UNIVERSITY OF WATERLOO | Department of Mechanical and Mechatronics Engineering

# ME 212 Billards Project Report

Alex Roman
Austin W. Milne

July 17, 2021

**Abstract**

Project report for ME 212 in Spring 2021. Using MatLab to simulation collisions of pool balls accounting for friction and restitution. Collision videos available Here.

# Contents



# List of Tables



# List of Figures



# Listings of Code

# 1 Part 1 - Collision of 2 Balls

## 1.1 Given Information

The given problem is to solve for the final velocities of 2 balls with arbitrary mass and initial velocity after colliding at a certain offset distance. Figure 1 shows the setup of the 2 balls for the collision.

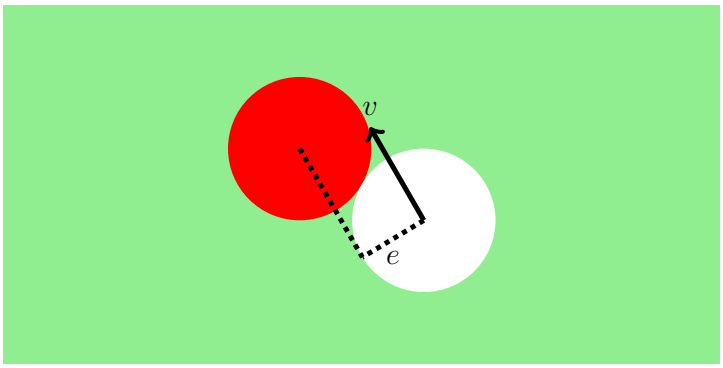


Figure 1: Part 1 Setup

## 1.2 Physics of The Collision

Since it can be assumed that there are no external forces during the collision, the problem becomes a straightforward conservation of momentum problem. The velocites can be split for each ball into the direction normal and tangent to the collision. Each part can be solved seperately. First, create a system of equations for the normal velocites before and after the impact. Equations 1 and 2 show the equations for conservation of momentum and for impact restitution, respectively.

$$v_a m_a + v_b m_b = v'_a m_a + v'_b m_b \tag{1}$$

$$e_{\text{rest.}} = \frac{v'_a - v'_b}{v_b - v_a} \tag{2}$$

Solving this system of equations gives the normal velocities of the 2 balls after the collision, $v'_a$ and $v'_b$. For tangent velocities, due to friction, energy loses need to be accounted for in the collision. Since the change in momentum in the normal direction is the intergal of the force over the impact time, the change in tangent velocity can be derived from the change in normal momentum. Equation 3 shows the derivation for a balls tangent velocity after impact.

$$
\begin{aligned}
mv + \int_{t_1}^{t_2} F dt &= mv' \\
mv_t + \int_{t_1}^{t_2} F_f dt &= mv'_t \\
mv_t + \mu\left(mv'_n - mv_n\right) &= mv'_t \quad \Leftarrow \text{Using } F_f = \mu_k F_n \\
v_t + \mu\left(v'_n - v_n\right) &= v'_t
\end{aligned}
\tag{3}
$$

## 1.3 Application of Physics

In order to apply the equations above, the normal and tangent velocities of the balls need to be determined. Basic trigonometery can be used to determine the relative velocities. In this case, it is difficult to determine the normal and tangent values from the velocity vector and the $e$ value alone. To make it easier, Figure 2 shows construction lines added for easier computation.

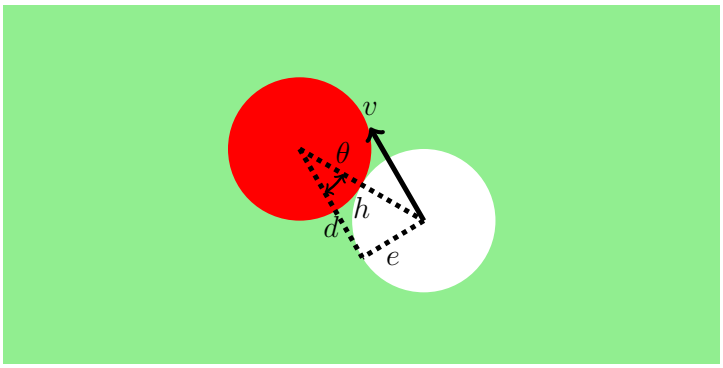


Figure 2: Part 1 Construction Lines

Using the construction lines above, the below equations can be derived for the tangent and normal velocities of the moving ball. Equation 4 shows the derivation of the impact angle and relative angles for vector rotation.

$$
\begin{aligned}
h &= 2r \\
\theta &= \sin^{-1}\left(\frac{e}{h}\right) = \sin^{-1}\left(\frac{e}{2r}\right) \\
\theta_{\text{velocity}} &= \tan^{-1}\left(\frac{v_x}{v_y}\right) \\
\theta_{\text{impact}} = \theta_{\text{velocity}} - \theta &= \tan^{-1}\left(\frac{v_x}{v_y}\right) - \sin^{-1}\left(\frac{e}{2r}\right)
\end{aligned}
\tag{4}
$$

Equation 5 shows the derivation for the normal and tangent velocities relative to the collision by way of matrix-based vector rotation.

$$
\begin{aligned}
\phi &= \theta_{\text{impact}} - \theta_{\text{velocity}} \\
R &= \begin{bmatrix} \cos(\phi) & -\sin\phi \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \\
v_{a_{\text{norm,tang}}} &= R \times v_a
\end{aligned}
\tag{5}
$$

Having the normal and tangent velocities for the collision, the below set Equation set 6 can be applied to get the final normal and tangent velocities for the both ball A (red) and ball b (white).

$$
\begin{aligned}
v_{a_n} m_a + v_{b_n} m_b &= v'_{a_n} m_a + v'_{b_n} m_b \\
e_{\text{rest.}} &= \frac{v'_{a_n} - v'_{b_n}}{v_{b_n} - v_{a_b}} \\
v'_{a_t} &= v_{a_t} + \mu \left( v'_{a_n} - v_{a_n} \right) \\
v'_{b_t} &= v_{b_t} + \mu \left( v'_{b_n} - v_{b_n} \right)
\end{aligned}
\tag{6}
$$

Finally, to get the final velocities of the 2 balls in the $x$-$y$ reference plane, another vector rotation needs to be done to revert to the $xy$ coordinate system. Equation 7 shows this vector rotation.

$$
\begin{aligned}
\phi &= \theta_{\text{velocity}} - \theta_{\text{impact}} \\
R &= \begin{bmatrix} \cos(\phi) & -\sin\phi \\ \sin(\phi) & \cos(\phi) \end{bmatrix} \\
v_a &= R \times v_{a_{\text{norm,tang}}} \\
v_b &= R \times v_{b_{\text{norm,tang}}}
\end{aligned}
\tag{7}
$$

# 2 Part 2 - Collisions of a Table of Balls

## 2.1 Given Information

The problem is given to simulate the motion of a set of 11 balls in a square enclosure, Similair to that of a billiards table. The initial position and velocity for each of the balls is provided, along with the mass and radius. Simplifications are made to assume that the system is effectively 2D without spin or rotation. Restitution coefficients for ball-on-ball and ball-on-wall are provided, as well as the frictional coefficient for ball-on-surface, ball-on-wall, and ball-on-ball. Figure 3 is provided as an example for the visualization of the balls and table.

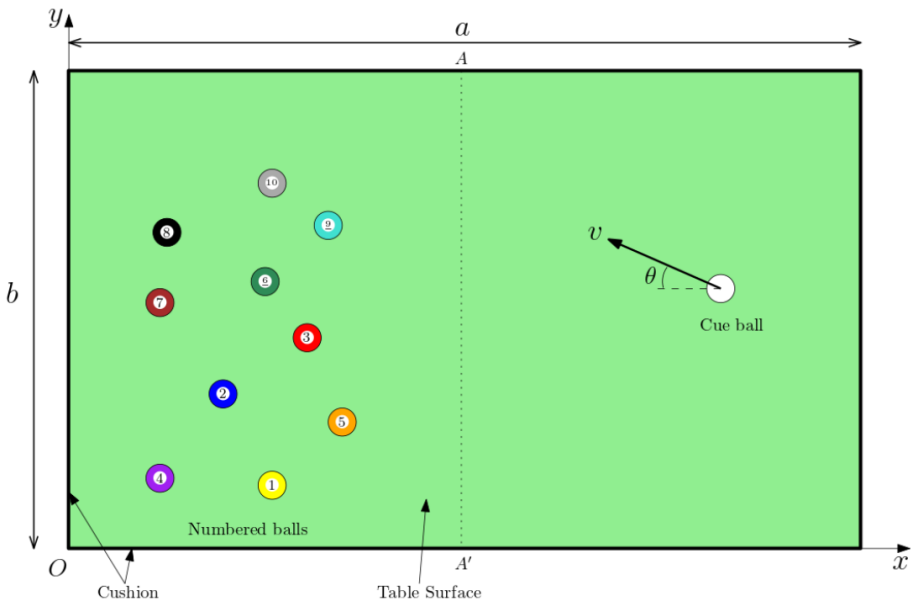


Figure 3: Part 2 Table Layout

## 2.2 Indentifying Collisions and Physics

In order to simulate the motion of the balls, the different motion states of the balls need to be identified. For balls on a table with bumpers, as outlined in section 2.1, there are 3 primary motions: movement for a balls velocity, impact of ball into a wall, and impact of a ball into another ball. Each of these types of motion can be modelled seperately. The strategy used to handle time scales and determine collisions of balls was to calculate the positions of all the balls after very small time changes. After each time interval, each ball is checked for positional overlap with walls and all the other balls on the table. If a ball is overlapping with a wall or another ball, the change in velocity is calculated for the collision. A couple assumptions are made in order to model the system this way. The first assumption is that collisions are instantaneous, and therefore do not have a time range of occurance. This leads to the post-collision velocities being applied instantaneously after balls are determined to be impacting a ball or wall. The second assumption is that the accuracy of using very small time chunks is sufficient enough for the purposes of this simulation. This assumption is substantiated by the calculations in Equation 8 which shows that, in the worst case of a glancing shot with maximum time error, the anglular inaccuracy of a collision calculation is equal to roughly 1 millimeter radial displacment over a 2 meter distance.

$$\theta_{\text{error}}^{\max} = \tan^{-1}\left(\frac{4\left[\frac{m}{sec}\right] * 39.3701\left[\frac{in}{sec}/\frac{m}{sec}\right] * \frac{1}{100000}\,[sec]}{2.25\,[in]} = 0.040102^{\circ}\right) \tag{8}$$

$$\sin(\theta_{\text{error}}^{\max}) * 2\,[m] = 0.0013998\,[m] = 1.3998\,[mm]$$

The equation for 2 balls colliding is already derived and shown in Equation 5, 6, and 7 above. Equation 9 below is the derivation for the velocities after a ball has impacted a wall.

$$\begin{aligned} v_n' &= -e_{\text{rest.}} v_n \\ v_t' &= v_t - \mu \left( v_n' - v_n \right) \end{aligned} \tag{9}$$

The majority of the math is carried over from Equation 6 with the velocity for the second ball being set to 0. Additionally, the movement for a ball with friction over the surface of the table can be modeled by Equation 10, where the position, $s$, changes with the average velocity over the specified time chunk.

$$\begin{aligned} v' &= v + (\Delta t \ \mu \ g) \\ v_{\text{avg}} &= \frac{v + v'}{2} \\ s' &= s + \Delta t \ v_{\text{avg}} \end{aligned} \tag{10}$$

## 2.3 Creating the Simulation Code

While converting the equations listed above for the motion of the balls on the table, many small changes were made in order to make the math more universal and less sensitive to angles and directions. For collision angle calculations in Equation 4, $tan^{-1}$ was replaced with `atan2()` in Listing 3, lines 58 and 121 to make the angle quadrant-aware. For calculation of frictional losses, the direction of relative velocity needed to be accounted for to ensure that the friction was applied correctly. This led to using the `sign()` function for checking the direction of relative motion in Listing 3, lines 131 and 132. Also, in order to ensure the friction does not change the direction of a ball, but instead only slows it to a stop, the sign of the balls' tangent velocities had to be checked and corrected, as seen in Listing 3, on lines: 61, 79, 89, 134, and 137.

The code for the simulation was written in an object-oriented style, with the `PoolBall` class, defined in Listing 3, containing all the physics and collision math for the motion of each ball. The main script, Listing 2, creates the set of balls and runs the physics functions repeatedly, storing the graphs as video frames and exporting the relevent information for the simulation.

The fourth shot of the simulation, where the cue ball is to be hit at 1.5 $m/s$ toward the nearest ball, is calculated at the start of the script. The subject ball and angle of cue shot are output as part of the log file in Listing 1, before the simulations are run.

The final positions of all the balls are output to the log file, Listing 1, with accuracy down to 1/100th of an inch.

# 3 Appendix

The video files of the collisions, both combined video and individual collisions are available on the University of Waterloo OneDrive, Here.

```
Full Link: https://uofwaterloo-my.sharepoint.com/:f:/g/personal/awbmilne_uwaterloo_ca/
    EiWN90NLaUVGpE0EcLgVf8wBgqZj-v4eaprjO5gxFSzF4A?e=SygQLX
```

Table 1: Starting Positions

| **Ball #** | **X pos** | **Y pos** |
|---|---|---|
| 1 | 40.08 | 21.23 |
| 2 | 17.52 | 37.29 |
| 3 | 12.53 | 2.75 |
| 4 | 10.47 | 25.10 |
| 5 | 27.32 | 4.87 |
| 6 | 16.25 | 13.80 |
| 7 | 7.60 | 21.49 |
| 8 | 10.81 | 9.21 |
| 9 | 26.19 | 21.84 |
| 10 | 19.78 | 26.81 |
| Cue | 66.00 | 22.00 |

Listing 1: Script Output — output.log

```
Closest ball to Cue ball: Ball  1
Attitude to closest ball: -178.298 deg

Simulation 1: Cue velocity = [-4.000, 0.000] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [27.37, 29.12] (in)
    Ball  1: [17.33,  3.93] (in)
    Ball  2: [17.97, 36.62] (in)
    Ball  3: [ 7.47,  6.32] (in)
    Ball  4: [10.47, 25.10] (in)
    Ball  5: [27.32,  4.87] (in)
    Ball  6: [24.66, 21.99] (in)
    Ball  7: [ 7.60, 21.49] (in)
    Ball  8: [14.76,  9.38] (in)
    Ball  9: [20.81, 25.31] (in)
    Ball 10: [16.85, 41.25] (in)

Simulation 2: Cue velocity = [-2.415, 0.647] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [ 2.31, 24.66] (in)
    Ball  1: [40.08, 21.23] (in)
    Ball  2: [ 4.12, 31.51] (in)
    Ball  3: [12.53,  2.75] (in)
    Ball  4: [10.63, 24.89] (in)
    Ball  5: [27.32,  4.87] (in)
    Ball  6: [16.25, 13.80] (in)
    Ball  7: [ 7.60, 21.49] (in)
    Ball  8: [10.81,  9.21] (in)
    Ball  9: [26.19, 21.84] (in)
    Ball 10: [19.78, 26.81] (in)

Simulation 3: Cue velocity = [-1.299, 0.750] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [28.72, 40.21] (in)
    Ball  1: [40.08, 21.23] (in)
    Ball  2: [17.52, 37.29] (in)
    Ball  3: [12.53,  2.75] (in)
    Ball  4: [10.47, 25.10] (in)
    Ball  5: [27.32,  4.87] (in)
    Ball  6: [16.25, 13.80] (in)
    Ball  7: [ 7.60, 21.49] (in)
    Ball  8: [10.81,  9.21] (in)
    Ball  9: [26.19, 21.84] (in)
    Ball 10: [19.78, 26.81] (in)

Simulation 4: Cue velocity = [-1.499,-0.045] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [43.16, 21.32] (in)
    Ball  1: [30.64, 21.41] (in)
    Ball  2: [17.52, 37.29] (in)
    Ball  3: [12.53,  2.75] (in)
    Ball  4: [10.47, 25.10] (in)
    Ball  5: [27.32,  4.87] (in)
    Ball  6: [16.25, 13.80] (in)
    Ball  7: [ 7.60, 21.49] (in)
    Ball  8: [10.81,  9.21] (in)
    Ball  9: [17.19, 21.28] (in)
    Ball 10: [19.49, 27.83] (in)
```

Table 2: Simulation Snapshots

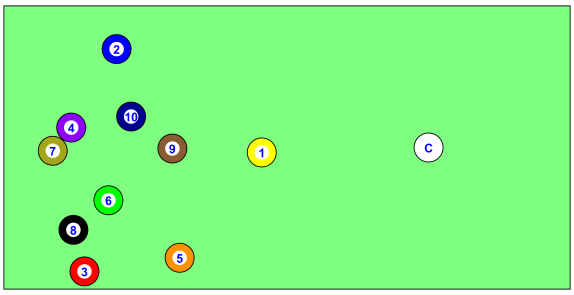

Figure 4: Simulation 1 Start

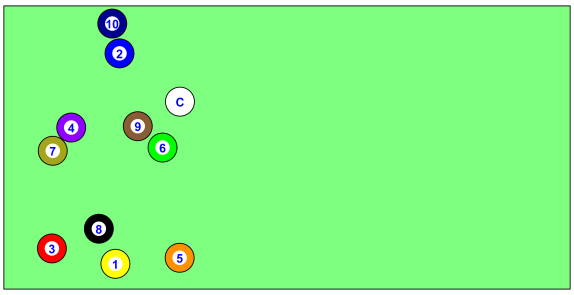

Figure 5: Simulation 1 End

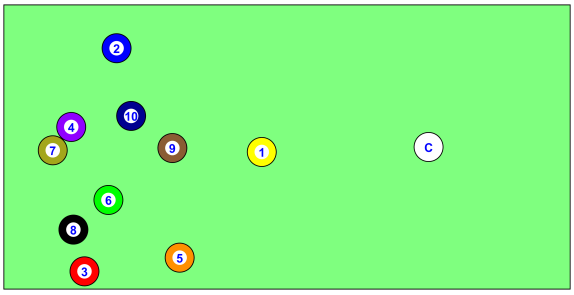

Figure 6: Simulation 2 Start

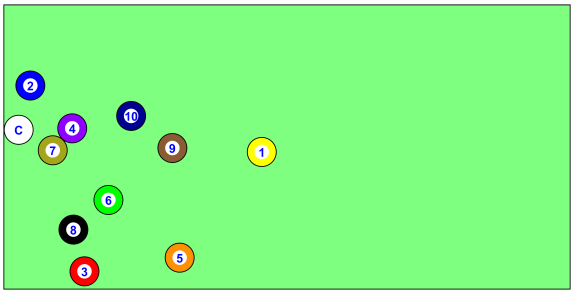

Figure 7: Simulation 2 End

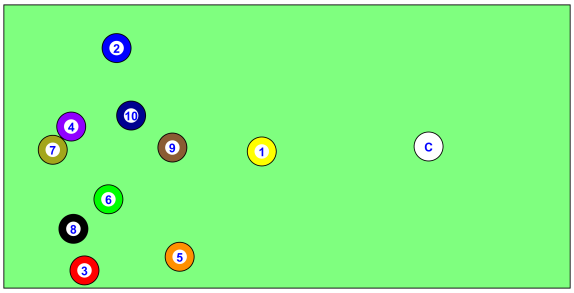

Figure 8: Simulation 3 Start

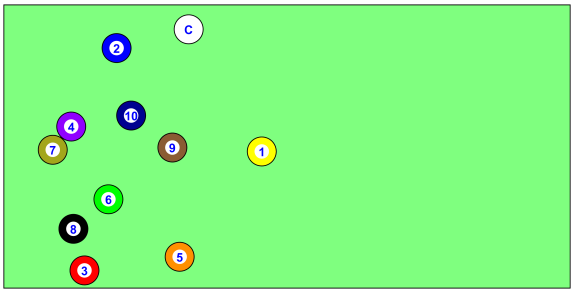

Figure 9: Simulation 3 End

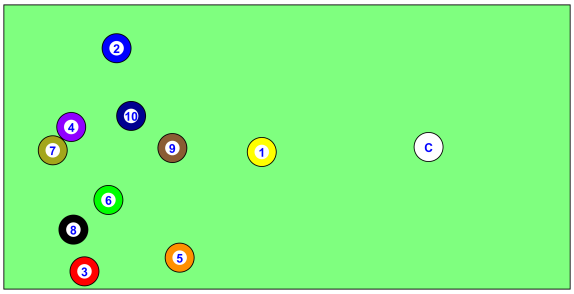

Figure 10: Simulation 4 Start

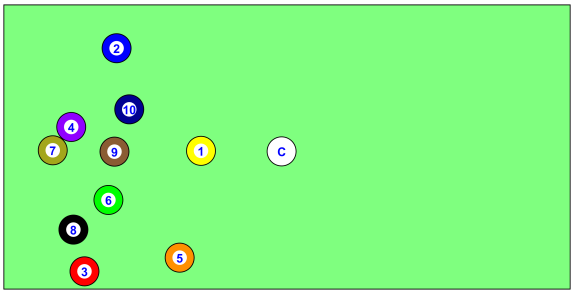

Figure 11: Simulation 4 End

Listing 2: MatLab Script — BilliardsCode.m

```
% Clear the commands and variables
close all;
clear all;
clc;
output_dir = "out/matlab/";

% Start logging the output to diary/log file
dfile = output_dir + 'output.log';
if exist(dfile, 'file') ; delete(dfile); end
diary(dfile)
diary on;

% Set the time interval for calculations
% (Determines precision)
time_slice = 0.000001; % Recompute position and velocity every 1/1,000,000 of a second
slices_per_sec = 1 / time_slice; % Theoretical frames/sec
frame_divider = round(slices_per_sec / 60); % Aim for roughly 60 fps for output video
frame_rate = slices_per_sec/frame_divider;

% Create frame and video objects for storing video frames
combinedVideo = VideoWriter(output_dir + 'videos/combinedCollisions.avi');
combinedVideo.FrameRate = frame_rate;
combinedVideo.Quality = 95;
open(combinedVideo);

% Setup and draw pool table background
grid on;
hold on;
axis equal;
axis off;
patch([ 0, 0, PoolBall.table_length, PoolBall.table_length], ...
      [ 0,PoolBall.table_width,PoolBall.table_width, 0], 'g','FaceAlpha',0.5);
axis([0 PoolBall.table_length 0 PoolBall.table_width]);

% Create array of Pool Balls with given properties and locations
%                  | Mass(g) | Rad(in) | Position(in) |    Colour(RGB)    | TT | is_cue?
balls = [PoolBall(       200,     2.25, [66.00,22.00], [255,255,255]/255, 'C',   true), ...
         PoolBall(       160,     2.25, [40.08,21.23], [255,255,  0]/255, '1',  false), ...
         PoolBall(       160,     2.25, [17.52,37.29], [  0,  0,255]/255, '2',  false), ...
         PoolBall(       160,     2.25, [12.53, 2.75], [255,  0,  0]/255, '3',  false), ...
         PoolBall(       160,     2.25, [10.47,25.10], [144,  0,255]/255, '4',  false), ...
         PoolBall(       160,     2.25, [27.32, 4.87], [255,144,  0]/255, '5',  false), ...
         PoolBall(       160,     2.25, [16.25,13.80], [  0,255,  0]/255, '6',  false), ...
         PoolBall(       160,     2.25, [ 7.60,21.49], [163,166, 27]/255, '7',  false), ...
         PoolBall(       160,     2.25, [10.81, 9.21], [  0,  0,  0]/255, '8',  false), ...
         PoolBall(       160,     2.25, [26.19,21.84], [138, 92, 51]/255, '9',  false), ...
         PoolBall(       160,     2.25, [19.78,26.81], [  0,  0,144]/255, '10', false)];
ball_pairs = num2cell(nchoosek(balls, 2), 2); % Create Cell array of ball pair combinations
cue_ball = balls(1); % Synonym for Cue Ball

% Determine Closest ball to cue and angle for shot
closest_ball = balls(1).find_nearest(balls(2:end));
vector_between = closest_ball.pos - balls(1).pos;
theta_closest = atan2(vector_between(2), vector_between(1));
fprintf("Closest ball to Cue ball: Ball %2s\n", closest_ball.label) % Log closest ball
fprintf("Attitude to closest ball: %7.3f deg\n\n", rad2deg(theta_closest)); % Log attitude
    to closest ball

% Create the array of Cue Ball Velocities to test
cue_velocities = ...
[
    [ -cos(deg2rad( 0))*  4,   sin(deg2rad( 0))*  4];... % 4 m/s directly left
    [ -cos(deg2rad(15))*2.5,   sin(deg2rad(15))*2.5];... % 2.5 m/s @ 15 deg upward toward
        the left
    [ -cos(deg2rad(30))*1.5,   sin(deg2rad(30))*1.5];... % 1.5 m/s @ 30 deg upward toward
        the left
```

```
    [cos(theta_closest)*1.5, sin(theta_closest)*1.5];... % 1.5 m/s @ angle to closest ball
];


%% Physics Simulation
% Interate over each cue velocity to test
for i=1: length(cue_velocities)
    % Reset the pool balls
    arrayfun(@(ball) ball.reset, balls);

    %% Setup Frame and Video
    % Create frame and video objects for storing video frames
    individualVideo = VideoWriter(output_dir + 'videos/' + sprintf("collision_%i", i) + '.avi');
    individualVideo.FrameRate = frame_rate;
    individualVideo.Quality = 95;
    open(individualVideo);
    % Save the start layout as an image
    exportgraphics(gca,output_dir + sprintf("images/Simulation_%i_start.png", i),'Resolution',600)

    % Set the cue velocity for the test
    cue_ball.set_vel(cue_velocities(i,:));

    % Simulate the shot
    frame_counter=0;
    while(true)
        % Break loop if no balls are moving
        if (~any(arrayfun(@(ball) any(ball.vel), balls))) break, end;

        % Increment frame counter
        frame_counter = frame_counter + 1;

        % Ball physics
        arrayfun(@(ball) ball.move(time_slice), balls); % Update the position with velocity and apply rolling friction
        arrayfun(@(ball) ball.compute_wall_collisions(), balls); % Compute the velocity changes from wall collisions
        arrayfun(@(pair) pair{1}(1).compute_ball_collision(pair{1}(2)), ball_pairs); % Check collisions in each pair of balls

        % Render and write frames to the videos, subject to the frame divider
        if (frame_counter == 0 || mod(frame_counter, frame_divider) == 0)
            arrayfun(@(ball) ball.draw, balls); % Draw all the balls
            frame = getframe(gcf); % Save the rendered graph as a video frame
            writeVideo(combinedVideo, frame); % Write the frame to the video file
            writeVideo(individualVideo, frame) % Write the frame to the video file
        end
    end

    % Add the frame to the video repeatedly for the next 1 second
    % (Creates a 1 second pause at the end of the collision)
    arrayfun(@(ball) ball.draw, balls); % Draw all the balls
    frame = getframe(gcf); % Save the rendered graph as a video frame
    for j=1: frame_rate
        writeVideo(combinedVideo, frame) % Write the frame to the video file
        writeVideo(individualVideo, frame) % Write the frame to the video file
    end
    close(individualVideo); % Finish the specific collision video
    exportgraphics(gca,output_dir + sprintf("images/Simulation_%i_end.png", i),'Resolution',600) % Save the final layout of the balls as an image

    % Output the simulation results
    fprintf("Simulation %i: Cue velocity = [%6.3f,%6.3f] (m/s)\n", i, cue_velocities(i,1), cue_velocities(i,2));
    fprintf("    Final Ball positions [X,Y]:\n")
    for j=1: length(balls) % List all ball positions
        fprintf("    Ball %2s: [%5.2f, %5.2f] (in)\n", ...
```

```
                balls(j).label,                          ...
                convlength(balls(j).pos(1),'m','in'),    ...
                convlength(balls(j).pos(2),'m','in')); ...
    end
    fprintf("\n");
end

%% Clean up Script
close(combinedVideo); % Wrap up the video file
diary off; % Close the diary/log file
```

Listing 3: PoolBall Class Definition — PoolBall.m

```
%% PoolBall class
% Carries the properties and states of a ball on the table
classdef PoolBall < handle
    %% Shared Properties
    properties (Constant)
        table_length = convlength(88,'in','m');  % Length of pool table (x-direction)
        table_width =  convlength(44,'in','m'); % Width of pool table (y-direction)

        mu_bb = 0.05; % Coefficient of friction between 2 balls
        mu_bs = 0.10; % Coefficient of friction between ball and surface
        mu_bc = 0.20; % Coefficient of friction between ball and wall cushion/bumper

        e_cn = 0.95; % Coefficient of resistution between cue ball and numbered ball
        e_nn = 0.90; % Coefficient of restitution between 2 numbered balls
        e_bc = 0.70; % Coefficient of restitution between ball and wall cushion/bumper

        gravity = 9.80665; % Earths gravity in m/s
    end
    %% Particular Properties
    properties
        mass; % Mass of the ball (kg)
        rad;  % Radius of the ball (m)
        home_pos % Starting position of the ball (m)
        pos; % Current position of the ball (m)
        vel; % Current velocity of the ball (m/s)
        color; % Color of the ball (RGB matrix)
        label; % Label of the ball (1-2 Characters)
        is_cue; % Boolean specifying if this ball is a/the cue ball
        colliding_balls= PoolBall.empty; % List of balls currently in collision with this
        ball_img; % Image object for the ball
        spot_img; % White spot object for the ball
        label_img; % Label img for the ball
    end
    %% Methods
    methods
        %% PoolBall Constructor
        % Initializes the ball given the specified parameters
        function PB = PoolBall(mass,rad,pos,color,label,is_cue)
            PB.mass = mass / 1000; % Convert grams to kilograms
            PB.rad = convlength(rad, 'in', 'm'); % Convert inches to meters
            PB.home_pos = convlength(pos, 'in', 'm'); % Convert inches to meters
            PB.pos = PB.home_pos;
            PB.vel = [0,0]; % Keep m/s as m/s
            PB.color = color;
            PB.label = label;
            PB.is_cue = is_cue;
            a=[0:0.1:2*pi];
            Xcircle=cos(a);
            Ycircle=sin(a);
            PB.ball_img = patch (PB.pos(1)+PB.rad*Xcircle, PB.pos(2)+PB.rad*Ycircle, PB.
                color, 'FaceAlpha',1);
            PB.spot_img = patch (PB.pos(1)+PB.rad*Xcircle/2, PB.pos(2)+PB.rad*Ycircle/2, 'w'
                , 'FaceAlpha', 1, 'LineStyle' ,'none');
            PB.label_img = text(PB.pos(1), PB.pos(2), PB.label, 'FontSize', 7, 'Color', 'b',
                 'HorizontalAlignment', 'center', 'VerticalAlignment', 'middle', 'FontWeight
                ', 'bold');
        end
        %% PoolBall Move
        % Update the position and velocity of the pool ball based in the
        % velocity and surface friction
        function move(obj, time_slice)
            vel_angle = atan2(obj.vel(2), obj.vel(1)); %
            old_vel = norm(obj.vel); % Get the normal of the old velocity
            new_vel = old_vel - (time_slice * PoolBall.mu_bs * PoolBall.gravity); % Loss of
                velocity from friction
            if (new_vel < 0) % Check if the new velocity was reduced past 0
```

```
62                  new_vel = 0;
63              end
64              effective_vel = [cos(vel_angle) * (old_vel + new_vel)/2, sin(vel_angle) * (
                    old_vel + new_vel)/2]; % Use velocity of average between old and new for
                    time section
65              obj.vel = [cos(vel_angle) * new_vel, sin(vel_angle) * new_vel]; % Update objects
                     velocity
66              obj.pos = obj.pos + effective_vel * time_slice; % Update the objects position
67          end
68          %% PoolBall Wall Collisions
69          % Compute velocity changes due to wall/cushion/bumper collisions
70          function compute_wall_collisions(obj)
71              max_x = PoolBall.table_length;
72              max_y = PoolBall.table_width;
73              % Check if we collided with the x=max or x=0 "bumpers"
74              if(obj.pos(1) > (max_x - obj.rad) || obj.pos(1) < obj.rad)
75                  old_vel = obj.vel; % Hold the original velocity for comparison
76                  obj.vel(1) = -(obj.vel(1)*PoolBall.e_bc); % Invert and reduce the
                        perpendicular velocity by restitution factor
77                  obj.pos(1) = min(max(obj.pos(1),obj.rad),max_x-obj.rad); % Put the ball back
                         within the limits (Eliminate glitches)
78                  obj.vel(2) = obj.vel(2) - sign(obj.vel(2))*(PoolBall.mu_bc * abs(obj.vel(1)-
                        old_vel(1))); % Solve for frictional loss to parallel velocity
79                  if (sign(obj.vel(2)) ~= sign(old_vel(2))) % Ensure we didnt reverse
                        direction due to friction
80                      obj.vel(2) = 0;
81                  end
82              end
83              % Check if we collided with the y=max or y=0 "bumpers"
84              if(obj.pos(2) > (max_y - obj.rad) || obj.pos(2) < obj.rad)
85                  old_vel = obj.vel; % Hold the original velocity for comparison
86                  obj.vel(2) = -(obj.vel(2)*PoolBall.e_bc); % Invert and reduce the
                        perpendicular velocity by restitution factor
87                  obj.pos(2) = min(max(obj.pos(2),obj.rad),max_y-obj.rad); % Put the ball back
                         within the limits (Eliminate glitches)
88                  obj.vel(1) = obj.vel(1) - sign(obj.vel(1))*(PoolBall.mu_bc * abs(obj.vel(2)-
                        old_vel(2))); % Solve for frictional loss to parallel velocity
89                  if (sign(obj.vel(1)) ~= sign(old_vel(1))) % Ensure we didnt reverse
                        direction due to friction
90                      obj.vel(1) = 0;
91                  end
92              end
93          end
94          %% PoolBall Ball Collisions
95          % Compute velocity changes to ball<->ball collisions
96          function compute_ball_collision(ball_a, ball_b)
97              % Check if the balls collided
98              if (norm(ball_a.pos - ball_b.pos) < (ball_a.rad + ball_b.rad))
99                  % Check if we are still colliding with this ball
100                 %   Approximation method needed for time-slice based approach.
101                 %   This avoids calculating multiple collisions while balls are
102                 %   clipped through eachother. Smaller time-slice sizes reduces
103                 %   approximation error here.
104                 if (~isempty(ball_a.colliding_balls))
105                     for i=1: length(ball_a.colliding_balls)
106                         if (isequal(ball_b, ball_a.colliding_balls(i)))
107                             return;
108                         end
109                     end
110                 end
111                 % Else, add the ball to the reference array
112                 ball_a.colliding_balls(end+1) = ball_b;
113                 % Check if either ball is the cue ball, select the correct e
114                 if (ball_a.is_cue || ball_b.is_cue)
115                     e = PoolBall.e_bc; % Coefficient for cue and numbered ball
116                 else
117                     e = PoolBall.e_nn; % Coefficient for 2 numbered balls
118                 end
```

```
119                 % Get the angles and velocities, relative to the impact
120                 difference = ball_b.pos - ball_a.pos;
121                 impact_angle = atan2(difference(2), difference(1)); %
122                 ball_a_vr = ball_a.vel * rotmat(impact_angle); % Velocity rotated to
                        collision angle [norm, tang]
123                 ball_b_vr = ball_b.vel * rotmat(impact_angle); % Velocity rotated to
                        collision angle [norm, tang]
124                 % Create system of equations for the normal velocities, and solve
125                 syms new_ball_a_norm_v new_ball_b_norm_v
126                 norm_momentum = ball_a_vr(1) * ball_a.mass + ball_b_vr(1) * ball_b.mass ==
                        new_ball_a_norm_v * ball_a.mass + new_ball_b_norm_v * ball_b.mass;
127                 norm_restitution = e == (new_ball_a_norm_v - new_ball_b_norm_v)/(ball_b_vr
                        (1) - ball_a_vr(1));
128                 [A,B] = equationsToMatrix([norm_momentum, norm_restitution], [
                        new_ball_a_norm_v, new_ball_b_norm_v]);
129                 X = linsolve(A,B); % Solve system
130                 % Determine frictional losses on tangent velocities
131                 new_ball_a_vr = [double(X(1)), ball_a_vr(2) - sign(ball_a_vr(2)-ball_b_vr(2)
                        )*(PoolBall.mu_bb * (ball_a_vr(1) - double(X(1))))]; %
132                 new_ball_b_vr = [double(X(2)), ball_b_vr(2) - sign(ball_b_vr(2)-ball_a_vr(2)
                        )*(PoolBall.mu_bb * (ball_b_vr(1) - double(X(2))))]; %
133                 % Ensure frictional loss didnt change sign of velocity
134                 if (sign(new_ball_a_vr(2)) ~= sign(ball_a_vr(2)) && sign(ball_a_vr(2)) ~= 0
                        ) %
135                     new_ball_a_vr(2) = 0;
136                 end
137                 if (sign(new_ball_b_vr(2)) ~= sign(ball_b_vr(2)) && sign(ball_b_vr(2)) ~= 0)
                         %
138                     new_ball_b_vr(2) = 0;
139                 end
140                 % Apply new velocities to balls
141                 ball_a.vel = new_ball_a_vr * rotmat(-impact_angle);
142                 ball_b.vel = new_ball_b_vr * rotmat(-impact_angle);
143                 return;
144             % If this ball isnt overlapping, Check if we were previously colliding with it
145             elseif (~isempty(ball_a.colliding_balls)) % Make sure array isnt empty
146                 for i=1: length(ball_a.colliding_balls)
147                     if (isequal(ball_b, ball_a.colliding_balls(i))) % Compare object
                            references
148                         ball_a.colliding_balls(i) = []; % Delete the ball reference
149                     end
150                 end
151             end
152         end
153         %% PoolBall Draw
154         % Draw the ball on the graph at the current position
155         function draw(obj)
156             a=[0:0.1:2*pi];
157             Xcircle=cos(a);
158             Ycircle=sin(a);
159             set(obj.ball_img,'XData',obj.pos(1)+obj.rad*Xcircle, 'YData', obj.pos(2)+obj.rad
                    *Ycircle);
160             set(obj.spot_img,'XData',obj.pos(1)+obj.rad*Xcircle/2, 'YData', obj.pos(2)+obj.
                    rad*Ycircle/2);
161             set(obj.label_img,'Position', [obj.pos(1) obj.pos(2) 0]);
162         end
163         %% PoolBall Reset
164         % Reset the position and velocity of the ball
165         function reset(obj)
166             obj.pos = obj.home_pos;
167             obj.vel = [0,0];
168             obj.draw();
169         end
170         %% PoolBall Set Velocity
171         % Set the velocity of the pool ball (Clearer than value accessor)
172         function set_vel(obj, vel)
173             obj.vel = vel;
174         end
```

```
175         %% PoolBall Find Closest
176         % Determine the closest ball from a given array of balls (Returns ball reference)
177         function closest = find_nearest(obj, balls)
178             closest = PoolBall.empty; % Empty reference
179             min_dist = norm([PoolBall.table_length, PoolBall.table_width]); % Set to max
                    value
180             for i=1: length(balls)
181                 distance = norm(balls(i).pos - obj.pos);
182                 if (distance < min_dist) % Check if this ball is closer
183                     min_dist = distance; % Update the minimum distance
184                     closest = balls(i); % Update the object reference
185                 end
186             end
187         end
188     end
189 end
190
191 % Rotation Matrix function for less verbose vector rotation math
192 function R = rotmat(angle)
193     R = [cos(angle), -sin(angle)
194          sin(angle),  cos(angle)];
195 end
```