equation (ODE), which describes its acceleration due to forces acting on it:

$$\frac{d^2P}{dt^2} = f(v)$$

where:

- $\mathbf{p}(t) = [x(t), y(t)]$ is the position of the projectile as a function of time.
- $\mathbf{v}(t) = \frac{d\mathbf{p}}{dt} = [v_x(t), v_y(t)]$ is the velocity.
- $\mathbf{f}(\mathbf{v})$ is the force acting on the projectile, typically including gravity and drag.

To solve this second-order ODE, it is transformed into a system of first-order equations:

$$\frac{d\mathbf{p}}{dt} = \mathbf{v}, \quad \frac{d\mathbf{v}}{dt} = \mathbf{f}(\mathbf{v}).$$

To solve these equations, numerical methods like **Euler's method** or **Runge-Kutta 2nd Order (RK2)** are used to approximate the position $(\mathbf{p})$ and velocity $(\mathbf{v})$ of the projectile at discrete time steps.

Euler's Method

Euler's method calculates the next position and velocity as:

$$\mathbf{p}_{n+1} = \mathbf{p}_n + \mathbf{v}_n h, \quad \mathbf{v}_{n+1} = \mathbf{v}_n + \mathbf{f}(\mathbf{v}_n)h,$$

where h is the time step.

Runge-Kutta 2nd Order Method (RK2)

RK2 improves accuracy by estimating an intermediate velocity and position:

$$k_1 = f(\mathbf{v}_n), \quad k_2 = f(\mathbf{v}_n + k_1 h),$$

The updates are:

$$\mathbf{v}_{n+1} = \mathbf{v}_n + \frac{1}{2}(k_1 + k_2)h, \quad \mathbf{p}_{n+1} = \mathbf{p}_n + \frac{1}{2}(\mathbf{v}_n + (\mathbf{v}_n + k_1 h))h.$$

Iterative Velocity Adjustment

The `shooting_v` function adjusts the initial velocity iteratively until the projectile reaches the target. Here's the detailed process:

Initial Guess

The velocity is initialized as:

$$\mathbf{v}_0 = \frac{\mathbf{target} - \mathbf{position}_0}{\text{steps} \cdot h}.$$

Simulating the Trajectory

Using the chosen numerical method (e.g., RK2), the function simulates the projectile's motion and calculates its final position after the given number of steps.

Error Correction

The error $(\mathbf{e})$ between the final position and the target is computed as:

$$\mathbf{e} = \mathbf{p}_{\text{final}} - \mathbf{target}.$$

Jacobian Matrix

The Jacobian matrix $(\mathbf{J})$ is used to estimate how changes in velocity $(\Delta\mathbf{v})$ affect the position:

$$J = \begin{bmatrix} \frac{\partial x_{\text{final}}}{\partial v_x} & \frac{\partial x_{\text{final}}}{\partial v_y} \\ \frac{\partial y_{\text{final}}}{\partial v_x} & \frac{\partial y_{\text{final}}}{\partial v_y} \end{bmatrix}.$$

Velocity Update

The velocity is updated iteratively using the Newton-Raphson method:

$$\mathbf{v}_{\text{new}} = \mathbf{v}_{\text{current}} - J^{-1} \cdot \mathbf{e}.$$

This process is repeated until the error is minimized (close to zero).

Simulation

this is where all the methods implemented so far are called, shooter point (point where the ball is shot from), is (330, 200), then we read the image, call `gaussian_blur`, `sobel_edge_detection`, and `dbscan`. with this we have our clusters.

```
x0, y0 = 330, 200
shooter_point = np.array([x0, y0])
image_path = "./targets.png"
img_gray = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)
blurred = gaussian_blur(img_gray)
edges = sobel_edge_detection(blurred)
points = np.column_stack(np.where(edges > 0))
labels = dbscan(points)
unique_labels = set(labels)
```

After that we call `calculate_bounding_circle` method, calculate the x, y and r of the targets (clusters) and save it in the circle array as a tuple and sort them according to which is the closest to the shooter point by calling the `sort_targets` method.

```
circles = []

for cid in unique_labels:
    if cid == -1:
        continue
    cluster_pts = points[labels == cid]
    cx, cy, r = calculate_bounding_circle(cluster_pts)
    circles.append((cx, cy, r))
sorted_circles = sort_targets(circles, shooter_point)
```

Then we calculate the trajectory of the shot ball, by going over them in a for loop, we shot the targets one by one, to calculate the velocity we call the `shooting_v` method and go over it using `euler_method_step`, then save our x and y values and plot them.

```
paths = []
for (cx, cy, r) in sorted_circles:
    velocity = shooting_v(np.array([x0, y0]), np.array([cx, cy]), steps=200)
    p = np.array([x0, y0], dtype=np.float64)
    v = velocity
    x_vals, y_vals = [p[0]], [p[1]]
    for _ in range(200):
        p, v = euler_method_step(p, v, ball_dynamics)
        x_vals.append(p[0])
        y_vals.append(p[1])
```

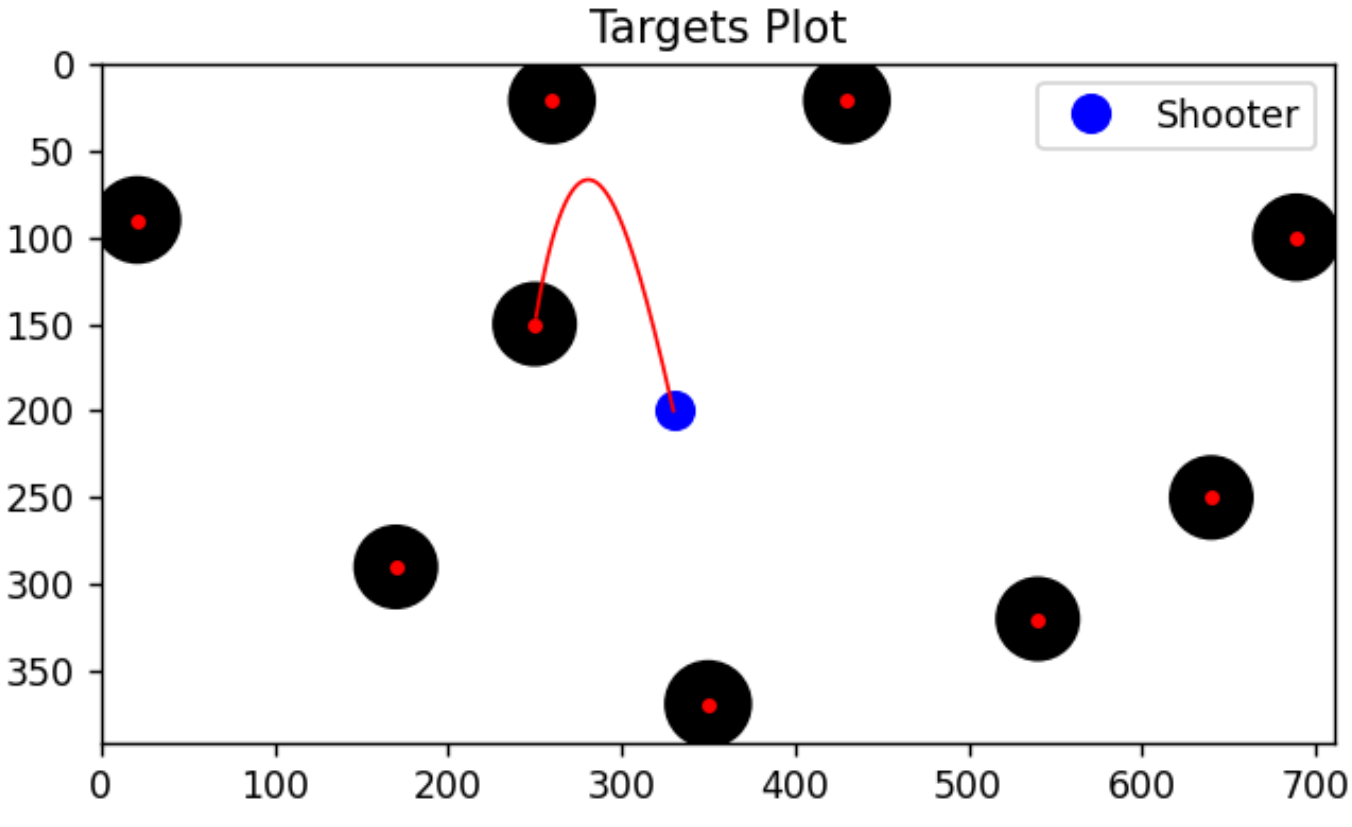
```
    paths.append((x_vals, y_vals))

fig, ax = plt.subplots(figsize=(6, 6))
ax.set_title("Targets Plot")
ax.set_aspect("equal", "box")
ax.invert_yaxis()
ax.set_xlim([0, img_gray.shape[1]])
ax.set_ylim([img_gray.shape[0], 0])
ax.plot(shooter_point[0], shooter_point[1], 'bo', markersize=10, label='Shooter')
```

Finally we have the animation using `matplotlib`, after each target is hit they disappear, and the trajectory taken is animated.

```
circle_patches = []
centroid_patches = []
for (cx, cy, r) in sorted_circles:
    circle = plt.Circle(
        (cx, cy), radius=r,
        color='black',
        linewidth=2
    )
    ax.add_patch(circle)
    centroid = ax.plot(cx, cy, 'ro', markersize=3)[0]
    circle_patches.append(circle)
    centroid_patches.append(centroid)

ax.legend(loc='upper right')
path_line, = ax.plot([], [], 'r-', linewidth=1)

def update(frame, path_line, paths, circle_patches, centroid_patches):
    current_path_index = frame // 200
    current_frame = frame % 200
    if current_path_index < len(paths):
        x_vals, y_vals = paths[current_path_index]
        path_line.set_data(x_vals[:current_frame + 1], y_vals[:current_frame + 1])
        if current_frame == 199:
            circle_patches[current_path_index].remove()
            centroid_patches[current_path_index].remove()
            path_line.set_data([], [])
    return path_line,


ani = FuncAnimation(fig, update, frames=len(paths) * 200, fargs=(path_line, paths, circle_patche
plt.show()
```

Result

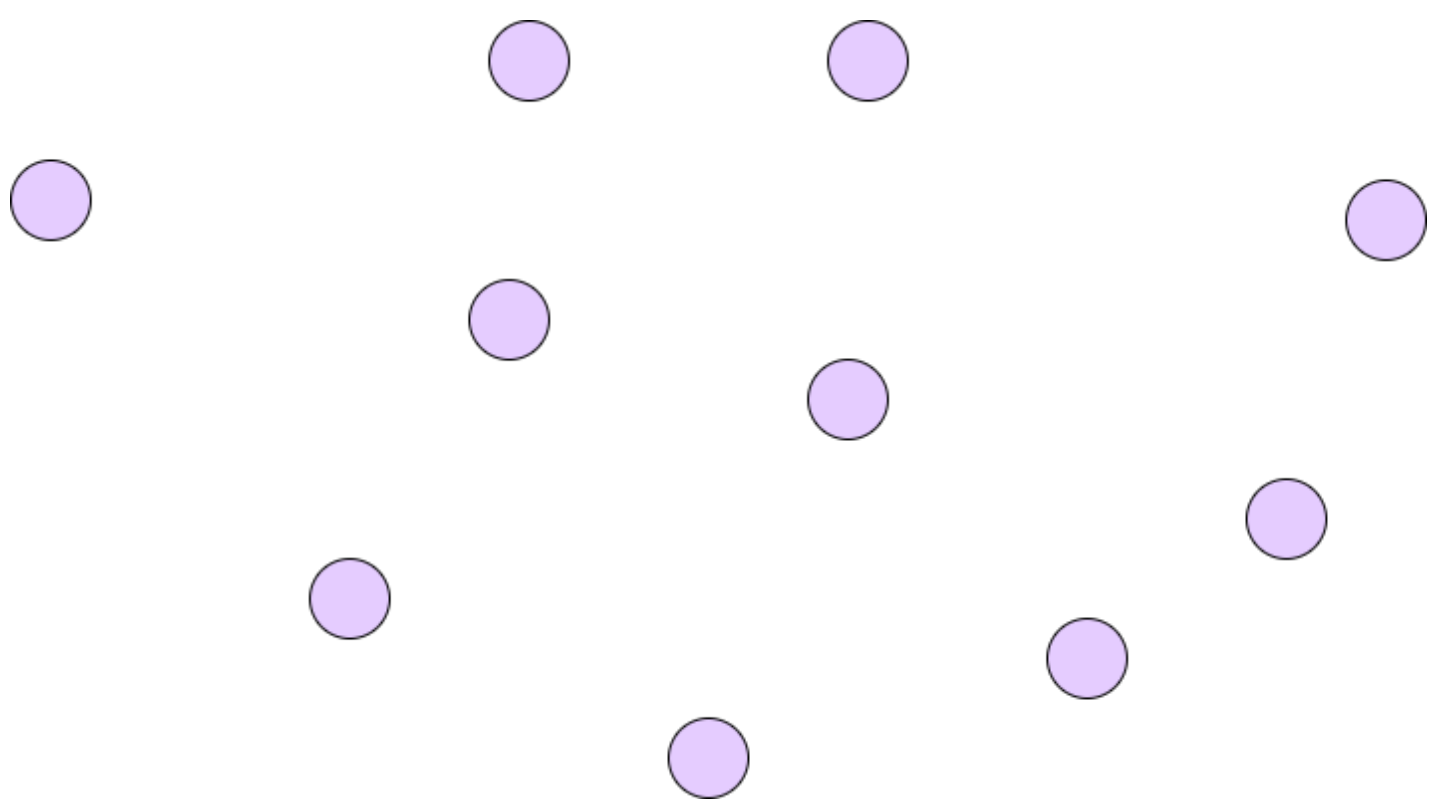
The ball is shot from the shooter point, and hits all the objects that are shown on the image, one by one, starting from the closest one, after they are hit they disappear, we use shooting method, Euler, RK2, and for error Newton.



Tests

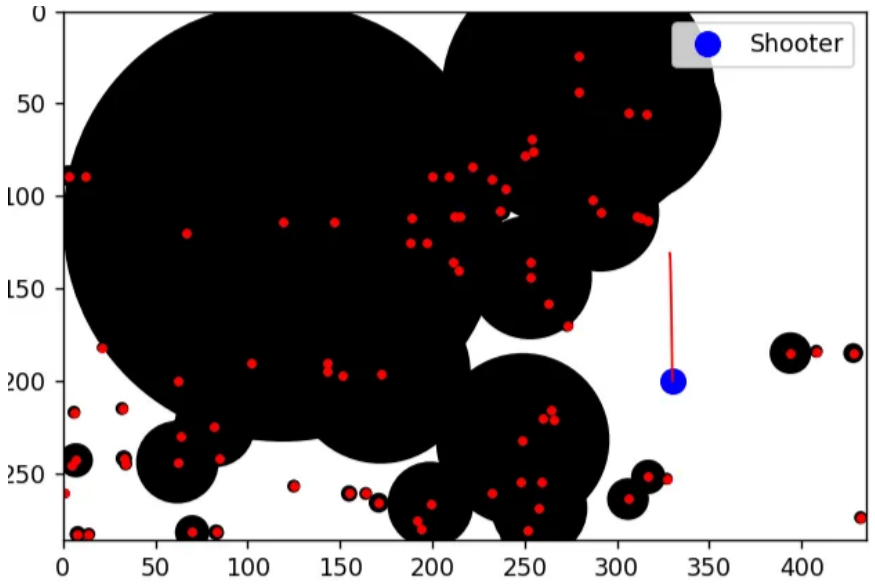
Successful Test

every image that does not have a complicated background, the ones that the edge detection and DBSCAN work for.



Unsuccessful Test

images that have a complicated background and for which the edge detection and DBSCAN do not work for.



Conclusion

This code implements detecting and targeting objects in an image, simulating projectile motion to visualize paths towards the targets. It begins with Gaussian blurring and Sobel edge detection to preprocess the image and extract edges, followed by clustering detected edge points using the DBSCAN algorithm to identify distinct target regions. Bounding circles are computed for each cluster, representing the targets, and these targets are sorted based on their proximity to a predefined shooter point. Using numerical methods like Euler and Runge-Kutta integration, the code computes and visualizes the projectile trajectories for each target based on Ball motion ODE. The interactive animation provides a clear visualization of the simulated targeting process.