# Numerical Programming

*Hit a Ball to a Fixed Target*

Lado Turmanidze

January 20, 2025

## Problem Statement

### Hit a ball to a 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.

### 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.

# 1 Hit a ball to a fixed target ⭘

For purposes of this project, non-max suppression, double thresholding and hysteresis is overkill, so edge detection will simply be reduced to simple Sobel operator convolution. The indexing in *grad_x* and *grad_y* might seem weird at first, but I believe it to be a clever way to apply the Sobel kernels to the image. The Sobel kernels are $3 \times 3$ matrices that are convolved with the image to calculate the gradient.

*grad_x[1:-1, 1:-1]*: This is selecting a sub-array of *grad_x* that starts from the second row and column (index 1) and goes up to the second last row and column (index $-1$). This is done to avoid the edges of the image, where the Sobel kernel would extend beyond the image boundaries.

*img[:-2, :-2]...* : These are selecting sub-arrays of *img* that are shifted by one or two rows/-columns relative to the current position. This is done to apply the Sobel kernel to the image.

```
def get_edge_points(img):
    # Sobel kernels for x and y matrices
    sobel_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobel_y = np.array([[-1, -2, -1], [0,  0,  0], [1,  2,  1]])

    grad_x = np.zeros_like(img, dtype=float)
    grad_y = np.zeros_like(img, dtype=float)

    grad_x[1:-1, 1:-1] = (
        sobel_x[0, 0] * img[:-2, :-2] + sobel_x[0, 1] * img[:-2, 1:-1] + sobel_x[0, 2] * img[:-2, 2:] +
        sobel_x[1, 0] * img[1:-1, :-2] + sobel_x[1, 1] * img[1:-1, 1:-1] + sobel_x[1, 2] * img[1:-1, 2:] +
        sobel_x[2, 0] * img[2:, :-2] + sobel_x[2, 1] * img[2:, 1:-1] + sobel_x[2, 2] * img[2:, 2:]
    )

    grad_y[1:-1, 1:-1] = (
        sobel_y[0, 0] * img[:-2, :-2] + sobel_y[0, 1] * img[:-2, 1:-1] + sobel_y[0, 2] * img[:-2, 2:] +
        sobel_y[1, 0] * img[1:-1, :-2] + sobel_y[1, 1] * img[1:-1, 1:-1] + sobel_y[1, 2] * img[1:-1, 2:] +
        sobel_y[2, 0] * img[2:, :-2] + sobel_y[2, 1] * img[2:, 1:-1] + sobel_y[2, 2] * img[2:, 2:]
    )

    grad_magnitude = np.sqrt(grad_x**2 + grad_y**2)
    strong_edges = grad_magnitude > 100.0  # 100 is simply a threshold for strong edges
    edge_points = np.column_stack(np.where(strong_edges))

    return edge_points
```

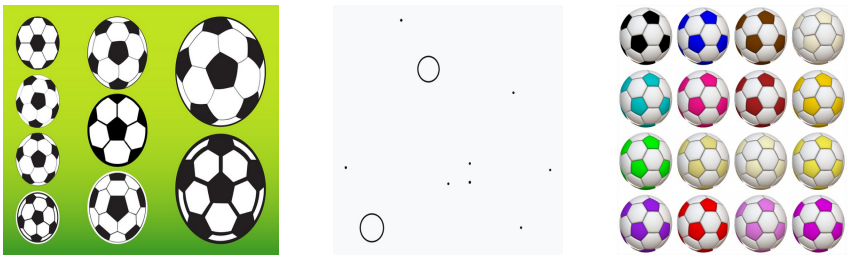


Figure 1: Original Images

The method above only does grayscaling and edge detection, but you can see how good it is for different kinds of images below:

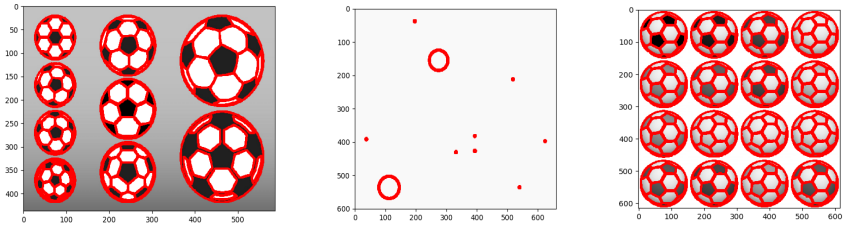


Figure 2: Images After Edge Detection(and Grayscaling)

I will continue demonstrating progress using the image in the middle, as it has proven to be the fastest for testing purposes:

```
Edge detection took 0.05 seconds
DBSCAN clustering took 25.58 seconds

Edge detection took 0.07 seconds
DBSCAN clustering took 2.46 seconds

Edge detection took 0.05 seconds
DBSCAN clustering took 48.70 seconds
```

As I am not expecting a specific number of balls in the image, I have decided to use DBSCAN for clustering instead of $K$-Means/$K$-Medoids. However, to enhance the efficiency of searching for neighboring points, I have made two key modifications: I utilize a Grid data structure and the Disjoint Set Union (DSU).

## Grid Data Structure

*Grid* class is used to efficiently locate neighboring points. Instead of comparing every point with every other point (which would be the case for naive implementation of DBSCAN), I divide the space into grid cells. Each point is mapped to a specific grid cell based on its coordinates. When searching for neighbors, one only needs to consider points in the same cell as the target point, as well as in the neighboring cells.

```python
class Grid:
    def __init__(self, cell_size):
        self.cell_size = cell_size
        self.grid = defaultdict(list)

    def get_cell_coords(self, point):
        return tuple((point // self.cell_size).astype(int))

    def insert(self, idx, point):
        cell_coords = self.get_cell_coords(point)
        self.grid[cell_coords].append(idx)

    def get_neighbors(self, point):
        cell_coords = self.get_cell_coords(point)
        neighbors = []
        for dx in (-1, 0, 1):
            for dy in (-1, 0, 1):
                neighbor_coords = (cell_coords[0] + dx, cell_coords
    [1] + dy)
                neighbors.extend(self.grid[neighbor_coords])
        return neighbors


class DSU:
    def __init__(self, size):
        self.parent = np.arange(size)
        self.rank = np.zeros(size, dtype=int)

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # Path
    compression
        return self.parent[x]

    def union(self, x, y):
        root_x = self.find(x)
        root_y = self.find(y)
        if root_x != root_y:
            if self.rank[root_x] > self.rank[root_y]:
                self.parent[root_y] = root_x
            elif self.rank[root_x] < self.rank[root_y]:
                self.parent[root_x] = root_y
            else:
                self.parent[root_y] = root_x
                self.rank[root_x] += 1
```

## Disjoint Set Union(DSU)

*DSU* class is used to group points that belong to the same cluster. The DSU data structure helps manage the merging of clusters in an efficient way by keeping track of the connected components. It supports two key operations:

- ☞ **find**: This operation determines the "root" or representative of the set that a point belongs to. It uses path compression to speed up future queries.

- ✌ **union**: This operation merges two sets (clusters) if they are connected, using union by rank to maintain a balanced tree structure.

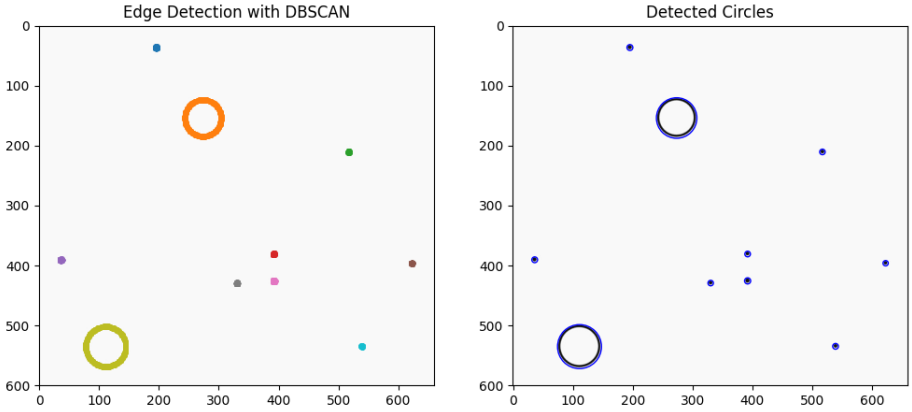
In the DBSCAN algorithm, as points are found within the specified epsilon distance (*eps*) from each other, I use the union operation to merge their sets (effectively assigning them to the same cluster). By doing so, points that are close enough to each other are grouped into the same cluster.

In short, *Grid* helps reduce the number of unnecessary comparisons, while the *DSU* ensures that clusters are formed quickly and efficiently.

Prior to moving on with *DBSCAN*, let's quickly define second vector norm, the most intuitive one as a measure of distance:

```
def norm2(a: np.array, b: np.array) -> float:
    return np.sqrt(np.sum((a - b) ** 2))
```

And because code for the *DBSCAN* is large, I will first demonstrate result of the *DBSCAN.dbscan(self, X)* and will then proceed with the entire class:




As one can see in the left figure, *DBSCAN(eps*=3*, min_pts*=10*)* perfectly identified all 10 different clusters and gave each of them a unique color. Code is written such that it will try to color each cluster in different color, but obviously, this might bee hard to see if there are many clusters.

In the figure on the right, I draw circles based on the radii and centers of the respective clusters, as the array of radii and centers are saved in *dbscan.cluster_radii_* and *dbscan.cluster_radii_* after *DBSCAN.dbscan(self, X)*.

## DBSCAN with Grid and DSU

```
class DBSCAN:
    # I think the constructor is pretty self explanatory
    def __init__(self, eps, min_pts, norm=norm2):
        self.eps = eps
        self.min_pts = min_pts
        self.norm = norm
        self.labels_ = None
        self.cluster_centers_ = None
        self.cluster_radii_ = None

    def dbscan(self, X):
        n_samples = X.shape[0]
        dsu = DSU(n_samples)
        grid = Grid(cell_size=np.ceil(self.eps))

        # Inserting points in the grid
        for idx, point in enumerate(X):
            grid.insert(idx, point)

        # Perform clustering using the grid
        for idx, point in enumerate(X):
            neighbors = grid.get_neighbors(point)
            for neighbor_idx in neighbors:
                if neighbor_idx != idx and self.norm(X[idx], X[
    neighbor_idx]) <= self.eps:
                    dsu.union(idx, neighbor_idx)

        # Assign cluster IDs based on connected components
        root_to_cluster = {}
        cluster_id = 0
        self.labels_ = np.full(n_samples, -2, dtype=int)  # -2
    represents noise points

        for i in range(n_samples):
            root = dsu.find(i)
            if root not in root_to_cluster:
                root_to_cluster[root] = cluster_id
                cluster_id += 1
            self.labels_[i] = root_to_cluster[root]

        # Calculate cluster centers and radii
        centers = []
        radii = []
        for c_id in set(self.labels_):
            if c_id >= 0:  # Ignore noise
            cluster_points = X[self.labels_ == c_id]
            if len(cluster_points) >= self.min_pts:
                center = np.mean(cluster_points, axis=0)
                radius = np.max([self.norm(point, center) for point
     in cluster_points])
                centers.append(center)
                radii.append(radius)

        filtered_centers = []
        filtered_radii = []
```

```
        for i, (center, radius) in enumerate(zip(centers, radii)):
            keep = True
            for j, (other_center, other_radius) in enumerate(zip(
    centers, radii)):
                if i != j and self.norm(center, other_center) <
    radius + other_radius:
                    if radius <= other_radius:
                        keep = False
                        break
            if keep:
                filtered_centers.append(center)
                filtered_radii.append(radius)

        self.cluster_centers_ = np.array(filtered_centers)
        self.cluster_radii_ = np.array(filtered_radii)

        return self
```

The *dbscan* method begins by initializing *DSU* to manage clusters and a *Grid* for efficient spatial indexing of points. Each point‘ is inserted into the grid based on its cell coordinates, which are computed using the grid's cell size set to *ceil(eps)*. After constructing the grid, the method processes each point to retrieve its neighbors using the grid. For every neighboring point that lies within a distance *eps*, the method merges the two points into the same cluster using *DSU.union(self, x, y)*.

Once all points are processed, the method identifies clusters by examining the connected components in *DSU*. Each unique component is assigned a cluster ID, while points that do not belong to any cluster are labeled as noise a.k.a. $-2$. After assigning cluster IDs, the method computes the centers and radii for each valid cluster. For clusters containing at least *min_pts* points, the center is calculated as the mean of all points in the cluster, and the radius is determined as the maximum distance from the center to any point in the cluster.

The final step (part of the code on this page) filters out intersecting circles based on their radii. It iterates over each circle represented by its center and radius. For each circle, it is compared with every other circle to check if they intersect. Two circles intersect if the distance between their centers is less than the sum of their radii. If an intersection is found, the circle with the smaller radius is marked for removal. This is achieved by setting a *keep* flag to *False* and breaking out of the inner loop. After all comparisons, only circles that are not marked for removal are added to the *filtered_centers* and *filtered_radii* lists. These filtered results are then assigned to the class attributes *self.cluster_centers_* and *self.cluster_radii_*.

Finally, the method stores the cluster labels, centers, and radii as class attributes: *labels_*, *cluster_centers_*, and *cluster_radii_*, respectively, and returns the results.

## Why is there a need for circle filtering?

Figure 3: Original

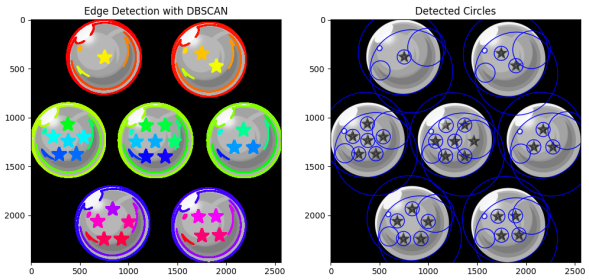



Figure 4: ED

Figure 5: Dragon Balls prior to Circle Filtering

We can see on right figure above that many circles intersect, but after filtering, on the right figure below, there are no extra circles. Although on the right figure below, there is a problem of overdetecting radius, which can be explained by "shining" of the balls in the top left corner, as pixels in those areas tend to "stretch" clusters more. Also, note how for balls of this size(the image is huge, compared to others: $2560 \times 2477$) there are clusters somewhat horizontally, but this is not an issue, since different circles in the same cluster get detected nonetheless.

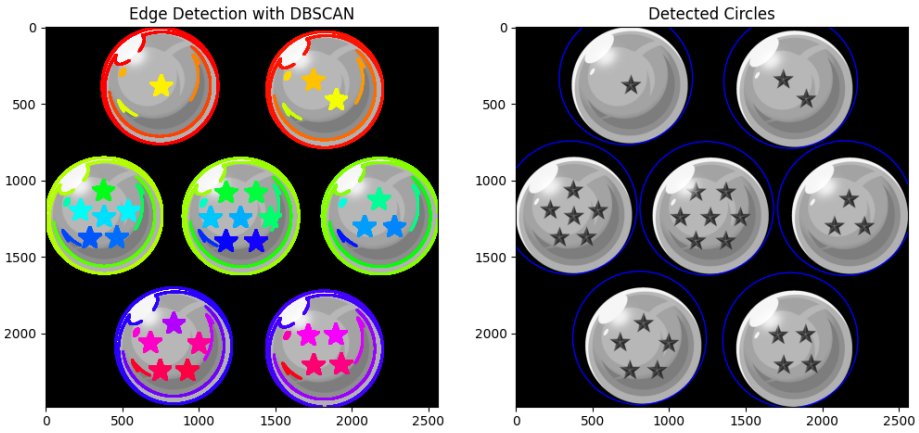



Figure 6: Dragon Balls with Circle Filtering

## 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.

*Shooter1vN* class will be responsible for the entire simulation. *get_edge_points* method will now become static method of this class. Constructor of this class:

```
class Shooter1vN:
    def __init__(self, img_path, seed=95):
        self.img = np.array(Image.open(img_path))
        if self.img is None:
            raise ValueError(f"Could not load image from path: {
    img_path}")
        (self.shooter_position, self.shooter_radius), (self.centers
    , self.radii) = self.setup(self.img, seed)
        # For output
        self.shooter = MainGuy(800, 600)
        self.setup_simulation()
```

Some things I want to make clear:

- ↯ I have added *seed=95* so that shooter's position is randomized among possible positions, but seed value makes the test reproducible. 95 in honor of Lightning McQueen ↯.
- ⊛ I raise error if invalid image path is used for creating instance of *Shooter1vN*. Happens to me all the time.
- ⊜ *MainGuy* is the class that will be used for simulation. $800 \times 600$ video will be created for the simulation that is set up on the next line.

```
def setup(self, img, seed=95, shooter_size=15):
    random.seed(seed)
    np.random.seed(seed)

    edge_points = self.get_edge_points(img)
    dbscan = DBSCAN(eps=5, min_pts=25)
    dbscan.dbscan(edge_points)
    centers = dbscan.cluster_centers_
    radii = dbscan.cluster_radii_

    def is_valid_position(candidate, centers, r, min_distance=5):
        for center, radius in zip(centers, r):
            distance = np.linalg.norm(candidate - center)
            if distance < radius + shooter_size + 2 or distance <
    min_distance:
                return False
            return True

    img_shape = np.array(img).shape[:2]
    while True:
        candidate = np.random.uniform(low=[0, 0], high=img_shape)
        if is_valid_position(candidate, centers, radii):
            break

    return (candidate, shooter_size), (centers, radii)
```

The *setup* method initializes a shooting position within an image by first setting a random seed for reproducibility. It retrieves edge points from the image using now static method of this class - *get_edge_points* and applies *DBSCAN* to identify cluster centers and their radii. A nested function checks if a randomly generated candidate position is valid by ensuring it maintains a safe distance from the cluster centers, considering the shooter size and a minimum distance. The method continues generating random positions until it finds one that meets these criteria and returns the valid candidate position along with the cluster centers and their radii.

```
   def current_state(self):
    img_copy = cv2.cvtColor(np.array(self.img.copy()), cv2.
    COLOR_BGR2RGB)

    for center, radius in zip(self.centers, self.radii):
        cv2.circle(img_copy,(int(center[1]), int(center[0])),  #
    Swap x and y for OpenCV's coordinate system
            int(radius),(255, 0, 0),2
        )

    cv2.circle(img_copy, (int(self.shooter_position[1]), int(self.
    shooter_position[0])),
        int(self.shooter_radius),(0, 255, 0), -1  # Filled circle
    )

    plt.figure(figsize=(8, 8))
    plt.imshow(img_copy)
    plt.axis('off')
    plt.title("Current State")
    plt.show()
```

The *current_state* method visualizes the current state of the image. It iterates through the identified cluster centers and their radii, drawing red circles around each cluster on the image. Then, it draws a filled green circle at the shooter's position, representing the shooter. Finally, the method displays the modified image. This function will only be used for testing purposes, as simulation will be entirely based on **raylib**'s python version: **pyray**.

```
def setup_simulation(self):
    self.shooter.set_shooter((self.shooter_position[1], self.
    shooter_position[0]), self.shooter_radius)
    for center, radius in zip(self.centers, self.radii):
        self.shooter.add_target(center[1], center[0], radius)


def run_simulation(self):
    self.shooter.simulate()
```

Both of the methods above heavily depend upon *MainGuy* class, which will be explained later on. The *setup_simulation* method prepares the simulation by configuring the shooter with its position and radius, ensuring it is set up correctly for the simulation environment. It then iterates through the identified cluster centers and their corresponding radii, adding each target to the shooter. The *run_simulation* method subsequently executes the simulation by calling the *simulate* function of the shooter, processes the shooting mechanics and interactions with the targets. Together, these methods establish the necessary parameters and initiate the simulation of the shooting scenario.

## MainGuy Pre-Requisites

*MainGuy* class uses *Ball* and PhysicsParams classes, along with methods for calculating accelaration, RK4 and the shooting method.

```
class Ball:
    def __init__(self, x: float, y: float, vx: float, vy: float,
    radius: float = 5.0):
        self.x = x
        self.y = y
        self.vx = vx
        self.vy = vy
        self.radius = radius
        self.active = True

    def get_state(self) -> np.ndarray:
        return np.array([self.x, self.y, self.vx, self.vy])

    def set_state(self, state: np.ndarray):
        self.x, self.y, self.vx, self.vy = state
```

*Ball* class represents a moving ball in the simulation. It stores the ball's position $(x, y)$, velocity $(v_x, v_y)$, and radius, with default values provided for the radius. The active attribute indicates if the ball is still in play. The class also includes *get_state* and *set_state* methods to retrieve and update the ball's state as a NumPy array, which will be useful for numerical simulations.

```python
@dataclass
class PhysicsParams:
    phi: float = 0.0005  # Air resistance coefficient
    g: float = 100.0  # Gravity
    dt: float = 0.01  # Time step
    time_steps: int = 200  # Number of simulation steps
    h: float = 0.001  # Step size for shooting method
```

**@***dataclass* decorator for *PhysicsParams* class automatically generates common methods like *__init__*, *__repr__*, and *__eq__*, simplifying the creation of classes intended to store data.

In this class, *phi* ($\varphi$) is equivalent to $\frac{k}{m}$.

### Helper Methods

```python
def acceleration(v: Tuple[float, float], params: PhysicsParams) ->
    Tuple[float, float]:
    vx, vy = v
    mag = np.sqrt(vx * vx + vy * vy)
    return (-params.phi * vx * mag, params.g - params.phi * vy *
    mag)

def derivative_vector(state: np.ndarray, params: PhysicsParams) ->
    np.ndarray:
    x, y, vx, vy = state
    ax, ay = acceleration((vx, vy), params)
    return np.array([vx, vy, ax, ay])

def rk4_step(ball: Ball, params: PhysicsParams) -> None:
    state = ball.get_state()
    k1 = derivative_vector(state, params)
    k2 = derivative_vector(state + 0.5 * params.dt * k1, params)
    k3 = derivative_vector(state + 0.5 * params.dt * k2, params)
    k4 = derivative_vector(state + params.dt * k3, params)

    new_state = state + (params.dt/6.0) * (k1 + 2*k2 + 2*k3 + k4)
    ball.set_state(new_state)
```

I will need acceleration of the ball in both directions for the output video.

$$a_x = \frac{dv_x}{dt} = -\frac{k}{m} v_x \sqrt{v_x^2 + v_y^2},$$
$$a_y = \frac{dv_y}{dt} = -g - \frac{k}{m} v_y \sqrt{v_x^2 + v_y^2}.$$

REMINDER: In the code *phi* ($\varphi$) is equivalent to $\frac{k}{m}$.
Derivative vector for all the functions give by ODEs is:

$$\frac{d}{dt}\begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix} = \begin{bmatrix} v_x \\ v_y \\ -\frac{k}{m} v_x \sqrt{v_x^2 + v_y^2} \\ -g - \frac{k}{m} v_y \sqrt{v_x^2 + v_y^2} \end{bmatrix}. \tag{4}$$

Moving on with RK4 method, let $\mathbf{y} = [x, y, v_x, v_y]$, the RK4 method calculates:

$$
\begin{aligned}
k_1 &= f(t, \mathbf{y}), \\
k_2 &= f\left(t + \frac{\Delta t}{2}, \mathbf{y} + \frac{\Delta t}{2} k_1\right), \\
k_3 &= f\left(t + \frac{\Delta t}{2}, \mathbf{y} + \frac{\Delta t}{2} k_2\right), \\
k_4 &= f\left(t + \Delta t, \mathbf{y} + \Delta t \cdot k_3\right).
\end{aligned}
$$

The next state is computed by:

$$\mathbf{y}_{\text{next}} = \mathbf{y} + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4). \tag{5}$$

This method accurately integrates the ball's motion considering drag and gravity described by (1), (2) and (3).

Let's also add Heun's RK2 for comparison later on:

```
def heun_rk2_step(ball: Ball, params: PhysicsParams) -> None:
    state = ball.get_state()

    # Predictor step (Euler's method)
    k1 = derivative_vector(state, params)
    predictor = state + params.dt * k1

    # Corrector step (average slope)
    k2 = derivative_vector(predictor, params)

    # Update state
    new_state = state + (params.dt / 2.0) * (k1 + k2)
    ball.set_state(new_state)
```

### 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.

Now comes the main method, THE Shooting Method:

*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:

$$J = \begin{bmatrix} \dfrac{\partial F_1}{\partial v_{x_0}} & \dfrac{\partial F_1}{\partial v_{y_0}} \\ \dfrac{\partial F_2}{\partial v_{x_0}} & \dfrac{\partial F_2}{\partial v_{y_0}} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial x}{\partial v_{x_0}} & \dfrac{\partial x}{\partial v_{y0}} \\ \dfrac{\partial y}{\partial v_{x_0}} & \dfrac{\partial y}{\partial v_{y0}} \end{bmatrix}. \tag{6}$$

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.

## MainGuy

```
class MainGuy:
    def __init__(self, width: int = 800, height: int = 600):
        self.width = width
        self.height = height
        self.params = PhysicsParams()
        self.current_ball: Optional[Ball] = None
        self.targets: List[Tuple[float, float, float]] = []  # x, y, radius
        self.active_targets: List[bool] = []  # Track which targets are still active
        self.shooter_pos = (0, 0)
        self.shooter_radius = 0
        self.current_target_index = 0
        self.shot_fired = False

    def set_shooter(self, pos: Tuple[float, float], radius: float):
        self.shooter_pos = pos
        self.shooter_radius = radius

    def add_target(self, x: float, y: float, radius: float):
        self.targets.append((x, y, radius))
        self.active_targets.append(True)
```

I believe most of the attributes of the constructor, *set_shooter* and *add_target* methods are self-explanatory, except for these attributes in the constructor that might need explanation: *active_targets* list mirrors the targets list, keeps track of whether each target is still "alive" or has been hit and *shot_fired* flag indicates whether a ball is currently in motion, ensuring no unnecessary actions take place during simulation.

```
def check_collision(self) -> bool:
    if not self.current_ball or not self.current_ball.active:
        return False

    target = self.targets[self.current_ball.target_index]
    if not self.active_targets[self.current_ball.target_index]:
        return False

    dx = self.current_ball.x - target[0]
    dy = self.current_ball.y - target[1]
    distance = np.sqrt(dx * dx + dy * dy)

    return distance < (self.current_ball.radius + target[2])
```

*check_collision* method determines whether the current ball has hit its target. It first checks if a ball is active and in motion. If not, it returns **False**, avoiding unnecessary calculations. The method then retrieves the target associated with the ball's *target_index*. If this target is inactive, the method again returns **False**. If both the ball and target are valid, the method calculates the Euclidean distance between the ball's center and the target's center using: distance $= \sqrt{(dx)^2 + (dy)^2}$ A collision is detected if this distance is less than the sum of the ball's and target's radii. If so, the method returns **True**; otherwise, it returns **False**.

```
def shoot_next_ball(self) -> bool:
    active_targets = [(i, t) for i, t in enumerate(self.targets) if
     self.active_targets[i]]
    if not active_targets:
        return False

    active_targets.sort(key=lambda t: np.hypot(t[1][0] - self.
    shooter_pos[0], t[1][1] - self.shooter_pos[1]))
    closest_index, target = active_targets[0]
    vx, vy = shooting_method(self.shooter_pos[0], self.shooter_pos
    [1], target[0], target[1], self.params)

    self.current_ball = Ball(self.shooter_pos[0], self.shooter_pos
    [1], vx, vy)
    self.current_ball.target_index = closest_index
    self.shot_fired = True
    return True
```

*shoot_next_ball* method is responsible for firing the next ball towards the closest active target. First, it identifies all active targets by filtering through the *targets* list and checking corresponding entries in *active_targets*. If no targets remain, the method returns *False*, signaling there are no more balls to shoot.

The active targets are then sorted based on their distance from the shooter's position. This is calculated using the Euclidean distance formula. The closest target is selected, and its index and coordinates are stored.

Next, the *shooting_method* function is invoked to compute the initial velocity ($v_x, v_y$ needed for the ball to hit the target. This method iteratively adjusts the velocities using a numerical approach to minimize errors in the ball's trajectory.

A new *Ball* object is created at the shooter's position, initialized with the computed velocity. The ball is associated with the selected target via its *target_index*. The method sets the *shot_fired* flag to **True** and returns **True**, indicating the ball has been successfully fired. This method combines dynamic target selection with precise trajectory calculation to ensure accurate shooting.

Finally, *simulate* method puts together the visual and interactive simulation of the entire system. Within each frame, the method draws the active targets in red and the shooter in green, providing visual context for the game. If no ball has been fired yet, *shoot_next_ball* is called to initiate a shot. If there are no remaining targets, the simulation exits gracefully.

Once a ball is active, its position and velocity are updated using the Runge-Kutta method (*rk4_step* or *heun_rk2_step*), which simulates realistic physics, including gravity and air resistance. The ball is drawn in blue as it moves, providing real-time feedback on its trajectory.

Collisions between the ball and targets are checked using *check_collision*. If a collision is detected, the target is deactivated, and the ball is removed. If the ball exits the screen bounds, it is also deactivated. The process then repeats, preparing for the next shot.

```
def simulate(self):
    init_window(self.width, self.height, "THE MainGuy Simulation")
    set_target_fps(60)

    while not window_should_close():
        begin_drawing()
        clear_background(WHITE)

        for i, (tx, ty, tr) in enumerate(self.targets):
            if self.active_targets[i]:
                draw_circle(int(tx), int(ty), tr, RED)

        draw_circle(int(self.shooter_pos[0]), int(self.shooter_pos
    [1]), self.shooter_radius, GREEN)

        if not self.shot_fired:
            if not self.shoot_next_ball():
                if self.current_target_index >= len(self.targets):
                    break
        elif self.current_ball and self.current_ball.active:
            rk4_step(self.current_ball, self.params)
            draw_circle(int(self.current_ball.x), int(self.
    current_ball.y), self.current_ball.radius, BLUE)

            if self.check_collision():
                self.active_targets[self.current_ball.target_index]
     = False
                self.current_ball.active = False
                self.shot_fired = False
                self.current_target_index += 1

            if (self.current_ball.x < 0 or self.current_ball.x >
    self.width or
                self.current_ball.y < 0 or self.current_ball.y >
    self.height):
                self.current_ball.active = False
                self.shot_fired = False

        end_drawing()
    close_window()
```

I tested with *heun_rk2_step* as solver, for all the images I provided as tests, but no visual results were visible.

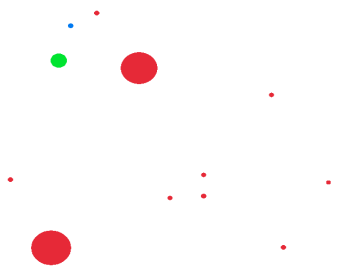


Figure 7: A single frame from the output video

## When does the code fail?

The code obviously fails with images that contain shadows and/or edges other than the ball, as it will probably see them as a "normal" edge and fail to remove it. This may be avoidable with more advanced techniques, such as background subtraction. More optimized and sophisticated implementations of edge detection will also probably do a better job.

Another case, where this code fails, is when balls are not entirely in the image and only part of them are visible:

Figure 8: Image destined to fail on the code

If you look at the top corners of the edge detected image below, you can see part of the red circle, meaning this code clustered all balls as one ball. As such, simulation will show only a single big ball and end instantly.

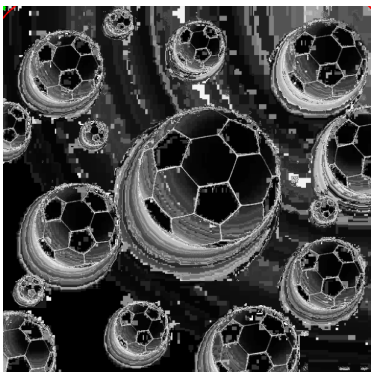


Figure 9: Mis-clustering of the balls

## References

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