Department of Mechanical and Mechatronics Engineering

ME 212 Billards Project Report

Alex Roman
Austin W. Milne

July 8, 2021

Abstract

Project report for ME 212 in Spring 2021. Using MatLab to simulation collisions of pool balls with friction and restitution.

Contents

1 Problem 1 **3**
1.1 Show Transformation 3

List of Tables

1 Starting Positions 3
2 Simulation Snapshots 4

List of Figures

1 Simulation 1 Start 4
2 Simulation 1 End 4
3 Simulation 2 Start 4
4 Simulation 2 End 4
5 Simulation 3 Start 4
6 Simulation 3 End 4
7 Simulation 4 Start 4
8 Simulation 4 End 4

List of Code

1 Script Output — output.log 5
2 MatLab Script — BilliardsCode.m 6
3 PoolBall Class Definition — PoolBall.m 9

1 Problem 1

1.1 Show Transformation

Since we got equation 1 back, equation 2 is just equation 1 with z substituted in. I want to then reference Table 1 as an example.

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

Table 2: Simulation Snapshots

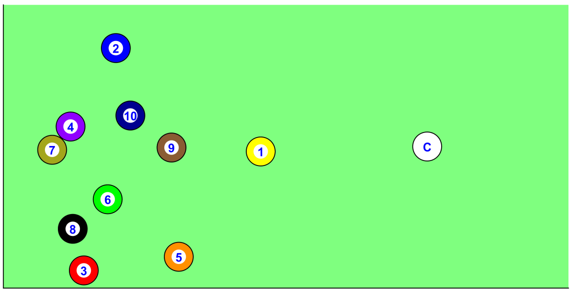

Figure 1: Simulation 1 Start

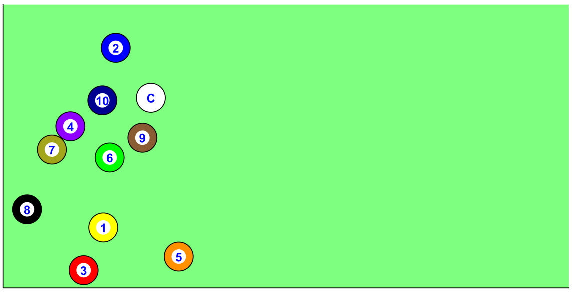

Figure 2: Simulation 1 End

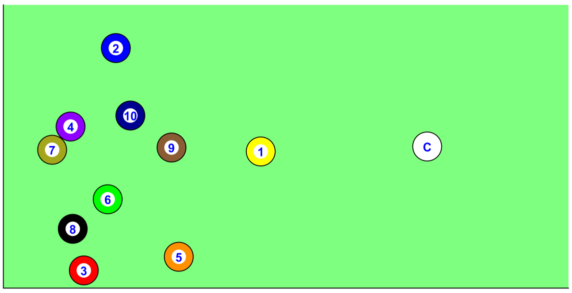

Figure 3: Simulation 2 Start

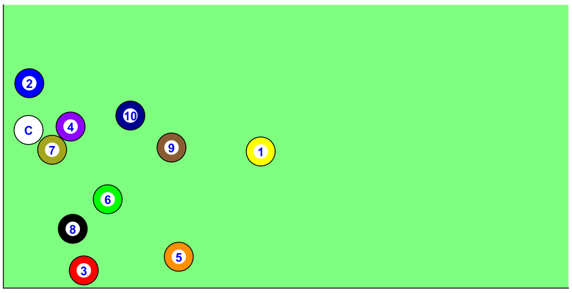

Figure 4: Simulation 2 End

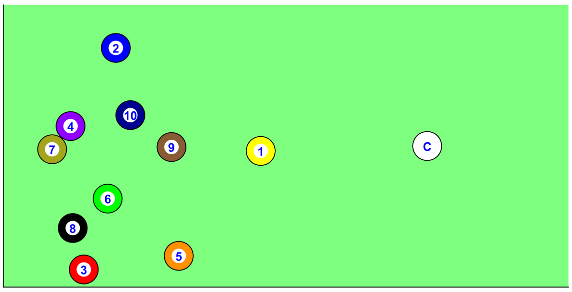

Figure 5: Simulation 3 Start

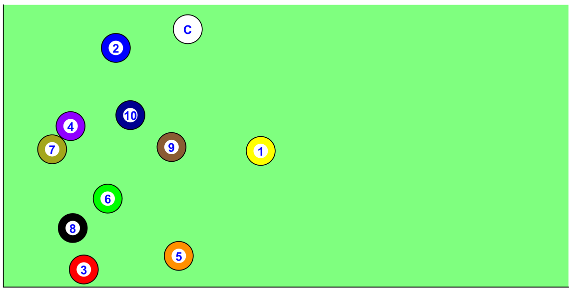

Figure 6: Simulation 3 End

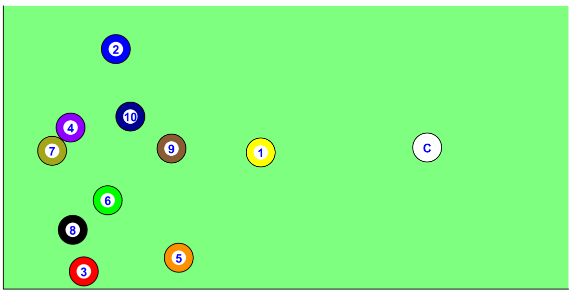

Figure 7: Simulation 4 Start

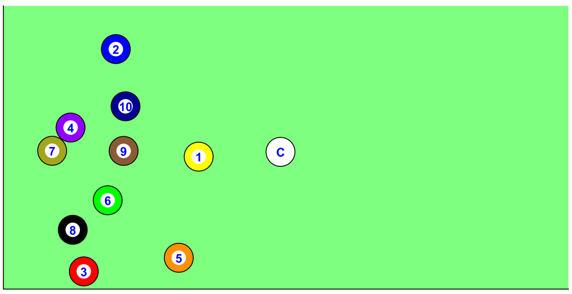

Figure 8: Simulation 4 End

Code 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: [22.9819, 29.5318] (in)
    Ball  1: [15.5860,  9.3905] (in)
    Ball  2: [17.5200, 37.2900] (in)
    Ball  3: [12.5300,  2.7500] (in)
    Ball  4: [10.4700, 25.1000] (in)
    Ball  5: [27.3200,  4.8700] (in)
    Ball  6: [16.5767, 20.2605] (in)
    Ball  7: [ 7.6000, 21.4900] (in)
    Ball  8: [ 3.7308, 12.2033] (in)
    Ball  9: [21.6689, 23.3315] (in)
    Ball 10: [15.4402, 29.1312] (in)

Simulation 2: Cue velocity = [-2.415, 0.647] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [ 3.9257, 24.5638] (in)
    Ball  1: [40.0800, 21.2300] (in)
    Ball  2: [ 4.0452, 31.8299] (in)
    Ball  3: [12.5300,  2.7500] (in)
    Ball  4: [10.4700, 25.1000] (in)
    Ball  5: [27.3200,  4.8700] (in)
    Ball  6: [16.2500, 13.8000] (in)
    Ball  7: [ 7.6000, 21.4900] (in)
    Ball  8: [10.8100,  9.2100] (in)
    Ball  9: [26.1900, 21.8400] (in)
    Ball 10: [19.7800, 26.8100] (in)

Simulation 3: Cue velocity = [-1.299, 0.750] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [28.7202, 40.2055] (in)
    Ball  1: [40.0800, 21.2300] (in)
    Ball  2: [17.5200, 37.2900] (in)
    Ball  3: [12.5300,  2.7500] (in)
    Ball  4: [10.4700, 25.1000] (in)
    Ball  5: [27.3200,  4.8700] (in)
    Ball  6: [16.2500, 13.8000] (in)
    Ball  7: [ 7.6000, 21.4900] (in)
    Ball  8: [10.8100,  9.2100] (in)
    Ball  9: [26.1900, 21.8400] (in)
    Ball 10: [19.7800, 26.8100] (in)

Simulation 4: Cue velocity = [-1.499,-0.045] (m/s)
    Final Ball positions [X,Y]:
    Ball  C: [43.1599, 21.3215] (in)
    Ball  1: [30.4163, 20.5811] (in)
    Ball  2: [17.5200, 37.2900] (in)
    Ball  3: [12.5300,  2.7500] (in)
    Ball  4: [10.4700, 25.1000] (in)
    Ball  5: [27.3200,  4.8700] (in)
    Ball  6: [16.2500, 13.8000] (in)
    Ball  7: [ 7.6000, 21.4900] (in)
    Ball  8: [10.8100,  9.2100] (in)
    Ball  9: [18.7316, 21.4707] (in)
    Ball 10: [19.0245, 28.4010] (in)
```

Code 2: MatLab Script — BilliardsCode.m

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

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

%% Setup Simulation parameters
% Set the time interval for calculations
% (Determines precision)
%% FIX: Reset to 1,000,000th
time_slice = 0.0001; % 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;

%% Setup Frame and Video
% Create frame and video objects for storing video frames
latest_frame = 0;
writerObj = VideoWriter(output_dir + 'videos/elasticCollision.avi');
writerObj.FrameRate = frame_rate;
writerObj.Quality = 95;
open(writerObj);

% Background
grid on;
hold on;
axis equal;
axis off;
patch([ 0, 0,88,88],[ 0,44,44, 0], 'g','FaceAlpha',0.5);
axis([0 PoolBall.table_length 0 PoolBall.table_width])

%% Setup Pool Balls
% 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)];
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));
% Log information to output
fprintf("Closest ball to Cue ball: Ball %2s\n", closest_ball.label)
fprintf("Attitude to closest ball: %7.3f deg\n", rad2deg(theta_closest));
fprintf("\n");

% 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
    for j=1: length(balls)
        balls(j).reset;
    end

    % 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)
        % Check if any balls are still moving.
        any_moving=false;
        for j=1:length(balls)
            if (any(balls(j).vel))
                any_moving=true;
                break;
            end
        end
        % If none are moving, stop calculating frames.
        if (not(any_moving))
            break;
        end

        % Increment frame counter
        frame_counter = frame_counter + 1;

        % Calculate motion for each ball
        for j=1:length(balls)
            balls(j).move(time_slice); % Update the position with velocity and apply rolling friction
            balls(j).compute_wall_collisions(); % Compute the velocity changes from wall collisions
            if j+1 <= length(balls) % Collide each ball with all the balls after it in the array
                for k=j+1:length(balls)
                    balls(j).compute_ball_collision(balls(k)); % Compute collision of 2 balls
                end
            end
        end

        % Render and write frames to the video, subject to the frame divider
        if (frame_counter == 0 || mod(frame_counter, frame_divider) == 0)
            for j=1:length(balls)
                balls(j).draw();
            end
            drawnow;
            frame = getframe(gcf); % Save the rendered graph as a video frame
            writeVideo(writerObj, frame); % Write the frame to the video file
        end
    end
```

```matlab

    % Render the last graph, regardless of divider
    for j=1:length(balls)
        balls(j).draw();
    end
    drawnow;

    % Save the final layout of the balls as an image
    exportgraphics(gca,output_dir + sprintf("images/Simulation_%i_end.png", i),'Resolution',600)

    % Add the frame to the video repeatedly for the next 1 second
    % (Creates a 1 second pause between video sections)
    frame = getframe(gcf); % Save the rendered graph as a video frame
    for j=1: frame_rate
        writeVideo(writerObj, frame)
    end

    % 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: [%7.4f, %7.4f] (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(writerObj); % Wrap up the video file
diary off; % Close the diary/log file
```

Code 3: PoolBall Class Definition — PoolBall.m

```matlab
%% 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
```

```
                new_vel = 0;
            end
            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
            obj.vel = [cos(vel_angle) * new_vel, sin(vel_angle) * new_vel]; % Update objects velocity
            obj.pos = obj.pos + effective_vel * time_slice; % Update the objects position
        end
        %% PoolBall Wall Collisions
        % Compute velocity changes due to wall/cushion/bumper collisions
        function compute_wall_collisions(obj)
            max_x = PoolBall.table_length;
            max_y = PoolBall.table_width;
            % Check if we collided with the x=max or x=0 "bumpers"
            if(obj.pos(1) > (max_x - obj.rad) || obj.pos(1) < obj.rad)
                old_vel = obj.vel; % Hold the original velocity for comparison
                obj.vel(1) = -(obj.vel(1)*PoolBall.e_bc); % Invert and reduce the perpendicular velocity by restitution factor
                obj.pos(1) = min(max(obj.pos(1),obj.rad),max_x-obj.rad); % Put the ball back within the limits (Eliminate glitches)
                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
                if (sign(obj.vel(2)) ~= sign(old_vel(2))) % Ensure we didnt reverse direction due to friction
                    obj.vel(2) = 0;
                end
            end
            % Check if we collided with the y=max or y=0 "bumpers"
            if(obj.pos(2) > (max_y - obj.rad) || obj.pos(2) < obj.rad)
                old_vel = obj.vel; % Hold the original velocity for comparison
                obj.vel(2) = -(obj.vel(2)*PoolBall.e_bc); % Invert and reduce the perpendicular velocity by restitution factor
                obj.pos(2) = min(max(obj.pos(2),obj.rad),max_y-obj.rad); % Put the ball back within the limits (Eliminate glitches)
                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
                if (sign(obj.vel(1)) ~= sign(old_vel(1))) % Ensure we didnt reverse direction due to friction
                    obj.vel(1) = 0;
                end
            end
        end
        %% PoolBall Ball Collisions
        % Compute velocity cahnges to ball<->ball collisions
        function compute_ball_collision(ball_a, ball_b)
            % Check if the balls collided
            if (norm(ball_a.pos - ball_b.pos) < (ball_a.rad + ball_b.rad))
                % Check if we are still colliding with this ball
                %   Approximation method needed for time-slice based approach.
                %   This avoids calculating multiple collisions while balls are
                %   clipped through eachother. Smaller time-slice sizes reduces
                %   approximation error here.
                if (~isempty(ball_a.colliding_balls))
                    for i=1: length(ball_a.colliding_balls)
                        if (isequal(ball_b, ball_a.colliding_balls(i)))
                            return;
                        end
                    end
                end
                % Else, add the ball to the reference array
                ball_a.colliding_balls(end+1) = ball_b;
                % Check if either ball is the cue ball, select the correct e
                if (ball_a.is_cue || ball_b.is_cue)
                    e = PoolBall.e_bc; % Coefficient for cue and numbered ball
                else
                    e = PoolBall.e_nn; % Coefficient for 2 numbered balls
                end
```

```
            % Get the angles and velocities, relative to the impact
            difference = ball_b.pos - ball_a.pos;
            impact_angle = atan2(difference(2), difference(1));
            ball_a_v = norm(ball_a.vel); % Absolute velocity of A
            ball_a_v_angle = atan2(ball_a.vel(2), ball_a.vel(1)); % Angle of velocity of A
            ball_b_v = norm(ball_b.vel); % Absolute velocity of B
            ball_b_v_angle = atan2(ball_b.vel(2), ball_b.vel(1)); % Angle of velocity of B
            ball_a_norm_v = cos(ball_a_v_angle - impact_angle) * ball_a_v; % norm portion of A velocity (compared to impact direction)
            ball_a_tan_v = sin(ball_a_v_angle - impact_angle) * ball_a_v;  % tan portion of A velocity (compared to impact direction)
            ball_b_norm_v = cos(ball_b_v_angle - impact_angle) * ball_b_v; % norm portion of B velocity (compared to impact direction)
            ball_b_tan_v = sin(ball_b_v_angle - impact_angle) * ball_b_v;  % tan portion of B velocity (compared to impact direction)
            % Create system of equations for the normal velocities, and solve
            syms new_ball_a_norm_v new_ball_b_norm_v
            norm_momentum = ball_a_norm_v * ball_a.mass + ball_b_norm_v * ball_b.mass == new_ball_a_norm_v * ball_a.mass + new_ball_b_norm_v * ball_b.mass;
            norm_restitution = e == (new_ball_a_norm_v - new_ball_b_norm_v)/(ball_b_norm_v - ball_a_norm_v);
            [A,B] = equationsToMatrix([norm_momentum, norm_restitution], [new_ball_a_norm_v, new_ball_b_norm_v]);
            X = linsolve(A,B); % Solve system
            new_ball_a_norm_v = X(1); % Post-impact normal velocity of A
            new_ball_b_norm_v = X(2); % Post-impact normal velocity of B
            % Determine frictional losses on tangent velocities
            new_ball_a_tan_v = ball_a_tan_v - sign(ball_a_tan_v)*(PoolBall.mu_bb * (ball_a_norm_v - new_ball_a_norm_v));
            new_ball_b_tan_v = ball_b_tan_v - sign(ball_b_tan_v)*(PoolBall.mu_bb * (ball_b_norm_v - new_ball_b_norm_v));
            % Ensure frictional loss didnt change sign of velocity
            if (sign(new_ball_a_tan_v) ~= sign(ball_a_tan_v))
                new_ball_a_tan_v = 0;
            end
            if (sign(new_ball_b_tan_v) ~= sign(ball_b_tan_v))
                new_ball_b_tan_v = 0;
            end
            % Apply new velocities to balls
            ball_a.vel = [double(cos(impact_angle)*new_ball_a_norm_v - cos(pi/2 - impact_angle)*new_ball_a_tan_v), ...
                          double(sin(impact_angle)*new_ball_a_norm_v + sin(pi/2 - impact_angle)*new_ball_a_tan_v)];
            ball_b.vel = [double(cos(impact_angle)*new_ball_b_norm_v - cos(pi/2 - impact_angle)*new_ball_b_tan_v), ...
                          double(sin(impact_angle)*new_ball_b_norm_v + sin(pi/2 - impact_angle)*new_ball_b_tan_v)];
            return;
        % If this ball isnt overlapping, Check if we were previously colliding with it
        elseif (~isempty(ball_a.colliding_balls)) % Make sure array isnt empty
            for i=1: length(ball_a.colliding_balls)
                if (isequal(ball_b, ball_a.colliding_balls(i))) % Compare object references
                    ball_a.colliding_balls(i) = []; % Delete the ball reference
                end
            end
        end
    end
    %% PoolBall Draw
    % Draw the ball on the graph at the current position
    function draw(obj)
        a=[0:0.1:2*pi];
        Xcircle=cos(a);
        Ycircle=sin(a);
        set(obj.ball_img,'XData',obj.pos(1)+obj.rad*Xcircle, 'YData', obj.pos(2)+obj.rad*Ycircle);
```

```matlab
            set(obj.spot_img,'XData',obj.pos(1)+obj.rad*Xcircle/2, 'YData', obj.pos(2)+obj.rad*Ycircle/2);
            set(obj.label_img,'Position', [obj.pos(1) obj.pos(2) 0]);
        end
        %% PoolBall Reset
        % Reset the position and velocity of the ball
        function reset(obj)
            obj.pos = obj.home_pos;
            obj.vel = [0,0];
            obj.draw();
        end
        %% PoolBall Set Velocity
        % Set the velocity of the pool ball (Clearer than value accessor)
        function set_vel(obj, vel)
            obj.vel = vel;
        end
        %% PoolBall Find Closest
        % Determine the closest ball from a given list of ball (Returns ball reference)
        function closest = find_nearest(obj, balls)
            closest = PoolBall.empty; % Empty reference
            min_dist = norm([PoolBall.table_length, PoolBall.table_width]); % Set to max value
            for i=1: length(balls)
                distance = norm(balls(i).pos - obj.pos);
                if (distance < min_dist) % Check if this ball is closer
                    min_dist = distance; % Update the minimum distance
                    closest = balls(i); % Update the object reference
                end
            end
        end
    end
end
```