Hitting Fixed Targets

Giorgi Otinashvili

30 December, 2024

Problem: Throw a ball and hit fixed targets on image one after another. Use edge detection and solve boundary value problem using shooting method. Create a corresponding animation.

I split the problem into 4 subproblems. Let's explain each one of them:

1. Generating an image

First of all, we need to create an image that contains fixed targets. For this, I created ***`image_generator.py`***. Its method ***`generate_image(self, num_targets)`*** generates an image with a specific number of randomly placed targets, where target is also an image that the user can pass. Users can pass their own image as parameter, so it doesn't necessarily need to be a target image. Targets can also be made of different sizes, but this is redundant and won't affect anything. The resulting image will look like this:

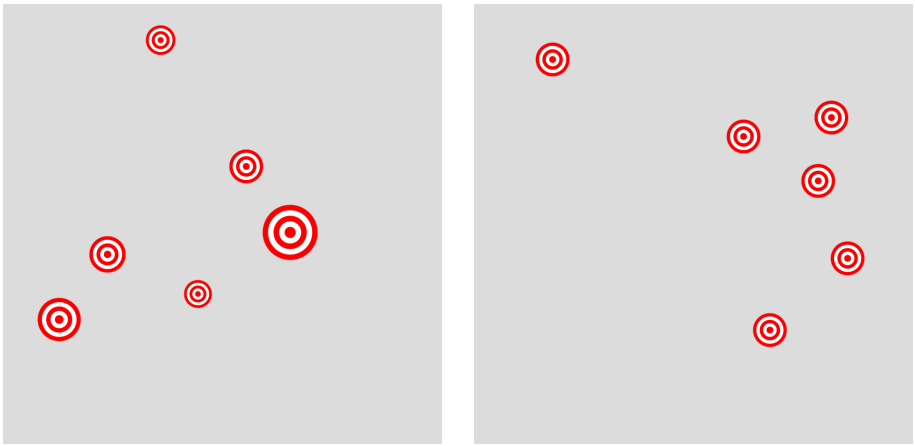


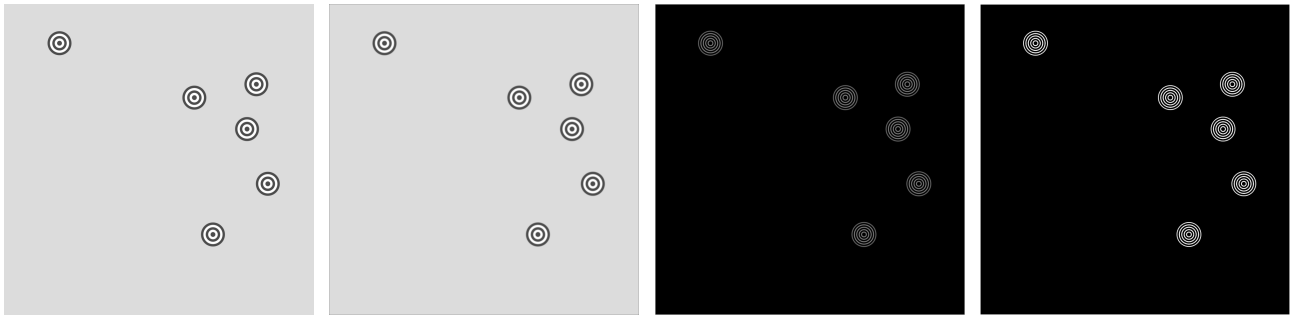
Figure 1: Images generated with different-sized and same-sized targets

I will only show examples with same-sized targets, but different sizes work as well.

2. Detecting targets

Now we need to reverse engineer what we did in the first part and detect the targets. This is done in ***detect_objects.py***.

To retrieve edges from the image, we will first convert it to grayscale and apply a Gaussian blur to reduce noise. Then, we will convolve the image with the Sobel operator using standard 3x3 kernels in both x and y directions to compute the gradient magnitude. Finally, we will apply a threshold to produce a binary image that hightlights the edges.



Grayscale | Blur | Sobel Edge Detection | Threshold

We can now find connected components (clusters) using DBSCAN algorithm similar to the one from my Computational Project 1.

The resulting objects for the above example look like this:

1 - {'center': (125, 178), 'radius': 40}
2 - {'center': (257, 813), 'radius': 40}
3 - {'center': (300, 613), 'radius': 40}
4 - {'center': (401, 783), 'radius': 40}
5 - {'center': (577, 850), 'radius': 40}
6 - {'center': (740, 673), 'radius': 40}

where centers' x and y coordinates are still in pixel values. To make it easier, we convert them to 2D coordinate system values and make a list of tuples (x, y, radius). The resulting targets are:

[(178, 874, 40), (813, 742, 40), (613, 699, 40), (783, 598, 40), (850, 422, 40), (673, 259, 40)]

3. Solving Boundary Value Problem - Shooting method

Solving the boundary value problem is the main computational task of the problem. We are given a start position (x_0, y_0) and a final position (x, y). We will also choose some time t, after which the ball should be at that final position. We then estimate initial velocities (v_x, v_y) and adjust them iteratively to ensure that the ball arrives at the target position at time t. This is done in ***bvp.py***.

This will transform the boundary value problem into an initial value problem, which will allow us to apply numerical methods to find the needed velocities.

By considering gravitational and drag forces, our ODE system is:

$$\frac{du_1}{dt} = u_2 \tag{1}$$

$$\frac{du_2}{dt} = -\frac{k\rho A}{2m}\sqrt{u_2^2 + u_4^2}u_2 \tag{2}$$

$$\frac{du_3}{dt} = u_4 \tag{3}$$

$$\frac{du_4}{dt} = -g - \frac{k\rho A}{2m}\sqrt{u_2^2 + u_4^2}u_4 \tag{4}$$

where u_1 is the x (horizontal) position of a ball, u_2 is the velocity in x direction, u_3 is the y (vertical) position and u_4 is the velocity in y direction. g is Earth's gravitational constant (g $\approx$ 10 m/s^2), k is the drag coefficient, ρ is the air density, A is the cross-sectional area of ball, m is the mass of ball.

Detailed explanation of this system was given in my Computational Project 2 and can also be found on the slides.

Since we focus on solving the boundary value problem, we will combine the constant term $\frac{k\rho A}{2m}$ into the term α and use it in our computations.

To iteratively change (v_x, v_y) at each iteration, I will use Newton's method which works well for multiple dimensions too, as opposed to Brent's (Bisection + Secant) method.

Newton's method works by finding zeroes of a vector $F : R^k \to R^k$ (in our case, F is the error function) and uses the Jacobian matrix to account for small changes in all dimensions.

General formula for Newton's method is

$$J_F(x_n)(x_{n+1} - x_n) = -F(x_n) \Rightarrow$$

$$\Rightarrow x_{n+1} = x_n - J_F(x_n)^{-1}F(x_n) \tag{5}$$

Our initial approximation for $\left(v_{x_0}, v_{y_0}\right)$ is based on distance between the shooting point and the target point. Approximating the ODE solution (finding final position at time t) with given $\left(v_x, v_y\right)$ is done using RK4 method. We start with initial state vector [x_0 v_{x_0} y_0 v_{y_0}] and make a step by $\delta t = 0.01$ seconds to get the next state vector.

The general formula for RK4 is

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

where

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

We use RK4 to find the final position from starting vector, then compare it to the target's position, and iteratively change initial velocities using Newton's method.

I also implemented the step with Euler's method and compared the two. The resulting initial velocities for each target were almost equal, so both methods worked correctly.

```
Euler:
Shooting from point (258, 411):
Target (178, 874): vx0 = -16.000 m/s, vy0 = 117.550 m/s
Target (813, 742): vx0 = 111.000 m/s, vy0 = 91.150 m/s
Target (613, 699): vx0 = 71.000 m/s, vy0 = 82.550 m/s
Target (783, 598): vx0 = 105.000 m/s, vy0 = 62.350 m/s
Target (850, 422): vx0 = 118.400 m/s, vy0 = 27.150 m/s
Target (673, 259): vx0 = 83.000 m/s, vy0 = -5.450 m/s
```

Euler's method

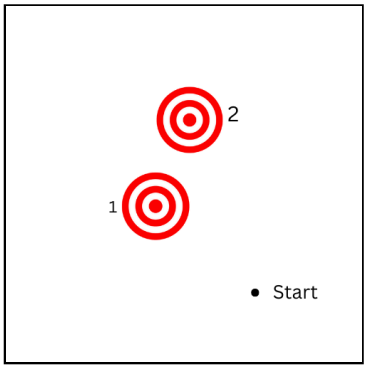
```
RK4:
Shooting from point (258, 411):
Target (178, 874): vx0 = -16.000 m/s, vy0 = 117.600 m/s
Target (813, 742): vx0 = 111.000 m/s, vy0 = 91.200 m/s
Target (613, 699): vx0 = 71.000 m/s, vy0 = 82.600 m/s
Target (783, 598): vx0 = 105.000 m/s, vy0 = 62.400 m/s
Target (850, 422): vx0 = 118.400 m/s, vy0 = 27.200 m/s
Target (673, 259): vx0 = 83.000 m/s, vy0 = -5.400 m/s
```

4th Order Runge-Kutta

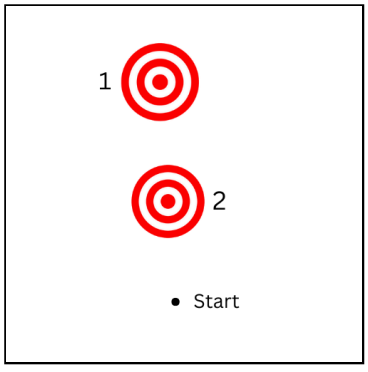
Figure 2: Comparison of Euler's method to RK4 method

We also need to avoid hiting another target while aiming for some other target. We can't just choose the closest target and aim for it. This case is shown in Figure 3.A. Target 1 is obviously closer to start point than Target 2 is. But when we choose to hit target 1, if time parameter t after which ball should reach that target is big enough, the ball will be thrown upwards and it will hit target 2 first on the trajectory to target 1. So we can't choose and hit the closest target.

Another approach is to choose such ordering of targets that guarantees that we won't hit target if there is some other target that needs to be hit before it. For this, we could construct a directed graph where each edge $i \to j$ means that we must hit i before j (i is predicate of j). Then we can use topological sort, which will arrange targets in such way that nodes with less indegree (less predicates) are hit first. But this approach may also fail in some cases. In Figure 3.B, to hit Target 2, we need to pass through Target 1 first, but to hit Target 1 (if t is big enough) we will need to throw ball upwards, meaning it should pass through target 2. Our graph will contain a cycle and there is no way to hit both targets.



A. Smallest distance doesn't work



B. Deadlock

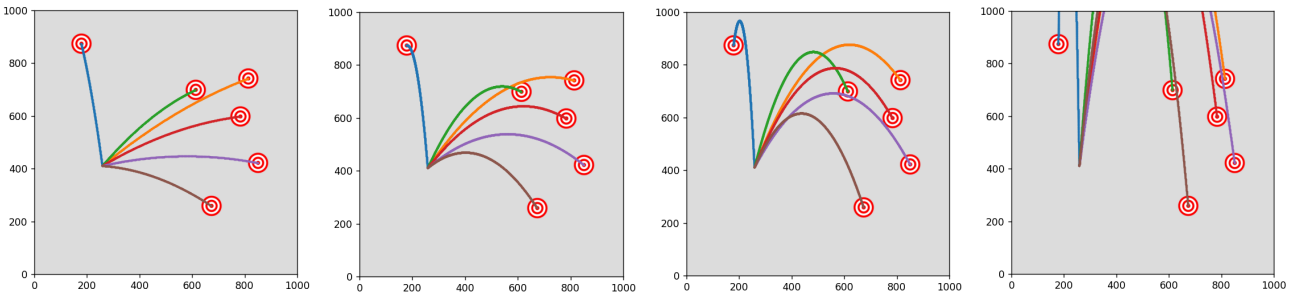
Figure 3: Cases when another target is in the way of desired target

Above cases clearly show why with fixed parameter t, it will sometimes be impossible to hit all targets without hitting others. That's why we need to be able to change t. User still inputs default t, but we can change it in some cases. Say we want to hit Target X, but on its way it hit Target Y and then Target Z. We need to first hit Y and Z, but for them we will use t equal to timestamp at which they intersected with trajectory that hit X. We estimate trajectory of Y with that decreased t, then the trajectory of Z and only then of X. To ensure that there is no other target on trajectory to hit Target Y (it's rare but it might happen), we need do the same operation with Y that we did with X: analyze its trajectory and check for overlaps with other targets. For this, I used recursion (stack) and cached trajectories (in above case, trajectory to hit X would be cached) to save computational time.

This solution ensures that we don't hit another targets while aiming for desired one. Animation of fix to Figure 3.B can be found in *output/tests/intermediate_target.mp4*.

4. Animating the trajectories

Plotted trajectories starting from some random point $(x, y) = (258, 411)$ look like this:



t = 5 seconds t = 10 seconds t = 15 seconds t = 30 seconds

Figure 4: Trajectories from a random point with different parameter t

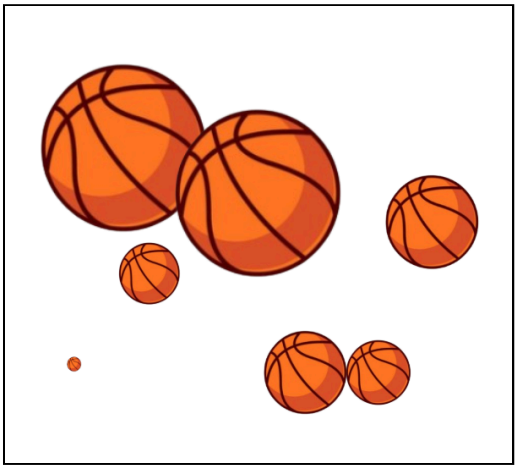
Now all we need to do is create an animation. I did it using *pygame*. All objects are replaced with target images and a ball image is used as the throwing object. After hitting a target, that target disappears. The final videos with different time parameter t can be found in *output/g=10/* folder. I also tested and made parameter $g = 100$ and $g = 200$, which are more realistic to the given scenario, since the current implementation assumes 1:1 pixel-to-meter ratio and throwing such a huge ball on such a huge map looks weird. So I scale up g to make this ratio normal, which resulted in much smoother and realistic animations. Those videos can be found in folders *output/g=100/* and *output/g=200/*. Higher g values can be tried out as well but in my opinion $g = 200$ looks the best.

The problem is solved. Now let's analyze our solution's limitations and when it may fail.

- **Bad image**

When the balls on the image either touch or are very close to each other, DBSCAN will consider them as one object and the targets will not be identified correctly.

Example:



Original Image (*assets/test4.png*)

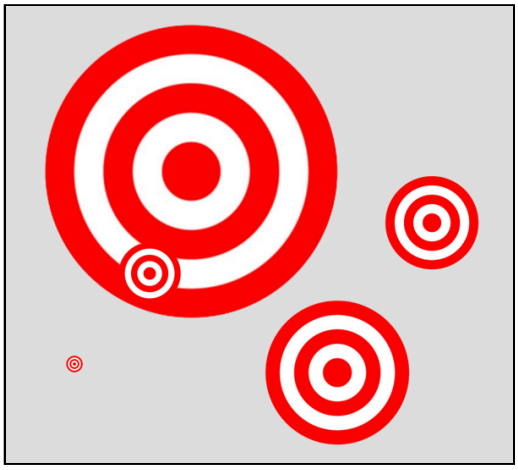


Image after detecting targets

Figure 5: Incorrectly detected balls

Two pairs of balls were identified as 1 target. This is the limitation of all primitive edge detection algorithms, so it's logical that our implementation can't handle this edge case. Hitting these targets works as expected though and it can be found in *output/tests/wrong_targets.mp4*.

There is another problem with DBSCAN is bad ε paremeter. We may need to increase it in case of bigger balls, or decrease in case of smaller ones.

Another bad case would be an image with low-contrast colored balls. Our implementation uses thresholding to turn the grayscale, edge detected image into a binary image. If the pixel's intensity doesn't exceed a certain threshold, it will be lost. This is also the limitation and we'll need to decrease threshold for such cases.

- **Bad consant parameters**

We use the $\alpha = \frac{k\rho A}{2m}$ in our ODE system. This term is heavily dependent on constant parameters. If we make α very small, for example $\alpha = 0.01$, we often won't be able to find solutions. This is because the terms $\alpha\sqrt{u_2^2 + u_4^2}u_2$ and $\alpha\sqrt{u_2^2 + u_4^2}u_4$ contain α, so a change in speed will have a large effect on the ODE system. These are the results after setting $\alpha = 0.01$.

Target (178, 874, 40): v_{x0} = nan m/s, v_{y0} = nan m/s
Target (813, 742, 40): v_{x0} = nan m/s, v_{y0} = nan m/s
Target (613, 699, 40): v_{x0} = nan m/s, v_{y0} = nan m/s
Target (783, 598, 40): v_{x0} = nan m/s, v_{y0} = nan m/s
Target (850, 422, 40): v_{x0} = nan m/s, v_{y0} = nan m/s
Target (673, 259, 40): v_{x0} = 3814.689 m/s, v_{y0} = 196.853 m/s

Thus, choosing good constants is also very important.

- **Numerical method limitations**

In the solution, we approximate initial velocities using distances and time parameter t. I tried setting initial velocities to all ranges of values, but it still worked, Newton's method still finds the right solutions. Only case when Newton's method doesn't work is when $\left(v_x, v_y\right) = (0, 0)$. It fails because when J is a zero matrix, $\det(J) = 0$, J is a singular matrix and it doesn't have an inverse. Thus, choosing at least some $\vec{v}$ different from $\vec{0}$ is important.

Another problem is choosing a bad step size δt. If it's too large, the trajectory will be very unsmooth, with rough turns. This is not the problem of my implementation, but general problem of numerical methods that will arise if not considered. Result with $\delta t = 0.5$ is shown in *output/tests/dt_too_large.mp4*.

Conclusion: Given a good input image, the methods should work fine.