Optimizing Pocket Shots in Spikeball

To Improve Athlete Performance

Cooper Cole (20815895)
Daniel Raymond (20836965)
Phillip Shahviri (20827421)

Table of Contents

Abstract **1**
1. Introduction **1**
2. State of the Art Review **2**
3. Experimental Setup **4**
3.1. Equipment Setup 4
3.2. Shot Setup 6
4. Trajectory Analysis **6**
4.1. Data Collection 6
4.2. Ball Tracking 7
4.3. Data Analysis 7
5. Dynamics Modeling **9**
5.1. Formulation 9
5.2. Validation 10
6. Shot Optimization **10**
7. Results **11**
7.1. Experimental Observations 11
7.2. Simulation Results 13
7.3. Pocket Analysis 13
8. Conclusion **15**
8.1. Impact on the Sport 15
8.2. Future Improvements 15
8.3. Summary 16
9. Acknowledgements **16**
References **17**
Appendix **18**
Appendix A: Experimental Data 18
Appendix B: Ball Tracking Code 20
Appendix C: Collision Simulation Code 28

Abstract

While the sport of Spikeball has grown significantly in recent years, little has been done to research the dynamics of the sport. This report investigates the mechanics of "pocket shots" where the ball rebounds off the net back towards the player who hit it. Experimental data of real pocket shots was collected and analyzed. In performing analysis of the parameters that go into creating a pocket shot, we provided recommendations to players on how to be more successful in achieving pockets, as well as the required conditions for creating pocket shots with various optimal attributes. Given that Spikeball has nearly zero research papers in literature today, it is our hope that this research can serve as a starting point for understanding the dynamics at play in Spikeball, while also helping to increase the skill ceiling of the game.

1. Introduction

Spikeball is dynamic, fast-paced, and relatively new in the world of sports and recreation. Teams of two face off around a circular net placed at ankle level. See Figure 1 below of a typical gameplay environment [1].

Figure 1: Spikeball gameplay

A small ball is bounced off the net during gameplay. The objective is for one team to bounce the ball off the net such that the opposing team is unable to return it. Between bounces on the net, teams can make up to three passes by hitting the ball with any part of their body. The ball cannot touch the ground at any time during a team's possession, otherwise the point is lost.

Typically when the ball impacts the net, it rebounds up into the air predictably, with the direction of its horizontal velocity unchanged. However, on rare occasions the ball will contact the net and rebound backwards. This is called a "pocket shot". Figure 2 shows a pocket shot with the trajectory of the ball marked by a red line. In the frame shown, the ball is moving away from the net. One can see how the inbound and outbound trajectories are quite similar.

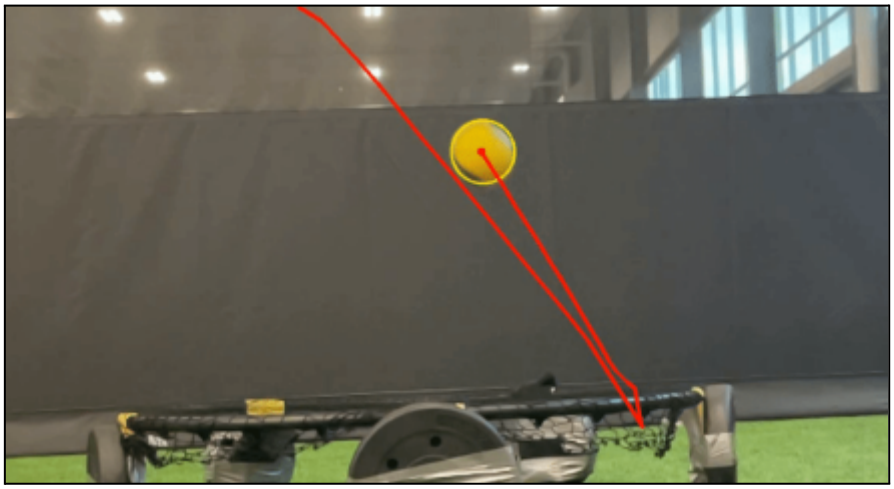

Figure 2: Trajectory of a pocket shot

Since teams are typically on opposite sides of the net, pocket shots can be extremely difficult to return. Our goal for this project was to develop a model of the ball-net interaction during a pocket shot, perform optimization and analysis, and then use our findings to make recommendations to players to help them intentionally hit more pocket shots.

2. State of the Art Review

Looking into state of the art research, we did not find any papers or technologies pertaining directly to the sport of Spikeball. Since we are modeling a net, we explored papers that made attempts at modeling trampolines, since the Spikeball net is essentially a trampoline.

We first looked at a paper out of Germany by Martin Kraft titled "A simple approximation for the vertical spring force of the trampoline" [2]. This paper developed the equation in Figure 3 below for the vertical spring force of the trampoline.

$$Fv = \frac{-D.s.[l_0 + \sqrt{(s^2 + b^2)} - b]}{\sqrt{(s^2 + b^2)}}$$

Where: s = vertical depression of the bed

D = stiffness of the springs

l_0 = stretch of the springs at equilibrium

b = distance from the frame to the nearest point of action

Figure 3: Trampoline vertical spring force [2]

This model is different from others seen in literature in that it takes into account the distance between the point of impact and the frame of the trampoline. This is relevant to pocket shots in Spikeball because the rigid rim of the net is critical in creating the conditions necessary for a pocket. With that being said, the major limitation of this model is that it only allows for calculation of the vertical component of force, whereas a pocket shot involves significant horizontal forces as well.

The second paper we looked at was "Determining and modeling the forces exerted by a trampoline suspension system" by Helen Phillippa Jaques in the UK [3]. This paper outlined modeling a gymnastics trampoline as a system of linear springs and point masses based on experimental measurements, as shown in Figure 4. The paper found that vertical force-displacement is non-linear, horizontal force-displacement is linear, and horizontal force is dependent upon vertical displacement. This model was helpful in showing how springs can be used to model a trampoline, but like in [2], this model only uses vertical impact forces on the trampoline, which does not align with our needs.

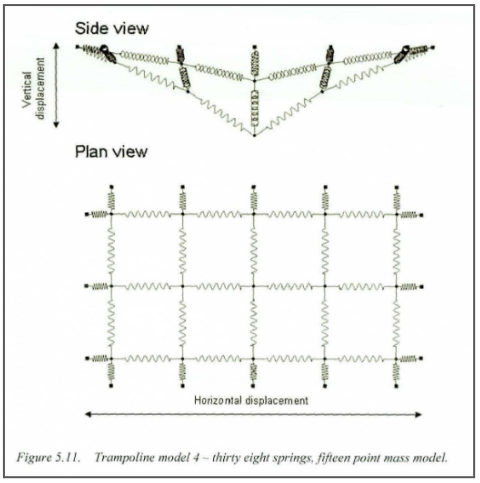

Figure 4: Gymnastics net modeling [3]

3. Experimental Setup

3.1. Equipment Setup

Data was collected at the UW Field House using standard Spikeball equipment. We set up a slow motion camera at a fixed distance from the net and filmed shots from both sides of the net. The global reference frame used for our data collection and analysis is shown in Figure 5.

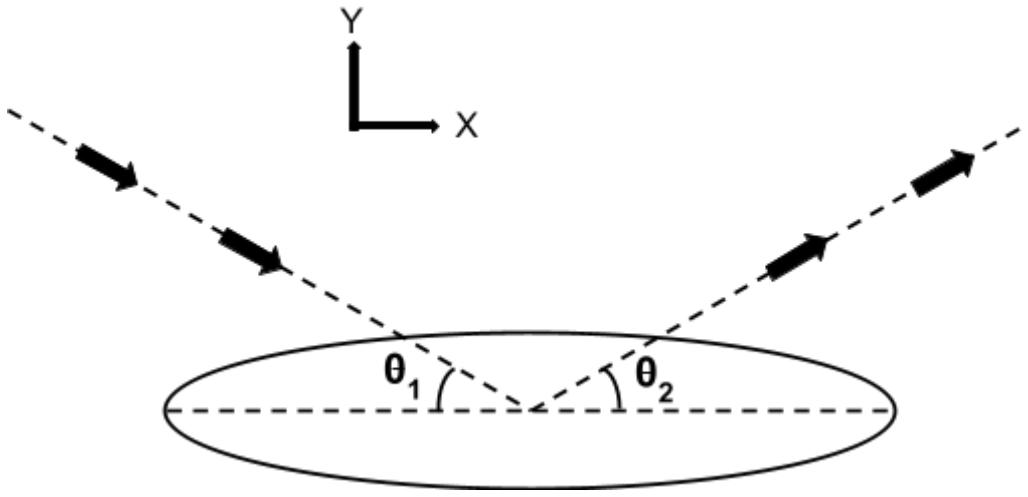

Figure 5: Global reference frame

Ball angles were always measured to be acute from the trajectory path to the horizontal surface of the net. Balls coming from the left had an inbound angle of θ_1 and an outbound angle of θ_2.

In real Spikeball games, the net tends to move slightly with each shot, since it is not rigidly secured to the ground. In our experimental setup, we attached weights to the legs of the net so that it did not move during gameplay. This greatly simplified our post processing and analysis of the data because our net was fixed in place within the global reference frame.

Using engineering knowledge and intuition, we identified parameters that would affect the results of our data collection. Table 1 below summarizes these parameters.

Table 1: Experimental parameter summary

Parameter	Variable/Constant	Value
Ball Pressure	Constant	2 psi
Net Tension	Constant	~327 N/m
Ball Incoming Angle	Variable	26° - 57°
Ball Incoming Speed	Variable	6m/s - 22 m/s

Ball pressure was determined using an air pump with an analog readout immediately before data collection began. The pressure was measured again after data collection was complete to ensure that the value did not change significantly.

A simple approximation of the tension on the net can be made by modeling the net as a spring and calculated using Hooke's law given by:

$$F = kx$$

Where F is the applied force, x is the displacement of the spring, and k is the spring constant. A 2 kg (19.62 N) mass was placed in the center of the net and the displacement was measured to be 0.06m. Using Hooke's law, a rough approximation for the spring constant of the net is 327 N/m. There are two reasons why this is only a rough approximation and not an exact value. Firstly, the displacement of the net would vary depending on the area of the mass placed on top of it. Our mass was quite small compared to the net (about 1⁄5 the radius), however this is still far from a point mass which would be more optimal for this measurement. Secondly, during normal gameplay the net is not guaranteed to always be operating in its linear region where Hooke's law is applicable.

3.2. Shot Setup

For the incoming ball angles, we varied the angle to create 3 categories of shots: high (~50°) medium (~40°), and low angle (~30°). Since this parameter was human-controlled, there was still variation between shots from the same category. The incoming ball speed was also varied, creating 2 categories of shots: fast (~20 m/s) and slow shots (~10 m/s). Again there was still some significant deviation in the speed of these shots due to human error. For each shot speed, several shots were completed at each angle, creating a total of six different categories of shots to analyze. Note that various shot categories were created simply to have a wider range of data. The fact that there is variation within each category does not make the collected data any less useful. The ball was still adequately tracked for almost all shots and velocity parameters could still be used for model fitting and validation. Also note that for every shot we were actively trying to create pocket shots. During normal gameplay pocket shots occur quite infrequently, meaning that it would take a very long time to gather a dataset of pocket shots with good variation. In trying to hit a pocket shot on each attempt, we were quickly able to get a dataset with a good mix of pocket and non-pocket shots for each shot type.

4. Trajectory Analysis

4.1. Data Collection

To collect the data, we recorded video of the three different categories of shots (high, medium, low angles) at fast and slow shot speeds. The videos were shot at 240 fps with consistent lighting to create the best conditions for the ball tracking software. The videos were sorted into their 3 respective categories, then filtered into “normal”, “vertical” and “pocket” shots, with rim shots and misses being discarded. “Normal” shots were defined as shots that hit the net and exited predictably in the same direction, i.e. roughly equal incoming and outgoing angles. “Vertical” shots were defined as when the ball bounced straight up into the air, ± 5° of the Y-axis. “Pocket” shots were defined as when the ball rebounded back the way it came.

4.2. Ball Tracking

To analyze the sorted videos for data collection, OpenCV in Python was used to read and analyze the video frame by frame [4]. To effectively track the ball, the code applies a

Gaussian blur to the frame, then converts the frame to HSV colour space, and constructs a mask for the HSV values of the tracked object; in our case the yellow Spikeball. A small script was written to extract the upper and lower limits of the HSV values of the ball. The mask has an "opening" image processing operation done on it to eliminate the small undesired blobs of yellow while maintaining the ball blob size. The minimum enclosing circle of the blob is identified using built-in OpenCV functions, then the centroid of the circle is calculated to identify the center of the ball. See Figure 6 below. The centroid X and Y points are then appended to an array of tracked points, frames with no tracked ball coordinates had their coordinates interpolated. The spin of the ball is not tracked to reduce complexity of our analysis. We focused on ball velocity, incoming and outgoing angles, and net impact and exit locations.

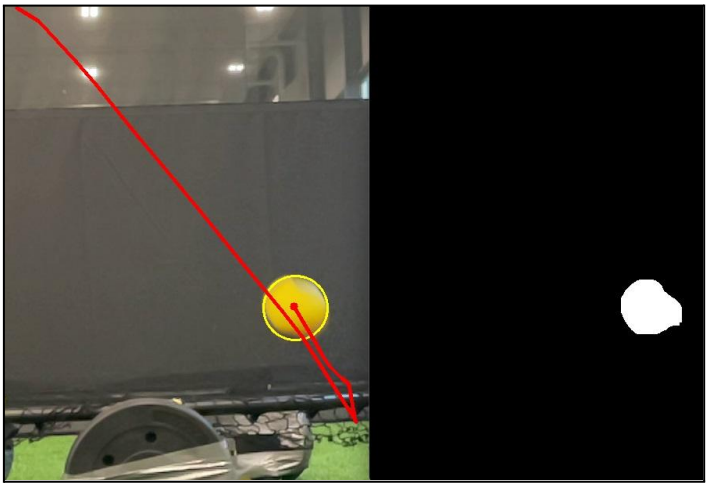

Figure 6: Tracked frame and mask

4.3. Data Analysis

With the array of tracked points, the trajectory of the ball is split into two arrays: incoming and outgoing; using the lowest Y-value to split them. These arrays can be plotted in a scatter plot of X and Y coordinates. These points are then fit to a parabola and plotted to obtain a smooth trajectory, and can be seen in Figure 7. The ball incoming and outgoing X and Y velocities are calculated using the first difference on the fit data. The incoming and outgoing angles are also calculated on the fit data.

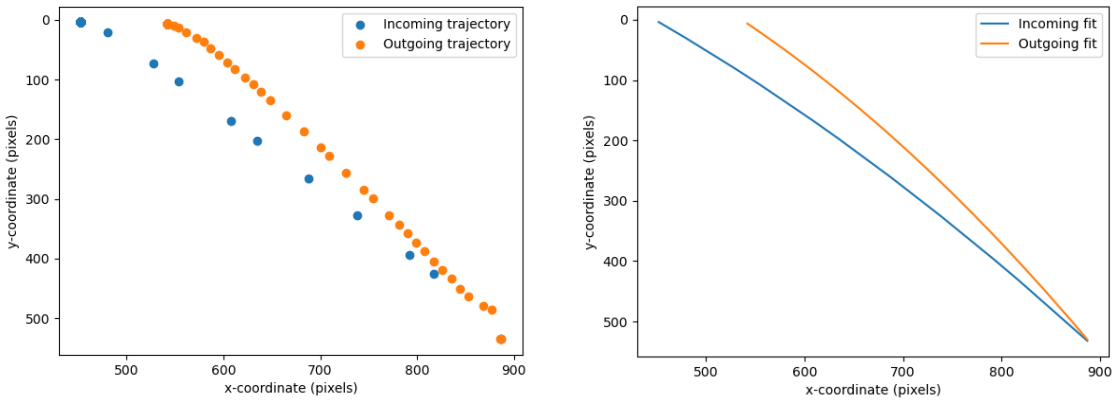

Figure 7: Incoming and outgoing trajectories

The ball impact and exit locations on the net were desired to fit the model. To extract these locations, a script was written to obtain the Y-value of the net, when the ball Y-location exceeded this value, the previous frame was shown. In that frame, the user manually selected the centroid of the ball and the inside edge of the rim; the impact X-distance was saved. For the exit location, the frame prior to the ball Y-location passing back over the Y-value of the net was shown and the user again selected the ball centroid and inside edge of the rim to save the exit X-distance. See Figure 8.

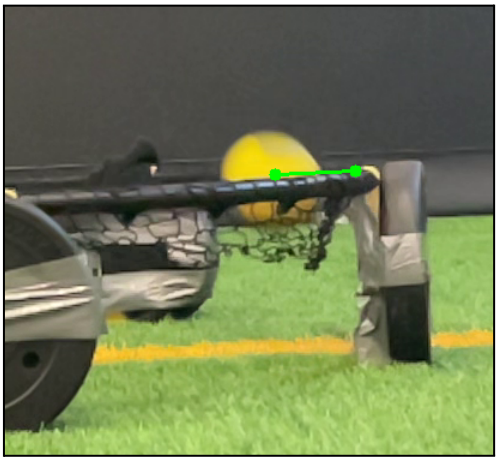

Figure 8: Determining ball impact location

These operations were done on all recorded shots by iterating through the folders of videos, and writing all the data to a CSV. A total of 73 data points were obtained for analysis and use on the model. For more detail, see Appendix A for the raw data and Appendix B for the code.

5. Dynamics Modeling

5.1. Formulation

The net is modeled as a soft body mesh with Hookean shear and tensile forces acting between each component. This allows the simulated net to deform under an applied load. The ball is modeled as a rigid sphere. While the ball does deform on impact, the deformation is negligible compared to the net. To model the rim, the top edge of circular mesh is pinned in place, acting as a boundary condition. To simplify the model, we assume the mesh net and the ball are the only components that are able to move. See Figure 9.

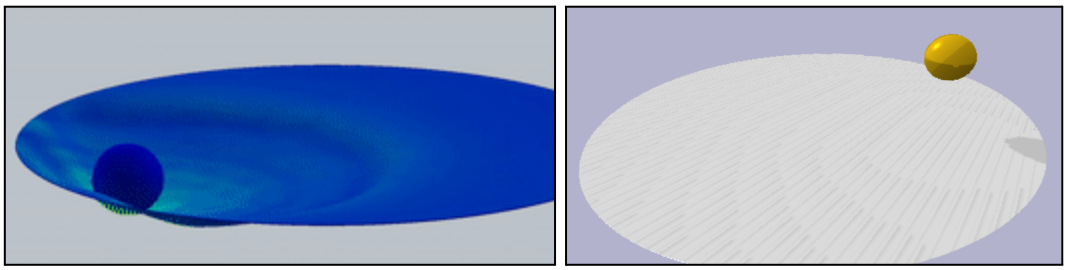

Figure 9: Simulated impact in SolidWorks vs PyBullet

Theoretically, the scenario could be modeled in two dimensions. The net follows a catenary curve, similar to a chain between two posts [5]. However, there are a few factors that complicate this model. The net is elastic, meaning that its length changes based on the applied load. Secondly, analytic catenary solutions are static. They typically assume the only force is the weight of the net itself, or at most a static general force. Solving the catenary equation over time results in a system of partial differential equations (PDEs), dramatically increasing complexity. Due to the limitations of numerically integrating PDEs and the complex boundary conditions, the 3D collision library *PyBullet* is used instead [6]. Various modeling techniques were employed to predict the outbound ball state, but this model proved the most reliable. Initially, *SolidWorks* was used for the collision simulation, but was soon disregarded as it could not easily be scripted to optimize pocket shots. Instead, *SolidWorks* was used to generate the mesh used in the *PyBullet* simulation. See Appendix C for the code.

5.2. Validation

To ensure the net model reliably simulated pocket shots, the error between the simulation and experiments was minimized. For each experiment, the inbound ball state, $\vec{x}_{in}$, was used as

the initial condition the model was run. Once the ball lost contact with the net, the ball state, $\vec{x}_{sim}$, was recorded. Due to experimental limitations, only the position and translational velocity of the ball were considered.

$$\vec{x}_{sim} = f(\vec{x}_{in}), \vec{x} = \begin{bmatrix} x \\ v_x \\ v_y \end{bmatrix}$$

By comparing the mean squared error of the simulation to the measured outbound ball states, the model was given a scalar error score.

$$error = \sum_j (f(\vec{x}_{in_j}) - \vec{x}_{out_j})^2$$

This error will vary depending on the assumed parameters of the net, such as the bending and tensile stiffness. Thus the error can be thought of as a function of these parameters.

$$error = g(k_{tensile}, k_{bending}, ...)$$

Then, classical optimization techniques were applied to this error function to improve the realism of the model. Unfortunately, due to limitations in *PyBullet's* softbody dynamics engine, the spring constants cannot be changed [7]. The mass of the net had to be used as a substitute parameter for model optimization. This reduces the generalizability of the model, but is considered an adequate replacement as the net will deform less for a larger mass, similar to having large spring constants.

6. Shot Optimization

There is no single optimal pocket shot; different situations call for different rebound angles and speeds. Therefore, we have optimized multiple different objective functions. In this analysis we defined three objective functions: fastest horizontal rebound, shortest airtime, and shallowest rebound angle. A fast horizontal rebound is useful when the opponent is defending against drop shots. A fast rebound will clear the ball away from the net quickly, forcing them to reposition. Short airtime gives the least time for the opponent to react to the shot. However, the optimal pocket drop shot will just clear the net, meaning the opponent might not have to move much to defend it. We expect a shallow rebound angle to be a balance between the two other objective functions. A shallow angle reduces air time, but doesn't necessarily make the ball land close to the net.

The feasible region of possible shots are constrained by the rules of the sport. First, the ball must bounce at exactly once. This prevents the inbound ball from being positioned beyond the rim of the net. To ensure the ball does not bounce more than once, the outbound velocity of the ball was used to calculate its trajectory. If this path intersects with the net, the shot is outside the feasible region. Second, the ball cannot hit the rim; this further reduces the inbound ball position. Finally, we limited the maximum inbound speed of the ball to 45 m/s due to limitations in the simulation.

At first, differential optimization methods were used. However, due to the highly nonlinear simulation, the optimization failed to converge. As a result, we opted to use genetic algorithms, which randomly sample the input space and combine promising results to narrow its search.

7. Results

7.1. Experimental Observations

A total of 73 valid shots were analyzed across the six shot types defined previously. It is important to remember that we as participants were actively trying to hit pocket shots on every single shot attempt. Table 2 below shows the percentage of shots that resulted in pockets for each shot type.

Table 2: Percentage of shots resulting in pockets for various shot types

	Shot Speed	
Shot Angle	**Fast**	**Slow**
Low	0%	17%
Medium	29%	50%
High	71%	43%

In an effort to explain the results above, it is helpful to first discuss the “pocket region”. The pocket region is a ring just inside the rim of the net made from 2 concentric circles. An illustration of this is shown in Figure 10 below.

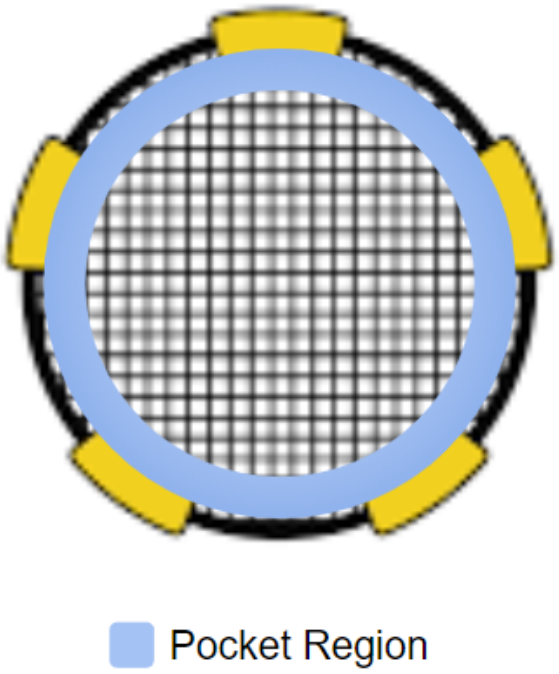

Figure 10: Experimental pocket region (approximate)

What we observed experimentally is that balls which contact the net in this region are more likely to be pocket shots. More discussion on this will follow in the Pocket Analysis section, where a precise definition of the pocket region will be formulated.

From Table 2 we see that fast shots at high angles had the highest success rate by far. After seeing this result, we hypothesized that it may be because the size of the pocket region increases at these higher angles and faster speeds, making pockets more likely. Intuitively, it made sense for impacts of this nature to have more margin of error in their position on the net while still being able to create the depression near the rim required for a pocket shot.

Another observation is that at low angles, pocket shots were very difficult to achieve in general. We believe that this is due to the size of the pocket region decreasing at lower angles. We found that it was incredibly difficult to place the ball in the correct location and create a pocket shot without the ball simply hitting the rim or bouncing off the net like a normal shot.

One insight seen from the data is that for medium angles, slow shots were actually preferable to fast shots. This is the opposite of what is seen for high angles, where faster shots were the most successful. A possible explanation for this is that when attempting medium shots, we

were simply more accurate when throwing the ball at slower speeds, hitting the pocket region a larger percentage of the time.

7.2. Simulation Results

After optimizing for the various objective functions, the following shots were found, as seen in Table 3.

Table 3: Optimal Pocket Shots

Criterion	**Shot Parameters**			**Optimum**
	cm from rim	*m/s right*	*m/s down*	
Fastest Horizontal Rebound	11.9	14.90	39.19	22 m/s left
Shallowest Rebound Angle	11.7	14.38	39.53	48°
Shortest Airtime	18.35	0.11	11.4	0.935 seconds

The fastest horizontal rebound and the shallowest rebound angle are remarkably similar, both requiring a fast shot around 60 degrees. This similarity is expected, since a shallow rebound will result in more kinetic energy horizontally. The optimal drop shot is almost entirely vertical; this imparts the least energy into the ball while still clearing the net. While we expect this type of shot results in the shortest airtime, uncertainty remains. The optimal duration discovered is quite long for drop shots, leading us to believe that the optimum has not yet been reached. Further investigation is required to determine the optimal pocket drop shot.

7.3. Pocket Analysis

Mathematically, the pocket region is defined as the set of points on the net the ball can initiate contact with that result in a pocket shot. As the model is radially symmetric, this can be viewed as a set of scalars, denoting a radial distance from the center of the net. Further, the assumption is made that this set is connected; if two radii result in a pocket, then all radii between will as well. Thus, the “pocket size” is the distance between the smallest and largest radius (see Figure 10).

From our experiments, we determined that the pocket size is dependent on the inbound shot velocity. Using the net model, we swept through all possible inbound shot parameters and

identified which shots result in pockets. By varying the impact position for each inbound velocity, the size of the pocket region can be measured. Figure 11 is the result of over 10 thousand simulated shots.

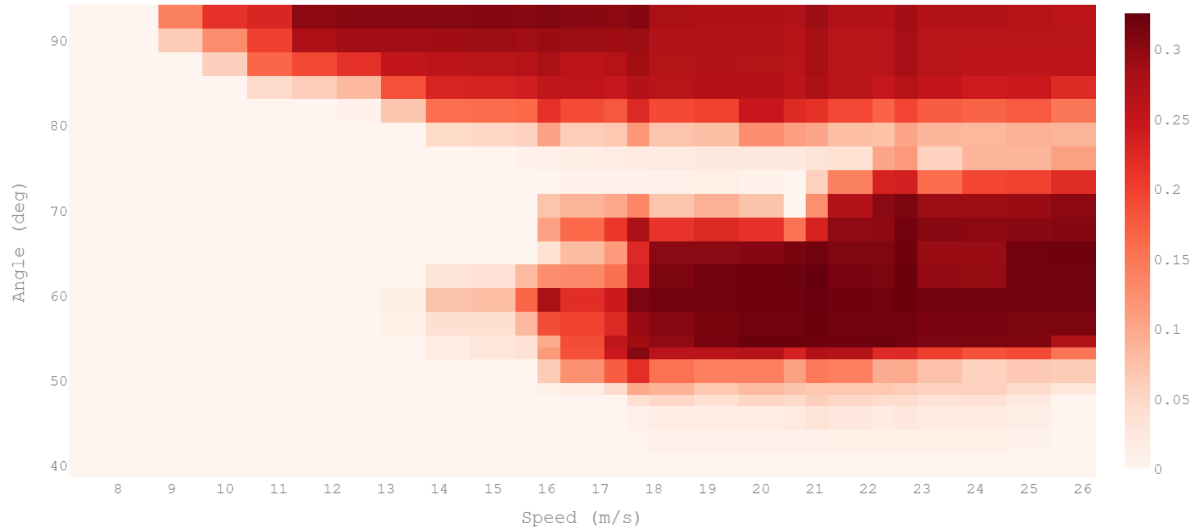

Figure 11: Pocket Size (m) vs Inbound Shot Speed & Angle

This heat map shows the types of shots that can lead to a pocket. The darker the region, the larger the pocket size. Notably, there are two predominant distributions. High angle shots near 90 degrees result in pockets at most speeds. This is reasonable, as there is not much horizontal kinetic energy to overcome. Since the ball must have sufficient energy to clear the net, there is a minimum inbound speed of ~9 m/s. Lower angle shots, around 65 degrees, require higher speeds in order to cause a pocket. This is reasonable as well, since the ball must bury into the net for it to sufficiently deform. This is a more traditional pocket that is more commonly seen in game; there is less use for a player to hit the ball directly downwards while over the net, as the opponent is already anticipating a change in direction. For traditional pocket shots, the ball must be at a minimum of ~15 m/s and ~45 degrees. Based on the overall distribution, we found a shot of 23.3 m/s at a 65 degree angle had the largest pocket region, at nearly ⅓ of the net diameter.

Interestingly, shots around 80 degrees seem to have a much smaller pocket region. Likely, this is because there is not enough horizontal energy to create a pocket, but too much to be easily overcome by the natural slope of the net. However, this could be due to simulation error; we did not perform any shots at these high angles to validate this claim. These results

do not perfectly line up with experiments, where numerous shots around 12 m/s and 50 degrees resulted in pocket shots. This suggests that the simulated net was more stiff than desired. However, we believe the distribution of pocket region sizes to be accurate.

8. Conclusion

8.1. Impact on the Sport

The results of this study have both practical implications and suggestions for athletes. The model that has been developed can be used for simulating different shots, doing "what-if" analysis to observe the outcomes of unique combinations of velocity, angle, and impact locations. Additionally, the model can be used for net design by tuning the net parameters to test different tensions and dimensions. For athletes, especially beginners, it is most important to understand that the pocket region increases for either increase in incoming angle, or incoming velocity. Hitting high angle, high velocity shots to increase your chances of hitting a pocket shot. Furthermore, avoid trying to hit a pocket shot at lower incoming angles, as it requires greater accuracy to hit the smaller pocket region. At the minimum required velocity for a pocket shot, this region becomes very difficult to hit, and is not worth the risk of missing the net or hitting the rim.

8.2. Future Improvements

Many improvements can be made to improve the simulation. First, the physics framework *PyBullet* should be replaced. Softbody mechanics aren't fully developed within the library, leading to occasional glitches between the ball and the mesh. This resulted in some simulated shots being incorrectly labeled as a pocket. Second, in future experiments we hope to collect information about the ball spin by using a camera with a higher framerate. This will allow us to fit the model to the experiments more accurately, as spin can cause large differences in the outbound ball state. Finally, we hope to implement our analysis in three dimensions. This will allow us to consider a wide variety of shots. "Side pockets", which cause the ball to veer left or right, are a large part of the sport which has not been thoroughly analyzed.

8.3. Summary

Overall, this study on Spikeball's pocket shots provides significant insights into the dynamics of the game and practical strategies for players. Future enhancements to the simulation and experimental setup could further refine these findings and expand the understanding of pocket shots in various playing conditions. We hope that this research serves as a starting point for understanding the dynamics at play in Spikeball.

9. Acknowledgements

We would like to acknowledge the contributions of Professor John McPhee. We are grateful for his insights and expertise which helped direct us in the early stages of this project. We would also like to thank the staff at Columbia Ice Field for allowing us to collect data in their facility and lending us a Spikeball net.

References

[1] “Spikeball,” spikeball.com. Available: https://spikeball.com/pages/about-us-spikeball. Accessed: Dec. 14, 2023.

[2] M. Kraft, "A simple approximation for the vertical spring force of the trampoline," Jul. 2001.[Online].Available:https://leopard.tu-braunschweig.de/servlets/MCRFileNodeServlet/dbbs_derivate_00001214/Document.pdf

[3] H. P. Jaques, "Determining and modeling the forces exerted by a trampoline suspension system," 2008. [Online] Available: https://scholar.google.ca/scholar?hl=en&as_sdt=0%2C5&as_vis=1&q=jacques+2008+trampoline&btnG=

[4] A. Rosebrock, "Ball Tracking with OpenCV," September 14, 2015. [Online]. Available: https://pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/. Accessed: Nov. 1, 2023.

[5] E. J. Routh, "On Strings," in *A Treatise on Analytical Statics*, Cambridge, UK: University Press, 1891, pp. 364–374. [Online]. Available: https://books.google.ca/books?id=3N5JAAAAMAAJ&pg=PA315&redir_esc=y#v=onepage&q&f=false. Accessed: Nov 1, 2023.

[6] PyBullet. [Online]. Available: https://pybullet.org/wordpress/. Accessed: Nov. 15, 2023

[7] Bullet Physics, Bullet 3 Issues. [Online]. Available: https://github.com/bulletphysics/bullet3/issues/2388. Accessed: Nov. 20, 2023.

Appendix

Appendix A: Experimental Data

Table A.1: Experimental Ball Trajectory Data

vx_in	vy_in	vx_out	vy_out	x_in	x_out	type_in	type_out
-7.54	8.49	-1.42	-6.87	-24.38	-11.52	high	normal
13.15	16.73	0.32	-11.96	23.04	15.77	high	normal
-15.84	16.20	-1.90	-6.25	-24.61	-11.63	high	normal
-8.09	9.05	-3.23	-8.35	-25.17	-13.42	high	normal
-14.77	17.00	-1.46	-4.56	-23.83	-10.85	high	normal
-8.75	10.62	-2.90	-9.62	-21.36	-10.51	high	normal
-6.42	8.65	-1.29	-8.49	-22.71	-11.86	high	normal
-7.52	9.41	-1.12	-7.17	-20.47	-9.62	high	normal
11.54	13.78	-1.40	-7.05	13.20	8.39	high	pocket
7.40	9.43	-5.96	-6.08	6.71	10.63	high	pocket
-7.20	8.79	1.85	-8.61		-8.84	high	pocket
10.21	11.24	-1.72	-6.40			high	pocket
13.96	18.05	-1.10	-5.70	12.64	11.63	high	pocket
7.25	9.27	-1.98	-8.93	12.86	10.96	high	pocket
13.69	15.96	-2.40	-4.27	5.26	8.05	high	pocket
13.42	16.93	-0.97	-6.27	13.76	13.42	high	pocket
11.71	14.16	-2.99	-7.40	6.94	8.95	high	pocket
12.08	15.20	-1.95	-4.19	6.94	9.73	high	pocket
12.89	15.94	-0.89	-4.39	11.97	10.07	high	pocket
8.76	11.42	-3.17	-8.73	8.50	9.17	high	pocket
11.28	15.12	-1.39	-4.26	8.84	10.63	high	pocket
-7.83	9.36	2.06	-9.01	-10.51	-8.50	high	pocket
7.55	10.48	-2.82	-5.03	9.62	11.19	high	pocket
-12.44	12.61	0.25	-18.26	-13.65	-8.39	high	vertical
13.69	16.87	0.23	-6.74	19.24	14.43	high	vertical
-13.96	14.53	0.44	-4.91	-16.00	-7.83	high	vertical
13.69	15.97	-0.45	-5.91	14.54	10.74	high	vertical
-12.89	15.38	-0.60	-4.10	-18.90	-12.98	high	vertical
12.62	13.78	-0.48	-5.42	14.77	10.63	high	vertical
-7.52	9.49	-0.65	-16.51	-20.25	-9.73	high	vertical
6.44	10.02	0.39	-4.47	26.17	20.02	high	vertical
13.15	16.61	-0.08	-6.66	17.79	12.53	high	vertical
-6.71	8.26	-0.33	-7.03	-19.24	-10.51	high	vertical
11.54	14.96	-0.92	-7.11	15.88	12.75	high	vertical
11.54	14.96	-0.92	-7.11			high	vertical
-6.98	5.70	-1.30	-5.63	-17.90	-6.82	medium	normal

-6.39	5.44	-1.17	-3.21	-23.04	-11.41	medium	normal
-11.85	10.57	-1.36	-3.44	-17.79	-8.84	medium	normal
-6.23	6.15	-2.40	-5.58	-22.71	-10.85	medium	normal
-15.12	14.10	-1.33	-5.75	-19.69	-9.96	medium	normal
-15.03	14.42	-4.23	-8.34	-28.19	-11.97	medium	normal
7.31	7.13	1.87	-8.57	23.83	15.44	medium	normal
-16.11	13.23	-1.32	-5.78	-19.46	-8.72	medium	normal
-9.30	8.97	-1.61	-9.08	-21.03	-9.73	medium	normal
7.55	7.80	2.63	-7.54	22.93	14.77	medium	normal
8.87	8.14	0.11	-5.79	16.33	9.84	medium	normal
-8.73	7.79	1.13	-8.15	-7.94	-5.37	medium	pocket
9.81	9.93	-1.19	-4.39	10.29	7.38	medium	pocket
7.61	8.79	-1.75	-2.52	5.82	8.28	medium	pocket
15.46	12.57	-0.99	-5.60	11.30	7.72	medium	pocket
-8.13	7.27	2.54	-5.78	-12.30	-10.74	medium	pocket
8.55	7.26	-1.37	-7.56	10.74	7.38	medium	pocket
7.75	7.46	-3.01	-5.99	11.07	11.41	medium	pocket
9.52	8.06	-2.49	-6.55	9.06	6.71	medium	pocket
16.38	15.71	-2.54	-5.02			medium	pocket
-7.38	6.14	-0.26	-8.94	-16.55	-9.40	medium	vertical
9.53	9.83	-0.12	3.08	13.98	7.83	medium	vertical
6.31	7.45	-0.11	3.36	14.43	10.07	medium	vertical
9.37	10.31	-0.29	-9.25	16.67	9.51	medium	vertical
4.77	5.94	-0.71	-8.53	12.53	8.05	medium	vertical
5.06	6.19	0.11	7.33	15.21	9.17	medium	vertical
9.88	9.74	1.28	-8.37	15.44	10.40	medium	vertical
8.45	8.86	1.13	-8.68	17.45	8.72	medium	vertical
-13.16	11.36	-0.59	-6.64	-18.34	-7.83	medium	vertical
-8.41	7.51	-0.37	-3.58	-15.44	-7.94	medium	vertical
8.67	7.43	0.56	-7.77	14.77	8.84	medium	vertical
4.79	3.63	0.73	-4.00	10.74	6.26	low	normal
9.48	7.09	1.04	-5.11	15.88	7.72	low	normal
-10.39	7.16	-2.62	-3.36	-27.29	-10.51	low	normal
-16.91	10.03	-6.91	-3.33			low	normal
7.73	4.00	4.42	-3.64			low	normal
-8.96	5.09	-7.23	-5.47	-40.27	-14.65	low	normal
14.25	7.23	3.63	-3.21			low	normal
-15.70	8.51	-3.87	-3.64	-32.44	-6.04	low	normal
16.65	8.60	2.90	-3.75			low	normal
-9.98	5.98	-7.88	-4.92	-48.43	-21.81	low	normal
9.14	5.05	4.66	-9.23	19.69	7.05	low	normal

-14.70	9.36	-3.28	-2.78	-32.33	-10.96	low	normal
14.47	7.55	3.88	-6.85	18.46	5.93	low	normal
-16.82	9.96	-4.91	-6.68	-26.62	-7.83	low	normal
-8.28	5.69	0.91	-3.57	-10.63	-6.38	low	pocket
5.05	3.61	0.13	-1.69			low	vertical

Appendix B: Ball Tracking Code

```
import numpy as np
import pandas as pd
import cv2
import imutils
import time
import csv
import matplotlib.pyplot as plt
import os
from scipy.optimize import curve_fit


root_video_folder = 'project_videos'

# Initialize empty lists to store video files for each category
high_videos = {'Normal':[], 'Pocket':[], 'Vertical':[]}
medium_videos = {'Normal':[], 'Pocket':[], 'Vertical':[]}
low_videos = {'Normal':[], 'Pocket':[], 'Vertical':[]}

# list of tracked points
tracked_pts = []


CROPPED_NET_Y_COORD = 350 # The y pixel coordinate of the cropped frame
ORIGINAL_NET_Y_COORD = 580
PIXELS_PER_METER = 894 # Points (546, 591) and (1291, 582) were selected on original image. x-diff is 745px
/ 3 ft => 894px / 1 m
RESIZED_IMG_PIXELS_PER_METER = 255.84 # Points (170, 185) and (404, 181) were selected on resized image.
x-diff is 234px / 3 ft => 255.84px / 1 m
MAX_START_Y_COORD = 100 # If the y values are higher than this at the start of the tracked points then
discard them

SHOW_PLOTS = False
PRINT_DATA = True
WRITE_DATA_TO_CSV = True
BALL_RADIUS_M = 0.04445
BALL_RADIUS_PX = BALL_RADIUS_M*PIXELS_PER_METER

START_Y_CROP = 230
END_Y_CROP = 686
START_X_CROP = 432
END_X_CROP = 1432
# Min HSV value in ROI: [20 42 91]
# Max HSV value in ROI: [ 30 255 248]
# define the lower and upper boundaries of the "yellow" ball in the HSV color space

# yellowLower = (20, 80, 100)
# yellowUpper = (30, 255, 248)


# TEST VALUES
yellowLower = (20, 125, 100)
```

```
yellowUpper = (30, 255, 248)

# Min HSV value in ROI: [ 21 214 202]
# Max HSV value in ROI: [ 25 255 245]


def get_video_files():
    # Iterate through the master folder
    for root, dirs, files in os.walk(root_video_folder):
        # Split the path into components
        path_components = root.split(os.path.sep)

        # Check if the path has enough components to identify category and subcategory
        if len(path_components) >= 3:
            _, category, subcategory = path_components[-3:]
            # Check if the current directory is a video subfolder
            if subcategory in ['Normal', 'Pocket', 'Vertical']:
                # Iterate through the files in the current subfolder
                for file in files:
                    # Check if the file is a video file (you may need to adjust this condition)
                    if file.endswith(('.mp4', '.avi', '.MOV')):
                        # Create the full path to the video file
                        video_path = os.path.join(root, file)

                        # Append the video file to the appropriate list based on the category
                        if category == 'high':
                            high_videos[subcategory].append(video_path)
                        elif category == 'medium':
                            medium_videos[subcategory].append(video_path)
                        elif category == 'low':
                            low_videos[subcategory].append(video_path)

    print("Videos in 'high':", high_videos)
    print("Videos in 'medium':", medium_videos)
    print("Videos in 'low':", low_videos)


def get_ball_hsv(frame):
        # Select ROI
    roi = cv2.selectROI(frame)
    # Crop image
    roi_cropped = frame[int(roi[1]):int(roi[1]+roi[3]), int(roi[0]):int(roi[0]+roi[2])]
    # Convert to HSV
    hsv_roi = cv2.cvtColor(roi_cropped, cv2.COLOR_BGR2HSV)

    # Calculate average HSV values
    average_color_per_row = np.average(hsv_roi, axis=0)
    average_color = np.average(average_color_per_row, axis=0)

    # Calculate min and max HSV values
    min_color = np.min(hsv_roi, axis=(0, 1))
    max_color = np.max(hsv_roi, axis=(0, 1))
    cv2.waitKey(0)
    cv2.destroyAllWindows()
    return min_color, max_color

# Track the ball
def track_ball(video, tracked_points, mask_lower, mask_upper, show_video=True):
    while True:
        # grab the current frame
        frame = video.read()
        # handle the frame from VideoCapture or VideoStream
        frame = frame[1] if video else frame
```

```
        # if we are viewing a video and we did not grab a frame,
        # then we have reached the end of the video
        if frame is None:
            break

        # crop the frame, blur it, and convert it to the HSV
        # color space

        frame = frame[START_Y_CROP:END_Y_CROP, START_X_CROP:END_X_CROP]
        blurred = cv2.GaussianBlur(frame, (11, 11), 0)
        hsv = cv2.cvtColor(blurred, cv2.COLOR_BGR2HSV)
        # construct a mask for the color "yellow", then perform
        # a series of dilations and erosions to remove any small
        # blobs left in the mask
        mask = cv2.inRange(hsv, mask_lower, mask_upper)
        mask = cv2.erode(mask, None, iterations=10)
        mask = cv2.dilate(mask, None, iterations=7)
        cv2.imshow("mask", mask)

        # find contours in the mask and initialize the current
        # (x, y) center of the ball
        cnts = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
            cv2.CHAIN_APPROX_SIMPLE)
        cnts = imutils.grab_contours(cnts)
        center = None

        # only proceed if at least one contour was found
        if len(cnts) > 0:
            # find the largest contour in the mask, then use
            # it to compute the minimum enclosing circle and
            # centroid
            c = max(cnts, key=cv2.contourArea)
            ((t, y), radius) = cv2.minEnclosingCircle(c)
            M = cv2.moments(c)
            center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
            # only proceed if the radius meets a minimum size
            if radius > 20:
                # draw the circle and centroid on the frame,
                # then update the list of tracked points
                cv2.circle(frame, (int(t), int(y)), int(radius),
                    (0, 255, 255), 2)
                cv2.circle(frame, center, 5, (0, 0, 255), -1)
        # update the points queue
        tracked_points.append(center)

        # loop over the set of tracked points
        for i in range(1, len(tracked_points)):
            # if either of the tracked points are None, ignore them
            if tracked_points[i - 1] is None or tracked_points[i] is None:
                continue
            # otherwise, compute the thickness of the line and
            # draw the connecting lines
            thickness = int(np.sqrt(512 / float(i + 1)) * 2.5)
            cv2.line(frame, tracked_points[i - 1], tracked_points[i], (0, 0, 255), thickness)

        if show_video:
            # show the frame to our screen
            cv2.imshow("Frame", frame)
            key = cv2.waitKey(1) & 0xFF
            if key == ord("q"):
                break
    return tracked_points
```

```
# Plot the trajectory
def interpolate_nones(A: np.ndarray):
    ok = ~np.isnan(A)
    xp = ok.ravel().nonzero()[0]
    fp = A[~np.isnan(A)]
    x  = np.isnan(A).ravel().nonzero()[0]
    A[np.isnan(A)] = np.interp(x, xp, fp)
    return A


def plot_trajectory(x1, y1, x2, y2, title1, title2):
    if x2 is None or y2 is None:
        plt.figure()
        plt.plot(x1, y1, label=title1)
        plt.gca().invert_yaxis()
        plt.xlabel('x-coordinate (pixels)')
        plt.ylabel('y-coordinate (pixels)')
        plt.legend()
        plt.show()
    else:
        plt.figure()
        plt.plot(x1, y1, label=title1)
        plt.plot(x2, y2, label=title2)
        plt.gca().invert_yaxis()
        plt.xlabel('x-coordinate (pixels)')
        plt.ylabel('y-coordinate (pixels)')
        plt.legend()
        plt.show()


def fit_line(x, y, order=1):
    coeffs = np.polyfit(x, y, order)
    fitted = np.polyval(coeffs, x)
    return fitted

def fit_parabola(x, y, order=2):
    coeffs = np.polyfit(x, y, order)
    fitted = np.polyval(coeffs, x)
    return fitted


def get_angle(x, y):
    # Calculate the slope of the line between the first and last points
    slope = (y[-1] - y[0]) / (x[-1] - x[0])
    # Calculate the angle in degrees
    angle = np.arctan(slope) * 180 / np.pi
    return angle

def plot_speeds(x, y, title1, title2):
    plt.figure()
    plt.subplot(2, 1, 1)
    plt.plot(x)
    plt.xlabel('Frame')
    plt.ylabel('Speed (m/s)')
    plt.title(title1)
    plt.subplot(2, 1, 2)
    plt.plot(y)
    plt.xlabel('Frame')
    plt.ylabel('Speed (m/s)')
    plt.title(title2)
    plt.tight_layout()
    plt.show()
```

```python
def fit_curve(velocity_x, velocity_y, order):
    # cubic fit
    coeffs_vx = np.polyfit(np.linspace(0, len(velocity_x), len(velocity_x)), velocity_x, order)
    coeffs_vy = np.polyfit(np.linspace(0, len(velocity_y), len(velocity_y)), velocity_y, order)
    # Calculate the fitted speeds
    fitted_vx = np.polyval(coeffs_vx, np.linspace(0, len(velocity_x), len(velocity_x)))
    fitted_vy = np.polyval(coeffs_vy, np.linspace(0, len(velocity_y), len(velocity_y)))
    return fitted_vx, fitted_vy

# Mouse callback function
def click_event(event, x, y, flags, params, click_coordinates, frame):
    # If the left mouse button was clicked, record the (x, y) coordinates
    if event == cv2.EVENT_LBUTTONDOWN:
        click_coordinates.append((x, y))

        # Draw a circle where the user clicked
        cv2.circle(frame, (x, y), 5, (0, 255, 0), -1)

        # If two points have been clicked, draw a line between them
        if len(click_coordinates) == 2:
            cv2.line(frame, click_coordinates[0], click_coordinates[1], (0, 255, 0), 2)

        # Display the image
        cv2.imshow('image', frame)


# Calculate the distance between two points
def get_ball_distance_to_edge(frame, return_type='x'):
    # Display the image and set the mouse callback function
    points = []
    cv2.imshow('image', frame)
    cv2.setMouseCallback('image', lambda *args: click_event(*args, click_coordinates=points, frame=frame))

    # Wait for a key press and then close the windows
    cv2.waitKey(0)
    cv2.destroyAllWindows()

    # If two points were clicked, calculate and print the distance between them
    if len(points) == 2:
        distance_to_edge = np.sqrt((points[1][0] - points[0][0])**2 + (points[1][1] - points[0][1])**2)
        x_distance = points[1][0] - points[0][0]
        y_distance = points[1][1] - points[0][1]
        if return_type == 'x':
            return x_distance
        elif return_type == 'y':
            return y_distance
        else:
            return distance_to_edge
    else:
        return -1


def pad_arrays(arrays):
    # Find the maximum length among the arrays
    max_length = max(len(arr) for arr in arrays)

    # Pad each array with -1s to match the maximum length
    padded_arrays = [np.pad(arr, (0, max_length - len(arr)), constant_values=-1) for arr in arrays]

    return padded_arrays

def get_distance_in_frame(cap, frame_index):
```

```
    # Skip to the frame at max_index
    try:
        cap.set(cv2.CAP_PROP_POS_FRAMES, frame_index)
    except Exception as e:
        print(f"Error setting frame index: {e}")
    # Read the frame at max_index
    ret, frame = cap.read()

    # Initialize the list of points
    ball_x_distance = np.round(get_ball_distance_to_edge(frame) / PIXELS_PER_METER * 100, 2)
    return ball_x_distance


def truncate_array_to_monotonic(arr, increment=1):
    result = [arr[0]]  # Initialize the result array with the first element

    for i in range(1, len(arr)):
        if arr[i] == result[-1] + increment:
            result.append(arr[i])
        elif arr[i] > result[-1] + increment:
            break  # Stop if the next value is greater than the expected increment

    return result

def trim_nones_from_tuples(array):
    non_none_indices = [i for i, value in enumerate(array) if (isinstance(value, tuple) and None not in
value) or value is not None]

    if not non_none_indices:
        return []  # Handle case where all elements are None or tuples with None

    start_index = non_none_indices[0]

    while (array[start_index] is not None and array[start_index][1] > MAX_START_Y_COORD):
        start_index += 1

    end_index = non_none_indices[-1] + 1

    return array[start_index:end_index]


get_video_files()

video_files = {'high_videos': high_videos, 'medium_videos': medium_videos, 'low_videos': low_videos}


for video_type_name, video_type in video_files.items():
    for shot_type, videos in video_type.items():
        for video in videos:
            # video capture object
            cap = cv2.VideoCapture(video)

            # allow the camera or video file to warm up
            time.sleep(2.0)
            # Get the frame rate
            frame_rate = cap.get(cv2.CAP_PROP_FPS)
            print(f'Frame rate: {frame_rate} fps')


            tracked_pts = track_ball(cap, tracked_pts, yellowLower, yellowUpper)
               # print tracked_pts
            if PRINT_DATA:
                print(f'tracked_pts: {tracked_pts}')
```

```
            # cap.release()
            # cv2.destroyAllWindows()
            # print number of not None elements in tracked_pts
            print(f"Number of tracked points: {len([pt for pt in tracked_pts if pt is not None])}")

            trimmed_tracked_pts = trim_nones_from_tuples(tracked_pts)

            x_coords = interpolate_nones(np.array([pt[0] if pt is not None else np.nan for pt in
tracked_pts]))
            y_coords = interpolate_nones(np.array([pt[1] if pt is not None else np.nan for pt in
tracked_pts]))


            # print x_coords and y_coords
            if PRINT_DATA:
                print(f'x_coords: {x_coords}')
                print(f'y_coords: {y_coords}')


            # Find the indicies of frames where the ball is in contact with the net
            impact_indicies = (np.argwhere(y_coords > CROPPED_NET_Y_COORD - BALL_RADIUS_PX))
            impact_indicies = impact_indicies.flatten()

            # Truncate to remove indicies where the ball was not found in frame
            impact_indicies = truncate_array_to_monotonic(impact_indicies, increment=1)
            impact_start_index = impact_indicies[0]
            print(f"Impact_start_index: {impact_start_index}")
            impact_end_index = impact_indicies[-1]
            print(f"Impact_end_index: {impact_end_index}")

            # Split x_coords and y_coords into incoming and outgoing arrays
            incoming_x = x_coords[:impact_start_index]
            incoming_y = y_coords[:impact_start_index]
            outgoing_x = x_coords[impact_end_index:]
            outgoing_y = y_coords[impact_end_index:]

            # print incoming_x, incoming_y, outgoing_x, and outgoing_y
            if PRINT_DATA:
                print(f'incoming_x: {incoming_x}')
                print(f'incoming_y: {incoming_y}')
                print(f'outgoing_x: {outgoing_x}')
                print(f'outgoing_y: {outgoing_y}')


            fitted_incoming_y = fit_parabola(incoming_x, incoming_y)
            fitted_outgoing_y = fit_parabola(outgoing_x, outgoing_y)

            # print fitted_incoming_y and fitted_outgoing_y
            if PRINT_DATA:
                print(f'fitted_incoming_y: {fitted_incoming_y}')
                print(f'fitted_outgoing_y: {fitted_outgoing_y}')


            incoming_angle = np.abs(np.round(get_angle(incoming_x, fitted_incoming_y), 2))
            outgoing_angle = np.abs(np.round(get_angle(outgoing_x, fitted_outgoing_y), 2))

            if PRINT_DATA:
                print(f"angle of incoming trajectory: {incoming_angle} degrees")
                print(f"angle of outgoing trajectory: {outgoing_angle} degrees")

            # calculate Velocities
            incoming_vx = np.trim_zeros(np.diff(incoming_x) * frame_rate / PIXELS_PER_METER)
```

```
            incoming_vy = np.trim_zeros(np.diff(fitted_incoming_y) * frame_rate / PIXELS_PER_METER)
            outgoing_vx = np.trim_zeros(np.diff(outgoing_x) * frame_rate / PIXELS_PER_METER)
            outgoing_vy = np.trim_zeros(np.diff(fitted_outgoing_y) * frame_rate / PIXELS_PER_METER)

            # print incoming_vx, incoming_vy, outgoing_vx, and outgoing_vy
            if PRINT_DATA:
                print(f'incoming_vx: {incoming_vx}')
                print(f'incoming_vy: {incoming_vy}')
                print(f'outgoing_vx: {outgoing_vx}')
                print(f'outgoing_vy: {outgoing_vy}')

            fitted_incoming_vx, fitted_incoming_vy = fit_curve(incoming_vx, incoming_vy, 5)
            fitted_outgoing_vx, fitted_outgoing_vy = fit_curve(outgoing_vx, outgoing_vy, 3)

            # print fitted_incoming_vx, fitted_incoming_vy, fitted_outgoing_vx, and fitted_outgoing_vy
            if PRINT_DATA:
                print(f'fitted_incoming_vx: {fitted_incoming_vx}')
                print(f'fitted_incoming_vy: {fitted_incoming_vy}')
                print(f'fitted_outgoing_vx: {fitted_outgoing_vx}')
                print(f'fitted_outgoing_vy: {fitted_outgoing_vy}')


            median_incoming_vx = np.median(fitted_incoming_vx[-5:])
            median_incoming_vy = np.median(fitted_incoming_vy[-5:])

            median_outgoing_vx = np.median(fitted_outgoing_vx[:5])
            median_outgoing_vy = np.median(fitted_outgoing_vy[:5])


            # Release and reopen the video file before getting the distance from ball center to rim edge
            cap.release()
            cap = cv2.VideoCapture(video)
            inbound_x_distance = get_distance_in_frame(cap, impact_start_index)
            print(f"Distance from ball to edge on impact: {inbound_x_distance} centimeters")
            cap.release()

            cap = cv2.VideoCapture(video)
            outbound_x_distance = get_distance_in_frame(cap, impact_end_index)
            print(f"Distance from ball to edge on exit: {outbound_x_distance} centimeters")

            cap.release()


            # print the fitted speeds
            if PRINT_DATA:
                print(f'Incoming fitted speed in x: {np.round(fitted_incoming_vx, 2)} m/s')

            if SHOW_PLOTS:
                # plot the trajectory
                plot_trajectory(x_coords, y_coords, None, None, 'Ball trajectory', None)
                # plot the incoming and outoging trajectories
                plot_trajectory(incoming_x, incoming_y, outgoing_x, outgoing_y, 'Incoming trajectory',
'Outgoing trajectory')
                # plot the incoming and outgoing fitted trajectories
                plot_trajectory(incoming_x, fitted_incoming_y, outgoing_x, fitted_outgoing_y, 'Incoming
fit', 'Outgoing fit')
                # Plot the speeds in x and y as separate sub plots
                plot_speeds(incoming_vx, incoming_vy, 'Incoming speed in x', 'Incoming speed in y')
                plot_speeds(outgoing_vx, outgoing_vy, 'Outgoing speed in x', 'Outgoing speed in y')
                # Plot the fitted speeds in x and y as separate sub plots
                plot_speeds(fitted_incoming_vx, fitted_incoming_vy, 'Incoming fitted speed in x', 'Incoming
fitted speed in y')
                plot_speeds(fitted_outgoing_vx, fitted_outgoing_vy, 'Outgoing fitted speed in x', 'Outgoing
```

```
fitted speed in y')

            if WRITE_DATA_TO_CSV:
                #data_arrays = [median_incoming_vx, median_incoming_vy, median_outgoing_vx,
median_outgoing_vy, [inbound_x_distance], [outbound_x_distance], [incoming_angle], [outgoing_angle]]
                #padded_arrays = pad_arrays(data_arrays)

                # Assuming fitted_incoming_vx, fitted_incoming_vy, ball_x_distance, incoming_angle, and
outgoing_angle are defined
                # ball_data = {
                #   'median_incoming_vx': padded_arrays[0],
                #   'median_incoming_vy': padded_arrays[1],
                #   "median_outgoing_vx": padded_arrays[2],
                #   "median_outgoing_vy": padded_arrays[3],
                #   'inbound_x_distance': padded_arrays[4],
                #   'outbound_x_distance': padded_arrays[5],
                #   'incoming_angle': padded_arrays[6],
                #   'outgoing_angle': padded_arrays[7]
                # }

                ball_data = {
                    'median_incoming_vx': [median_incoming_vx],
                    'median_incoming_vy': [median_incoming_vy],
                    "median_outgoing_vx": [median_outgoing_vx],
                    "median_outgoing_vy": [median_outgoing_vy],
                    'inbound_x_distance': [inbound_x_distance],
                    'outbound_x_distance': [outbound_x_distance],
                    'incoming_angle': [incoming_angle],
                    'outgoing_angle': [outgoing_angle]
                }


                #print("Hello {} {}, hope you're well!".format(first_name,last_name))

                # Create a DataFrame from the data
                video_name_df = pd.DataFrame({"{} {}".format(video_type_name, shot_type): [video]})
                ball_data_df = pd.DataFrame(ball_data)
                empty_df = pd.DataFrame(columns=range(8))

                # Write the DataFrame to a CSV file
                video_name_df.to_csv(video_type_name + '.csv', index=False, mode='a')
                ball_data_df.to_csv(video_type_name + '.csv', index=False, mode='a')
                empty_df.loc[0] = [None] * 8
                empty_df.to_csv(video_type_name + '.csv', mode='a', header=False, index=False)
                empty_df.to_csv(video_type_name + '.csv', mode='a', header=False, index=False)
                empty_df.to_csv(video_type_name + '.csv', mode='a', header=False, index=False)

                tracked_pts = []
```

Appendix C: Collision Simulation Code

Figure C.1: Simulation Environment

```
import numpy as np
import pybullet as p
from functools import cache
from PIL import Image
from time import sleep

np.seterr(invalid='raise')
```

```
NET_RADIUS = 0.42 # m
BALL_RADIUS = 0.04445 # m
NET_MODEL = "net.obj"
NET_EDGE_NODES = 520
GRAVITY = 9.81 # m/s^2
# NET_NODES = 5720


class SpikeBallSimulator:
    def __init__(self, *net_params, max_duration=0.1, g=GRAVITY, plot=False, trimetric=False, **net_kwargs)
-> None:
        self.step_rate = 20000
        self.max_steps = int(max_duration*self.step_rate)
        self.plot = plot
        p.connect(p.GUI if plot else p.DIRECT)
        p.setTimeStep(1/self.step_rate)
        # Camera settings
        cameraTargetPosition = [0, 0, 0]  # x, y, z
        cameraDistance = 0.56
        cameraYaw = 0
        cameraPitch = -30 if trimetric else 0
        p.resetDebugVisualizerCamera(cameraDistance, cameraYaw, cameraPitch, cameraTargetPosition)
        p.setGravity(0, 0, -g)
        # Net model
        self.contact_margin = 0.01
        # Ball model
        self.ball = p.createMultiBody(
            baseMass=0.150, # kg
            baseCollisionShapeIndex=p.createCollisionShape(p.GEOM_SPHERE, radius=BALL_RADIUS),
            basePosition=[0, 0, BALL_RADIUS],
            baseOrientation=p.getQuaternionFromEuler([0, 0, 0]),
        )
        self.net = None
        self.init_state = None
        self.update_net(*net_params, **net_kwargs)
        self.init_state = p.saveState()

    def update_net(self, mass=0.1, scale=1):
        if self.init_state is not None:
            p.restoreState(self.init_state)
        if self.net is not None:
            raise NotImplementedError("Updating net parameters is not yet implemented")
            p.removeBody(self.net)
        self.net = p.loadSoftBody(
            NET_MODEL,
            basePosition=[-0.5, 0.5, 0], # Center of the net is not at the origin
            baseOrientation=p.getQuaternionFromEuler([np.pi/2, 0, 0]),
            scale=0.0005*scale, # Model has a diameter of 1828.8 mm
            mass=mass,
            useSelfCollision=1,
            collisionMargin=self.contact_margin,
        )
        # Fix rim
        for nodeIndex in range(NET_EDGE_NODES):
            p.createSoftBodyAnchor(self.net, nodeIndex, -1, -1)  # Anchor to a fixed point in space
        self.init_state = p.saveState()

    def run(self, rim_contact_dist, vx, vy, min_steps=4, save=None, demo=False):
        p.restoreState(self.init_state)
        contact_height = BALL_RADIUS + self.contact_margin/2
        # Start ball from higher up if it's a demo, but so that it contacts the net at the same point
        if demo:
            # p.setGravity(0, 0, 0)
```

```
            contact_height += vy/70
            rim_contact_dist += vx/70
        p.resetBasePositionAndOrientation(self.ball, [NET_RADIUS - rim_contact_dist, 0, contact_height], [0,
0, 0, 1])
        p.resetBaseVelocity(self.ball, linearVelocity=[vx, 0, -vy], angularVelocity=[0, 0, 0])

        ball_coords = []
        frames = []
        sim_started = False
        step_count = 0
        if self.plot:
            p.setRealTimeSimulation(True)
        while p.isConnected():
            if self.plot:
                sleep(0.01)  # Time in seconds.
            else:
                p.stepSimulation()
            if save is not None:
                width, height, rgbImg, depthImg, segImg = p.getCameraImage(720, 480,
renderer=p.ER_BULLET_HARDWARE_OPENGL)
                rgb = np.array(rgbImg, dtype=np.uint8)
                rgb = np.reshape(rgb, (height, width, 4))[:, :, :3]
                # Crop height and width to remove bottom and right third
                rgb = rgb[:height*5//6, :width*4//5]
                frames.append(Image.fromarray(rgb, 'RGB'))
            step_count += 1
            # Get ball position
            (x, y, z), _ = p.getBasePositionAndOrientation(self.ball)
            ball_coords.append((x, y, z))
            if not sim_started and z < contact_height:
                sim_started = True
            elif not demo and sim_started and z > contact_height and step_count > min_steps:
                break
            elif np.linalg.norm([x, y]) > NET_RADIUS + BALL_RADIUS and not demo:
                break
            elif step_count > self.max_steps:
                if not sim_started:
                    raise Exception(f"Ball never hit net with input x={rim_contact_dist}, {vx}, vy={vy}")
                if not demo:
                    print("\nWarning: simulation timed out")
                break
            elif z < -3*BALL_RADIUS and not demo:
                raise Exception(f"Ball fell through net with input x={rim_contact_dist}, vx={vx}, vy={vy}")
        if save:
            frames[0].save(save, format='GIF', append_images=frames[1:], save_all=True,
duration=len(frames)/self.step_rate, loop=0)
        return np.array(ball_coords) # (step_count, 3)

    @cache
    def get_output_state(self, *input_state, **kwargs) -> tuple[float, float, float]:
        """Returns the output state (rim_dist, vx_out, vy_out) of the ball given the ball coordinates"""
        ball_coords = self.run(*input_state, **kwargs)
        rim_dist = NET_RADIUS - ball_coords[-1, 0]
        v_vec = p.getBaseVelocity(self.ball)[0]
        vx_out, vy_out = v_vec[0], -v_vec[2]
        return rim_dist, vx_out, vy_out
```

Figure C.2: Shot Optimization

```
import numpy as np
from scipy.optimize import differential_evolution, LinearConstraint, NonlinearConstraint
```

```python
from tqdm import tqdm
import sys

from bullet import GRAVITY, BALL_RADIUS, NET_RADIUS, SpikeBallSimulator


def compute_angle(vx_out, vy_out):
    rad = np.arctan2(-vy_out, -vx_out)
    if rad < 0:
        rad += 2*np.pi
    return np.rad2deg(rad)

def get_air_time(vy_out, g=GRAVITY):
    return -vy_out/g

def rebound_dist_from_rim(rim_dist, vx, vy, g=GRAVITY, raise_on_invalid=True):
    # Measured from closest point on rim, positive is away
    if vy >= 0:
        if raise_on_invalid:
            raise ValueError("vy must be negative (upward))")
        else:
            return -np.inf
    if vx >= 0:
        if raise_on_invalid:
            raise ValueError("vx must be negative (rightward)")
        else:
            return -np.inf
    x = - 2*NET_RADIUS + rim_dist
    t = get_air_time(vy, g)
    return x - vx*t - 2*NET_RADIUS


def optimize_pocket_shot(sim: SpikeBallSimulator, max_v=22.81563379224826, popsize=15, maxiter=2,
opt_func='max_xdist'):
    # Find the optimal pocket shot, i.e. the shot rebounds at the shallowest angle possible while still
clearing the net
    pbar = tqdm(total=(maxiter + 1) * popsize * 214)

    def max_xdist(rim_dist, vx_out, vy_out):
        return vx_out # Want it to bounce as fast as possible in the negative x direction

    def min_angle(rim_dist, vx_out, vy_out):
        return compute_angle(vx_out, vy_out)

    def min_air_time(rim_dist, vx_out, vy_out):
        air_time = get_air_time(vy_out)
        if air_time < 0: # TODO: Necessary?
            return np.inf
        return air_time

    func_map = {
        'max_xdist': max_xdist,
        'min_angle': min_angle,
        'min_air_time': min_air_time,
    }

    def objective(x):
        rim_dist, vx, vy = x
        xdist_out, vx_out, vy_out = sim.get_output_state(rim_dist, vx, vy)
        obj = func_map[opt_func](xdist_out, vx_out, vy_out)
        pbar.update()
        pbar.set_description(f"x={rim_dist:.5f}, vx={vx:.5f}, vy={vy:.5f} -> vx_out={vx_out:.3f},
vy_out={vy_out:.3f} (obj={obj:.3f})")
```

```
        return obj

    # rebound_constr = NonlinearConstraint(lambda x: sim.get_output_state(*x)[1], -np.inf, 0)
    # min_outbound_constr = NonlinearConstraint(lambda x: np.linalg.norm(sim.get_output_state(*x)[1:]), 0,
np.inf)
    rim_dist_bounds = [2*BALL_RADIUS, 2*NET_RADIUS - 2*BALL_RADIUS]
    vx_bounds = [0, max_v]
    vy_bounds = [0, max_v]
    vin_constr = LinearConstraint([0, 1/np.sqrt(2), 1/np.sqrt(2)], [0], [max_v], keep_feasible=True)
    constraints = [vin_constr]
    if opt_func == 'min_air_time':
        clear_rim_constr = NonlinearConstraint(lambda x: rebound_dist_from_rim(*sim.get_output_state(*x),
raise_on_invalid=False), 0, np.inf)
        constraints.append(clear_rim_constr)
    res = differential_evolution(
        objective,
        bounds=[rim_dist_bounds, vx_bounds, vy_bounds],
        constraints=constraints,
        maxiter=maxiter,
        popsize=popsize,
    )
    pbar.close()
    print(res)
    return res.x
```