Billiards Simulation

1 Introduction

Billiards is a casual game played to pass the time. The game is played with 16 balls. There are two sets of balls plus two special balls. You choose which set you want to play with after the first move of the game. The balls numbered 1-7 are solid and the balls numbered 9-15 are striped. To play you use a special stick called a cue stick to hit the cue ball. The cue ball, one of the special balls, is the only white ball on the table. The objective of the game is to get balls in your group, either solids or stripes, to fall into pockets located on the edge of the pool table by knocking them in using the cue ball. Once you have gotten all of your balls to fall into the pockets, then you are allowed to knock the final ball in, the 8-ball, which is the only black ball on the table and is the second special ball. If you knock all your balls in plus the 8-ball before your opponent does, then you win the game. There are various ways to play and different sets of rules that I will not go over here.

The game itself is speculated to have originated sometime during the 15th century. Billiards is also known as pool and it was the first sport to have a world championship in 1873. There are several governing organizations for competitive billiards, one of which is the World Pool-Billiard Association (WPA). They have events and also a list of regulations for pool tables that set the standard for pool tables as well as regulations for pool balls. For the most part, there are two types of tables, 8 foot tables and 9 foot tables. The both have playable dimensions of Lx2L (area between the bumpers). Balls are roughly 2 and a quarter inch in diameter and weigh approximately 6 ounces.

In my simulation I used constants to define the playing area, table area, ball radius, and ball weight that are all in accordance with the WPA regulations. The physics which are relevant for billiards are Newtonian Mechanics. I could not find a regulation on what the coefficients of friction and restitution were for various parts of the table, so I used values found in *Physics of Pool and Billiards*. I also made several assumptions in my simulation to simplify the model.

Those are:

- When you take a cue shot, the cue stick is in contact with the ball for .01 s
- Ball on ball collisions are elastic
- Ball on ball collisions produce no torque (ball on ball contact is friction-less)
- Ball on rail collisions produce no torque (ball on rail contact is friction-less)
- There is no linear motion or force along the z-axis besides gravity (ie. ball does not jump or bounce)

Here are the constants used for the simulation:

- When you take a cue shot, the cue stick is in contact with the ball for .01 s

- Table Width: 1.2716 m
- Table Length: 2.4416 m
- Thickness of the Railings: .0508 m
- Playable Area Width: 1.17 m
- Playable Area Length; 2.34 m
- Gravity: 9.81 ms**-2
- Radius of Balls: .028575 m
- Mass of Balls: .160 kg
- Moment of Inertia of the Balls: .4 * Mass * Radius**2 = .05225796
- Kinetic Friction Coefficient Between the Ball and the Felt: .2
- Rolling Friction Coefficient Between the Ball and the Felt: .01
- Coefficient of Restitution for Bumper to Ball Collision: .7

2 Models

The model I created is based off of Newtonian Mechanics. The model can be expressed by dividing it into 4 categories, the shot, basic motion, collision detection, and collision handling.

2.1 The Shot

The shot involves hitting the cue ball with a cue stick perfectly horizontally (meaning no force is exerted along the z-axis by the cue stick). For the current state of the simulation, only one shot can be performed and that is at the very beginning of the simulation. As previously mentioned, the time of contact between the ball and the cue tip is assumed to be a constant .01s. This is a pure guess on how long they should be in contact and is not based on any research. The force vector the cue stick exerts on the cue ball can be any 3-dimensional vector so long as the z component is 0. The initial vectors of the ball are all considered to be zero if there is a cue shot (in other words the ball being stroked is motionless when stroked). The calculation for the initial position, initial linear velocity, and initial angular velocity of the ball after the cue strike goes as follows:

Position:

$$\vec{F} = m\vec{a}, \vec{a} = \frac{\vec{F}}{m} \tag{1}$$

$$\Delta\vec{p} = \vec{v_0}\Delta t + \frac{1}{2}\vec{a}\Delta t^2, \vec{p_1} - \vec{p_0} = \vec{v_0}\Delta t + \frac{1}{2}\vec{a}\Delta t^2 \tag{2}$$

$$\vec{v_0} = \vec{0}, \vec{p_0} = \vec{0}, \vec{p_1} = \frac{1}{2}\vec{a}\Delta t^2 \tag{3}$$

Linear Velocity:

$$\vec{F} = m\vec{a} = m\frac{\Delta\vec{v}}{\Delta t} \tag{4}$$

$$\Delta\vec{v} = \frac{\vec{F}}{m}\Delta t, \vec{v_1} - \vec{v_0} = \frac{\vec{F}}{m}\Delta t \tag{5}$$

$$\vec{v_0} = \vec{0}, \vec{v_1} = \frac{\vec{F}}{m}\Delta t \tag{6}$$

Angular Velocity:

$$\tau = I\alpha, \vec{\tau} = \vec{r} \times \vec{F} \tag{7}$$

where r represents the vector from the center point of the cue ball to the outer surface. r is fairly easy to calculate. Depending on your point of reference you always know two of its dimensions. Two of the dimensions are simply the displacement from the center of the ball. Since the magnitude of the vector is always equal to the radius of the ball, the thrid is simply equal to the square root of the radius squared minus the knowns squared.:

$$\vec{r} = \begin{pmatrix} \sqrt{R^2 - \texttt{known1}^2 - \texttt{known2}^2} \\ \texttt{known1} \\ \texttt{known2} \end{pmatrix} \tag{8}$$

$$I\vec{\alpha} = \vec{r} \times \vec{F}, \vec{\alpha} = \frac{\vec{r} \times \vec{F}}{I} \tag{9}$$

$$\vec{\alpha} = \frac{\Delta\vec{\omega}}{\Delta t}, \vec{\omega_1} - \vec{\omega_0} = \vec{\alpha}\Delta t \tag{10}$$

where R is the radius of the ball I is the moment of inertia

$$I = \frac{2}{5}mR^2, \vec{\omega_0} = \vec{0} \tag{11}$$

$$\vec{\omega_1} = \vec{\alpha}\Delta t \tag{12}$$

2.2 Basic Motion

Basic motion of the billiard ball describes the movement of a ball when no collisions occur. When the ball is not undergoing a collision, the only force acting on it is friction. The direction of friction is dependent on the perimeter velocity of the ball at its point of contact with the table. The force of friction will be opposite to this velocity. Perimeter velocity is given by:

$$\vec{v_p} = (\vec{\omega} \times \vec{r}) + \vec{v} \tag{13}$$

where w is the angular velocity of the ball, v is the linear velocity of the ball (velocity of the center of mass), and r is the vector representing the line from the center of the ball to the point at which its touching the table, which is always:

$$\vec{r} = \begin{pmatrix} 0 \\ 0 \\ -radius \end{pmatrix} \tag{14}$$

Friction:

$$F_f = \mu m g, \vec{F_f} = -\mu m g \frac{\vec{v_p}}{|\vec{v_p}|} \tag{15}$$

where mu is the coefficient of friction which we will analyze later m is mass and g is gravity. Because it is the only force acting on the ball:

$$\vec{F_f} = m\vec{a} \tag{16}$$

$$-\mu m g \frac{\vec{v_p}}{|\vec{v_p}|} = m\vec{a} \tag{17}$$

Linear Acceleration:

$$\vec{a} = -\mu g \frac{\vec{v_p}}{|\vec{v_p}|} \tag{18}$$

Update Position

$$\vec{p} = \vec{p_o} + \vec{v_0}\Delta t + \frac{1}{2}\vec{a}\Delta t^2 \tag{19}$$

Update Linear Velocity:

$$\vec{v} = \vec{v_0} + \vec{a}\Delta t \tag{20}$$

Angular Acceleration:

$$\vec{\tau_f} = \vec{r} \times \vec{F_f}, \vec{\tau_f} = \vec{r} \times (-\mu m g \frac{\vec{v_p}}{|\vec{v_p}|}) = I\vec{\alpha} \tag{21}$$

$$\vec{\alpha} = \frac{\vec{r} \times (-\mu m g \frac{\vec{v_p}}{|\vec{v_p}|})}{I} \tag{22}$$

Update Angular Velocity

$$\vec{\omega} = \vec{\omega_0} + \vec{\alpha}\Delta t \tag{23}$$

where Δ t = the time step for a given iteration.

Now, some analysis is needed to determine the coefficient of friction, mu, that should be used. The calculations above work at any given time but are influenced by different mus depending on whether or not the ball is sliding or in pure roll. For a sliding ball,

$$\mu = .2 \tag{24}$$

and for a ball in pure roll,

$$\mu = .01 \tag{25}$$

A ball is considered to be in pure roll if

$$|\vec{v_p}| = R * |\vec{\omega}| \tag{26}$$

For the purpose of the simulation however, if this condition is true,pure roll has begum:

$$(|\vec{v_p}| - R * |\vec{\omega}|) < .01 \tag{27}$$

This is a fairly large margin of error, but it is to assure that the start of pure roll is not missed.

2.3 Collision Detection

Detecting collisions comes in the shape of detecting ball on ball collisions and ball on wall collisions. For ball on ball collisions:

If after calculating the new ball positions, the distance between the center of any ball and any other ball is less than or equal to *2R*, then they have collided. The distance formula is given by:

$$\sqrt{(\vec{p_1} - \vec{p_2}) \cdot (\vec{p_1} - \vec{p_2})} <= 2R \tag{28}$$

The result of the dot product inside the square root gives

$$(p_{1x} - p_{2x})^2 + (p_{1p} - r_{2y})^2 + 0^2 \tag{29}$$

If after calculating the new ball positions, the distance between the center of any ball and any rail is less than or equal to R, then they have collided. To check for rail conditions simply check:

$$(\vec{p} - radius) <= \overrightarrow{\texttt{Defined Lower Bounds}} \tag{30}$$

for any element in p or

$$(\vec{p} + radius) >= \overrightarrow{\texttt{Defined Upper Bounds}} \tag{31}$$

for any element in *(*p), a wall collision has occurred. Part of detecting collisions is also detecting the one that comes first. To find that I need to compare the amount of time it takes for each collision and find which one takes the least amount of time. For a ball on ball collision you can find the time by doing this:

Although less accurate it's much easier to solve for t using Euler's scheme for consecutive positions, so for figuring out time until collision we assume:

$$\vec{p} = \vec{p_o} + \vec{v_0}\Delta t \tag{32}$$

You can then expand equation (28) to obtain an equation with a time variable:

$$\sqrt{(\vec{p_{01}} - \vec{p_{02}}) \cdot (\vec{p_{01}} - \vec{p_{02}}) + 2(\vec{p_{01}} - \vec{p_{02}}) \cdot (\vec{v_1} - \vec{v_2})t + (\vec{v_1} - \vec{v_2}) \cdot (\vec{v_1} - \vec{v_2})t^2} = 2R \tag{33}$$

$$(\vec{p_{01}} - \vec{p_{02}}) \cdot (\vec{p_{01}} - \vec{p_{02}}) + 2(\vec{p_{01}} - \vec{p_{02}}) \cdot (\vec{v_1} - \vec{v_2})t + (\vec{v_1} - \vec{v_2}) \cdot (\vec{v_1} - \vec{v_2})t^2 = 4R^2 \tag{34}$$

$$(\vec{v_1}-\vec{v_2})\cdot(\vec{v_1}-\vec{v_2})t^2+2(\vec{p_{01}}-\vec{p_{02}})\cdot(\vec{v_1}-\vec{v_2})t+(\vec{p_{01}}-\vec{p_{02}})\cdot(\vec{p_{01}}-\vec{p_{02}})-4R^2=0 \tag{35}$$

We can then see that there is a quadratic solution to the equation and make these substitutions:

$$a=(\vec{v_1}-\vec{v_2})\cdot(\vec{v_1}-\vec{v_2}) \tag{36}$$

$$b=2(\vec{p_{01}}-\vec{p_{02}})\cdot(\vec{v_1}-\vec{v_2}) \tag{37}$$

$$c=(\vec{p_{01}}-\vec{p_{02}})\cdot(\vec{p_{01}}-\vec{p_{02}})-4R^2 \tag{38}$$

$$at^2+bt+c=0 \tag{39}$$

We can now solve using the quadratic equation:

$$t=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{40}$$

We receive two answers from this so we take the smallest non-negative answer in all cases. For Ball on rail collisions we find t simply by solving for t for these two equations:

$$\texttt{Lower Bound}=p_0+v\Delta t-R, \texttt{Upper Bound}=p_0+v\Delta t+R \tag{41}$$

$$t=\frac{\texttt{Lower Bound}+R-p_0}{v}, t=\frac{\texttt{Upper Bound}-R-p_0}{v} \tag{42}$$

(41) and (42) deal with only the element of the vector in question (if the ball is above the maximum y value of the playing area, the the the Upper Bound equation is used as is the y element of the vectors p, v, and *Upper Bound*)

2.4 Collision Handling

Once all the collisions are found and the soonest one is established, all of the equations for Basic motion are recalculated using the smallest time until a collision occurs. Then, the collision handler deals with the collision in question. For a ball on ball collision: Finding the normal vector: The easiest way is to normalize the difference in position vectors

$$\vec{n}=\frac{\vec{p_1}-\vec{p_2}}{|\vec{p_1}-\vec{p_2}|} \tag{43}$$

The next step is to split the velocity vectors into normal and tangential components. The normal component of the second ball will oriented the same direction as the normal while the normal component of the first ball will be in the opposite direction of the normal. The magnitudes of these vectors can be found through dot products and then given direction by multiplying the magnitude times a directional vector:

$$\vec{v_{n1}}=(\vec{v_1}\cdot(-\vec{n}))(-\vec{n}) \tag{44}$$

$$\vec{v_{n2}}=(\vec{v_2}\cdot\vec{n})(\vec{n}) \tag{45}$$

Once we've found the normal components of the velocities, whatever is left is the tangential component and it can be found by subtracting the normal component from the original velocity vector:

$$\vec{v_{t1}} = v_1 - \vec{v_{n1}} \tag{46}$$

$$\vec{v_{t2}} = v_2 - \vec{v_{n2}} \tag{47}$$

Then because we assume ball on ball collisions are elastic, to compute the new velocities we simply switch the normal components and then add the tangent component back in:

$$\vec{v_1}' = v_{t1} + \vec{v_{n2}} \tag{48}$$

$$\vec{v_2}' = v_{t2} + \vec{v_{n1}} \tag{49}$$

The ball on ball collision has now been handled and the simulation can proceed with the next time step.

Now for ball on rail collisions you simply have to negate the component of velocity that is touching the bound and then multiply it by the coefficient of restitution for the rail.

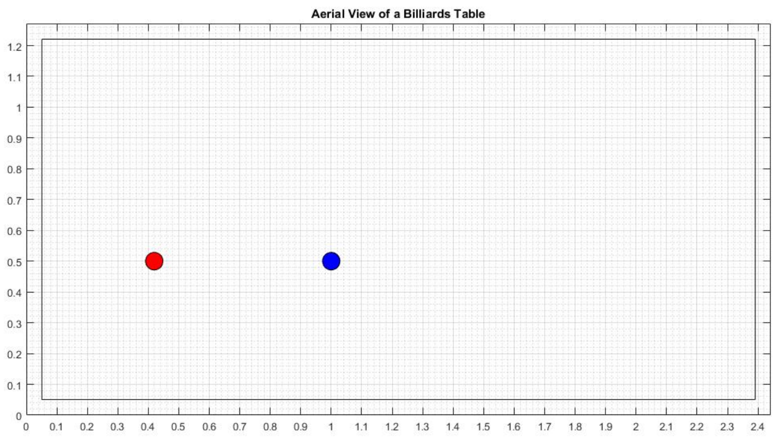
As mentioned in the introduction, I ignore torque during collisions.

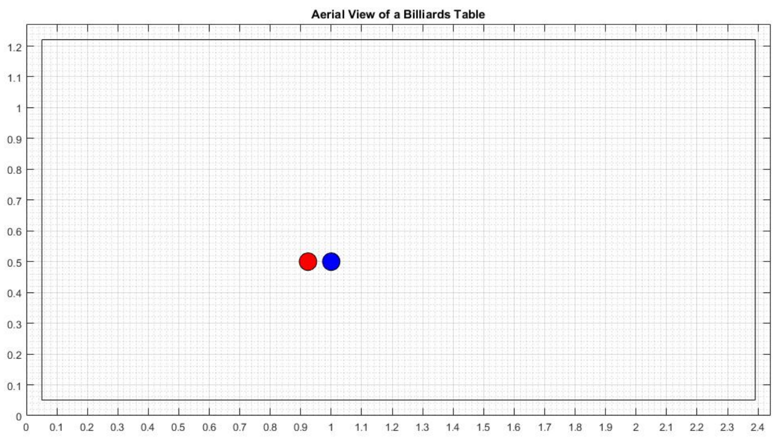
3 Results

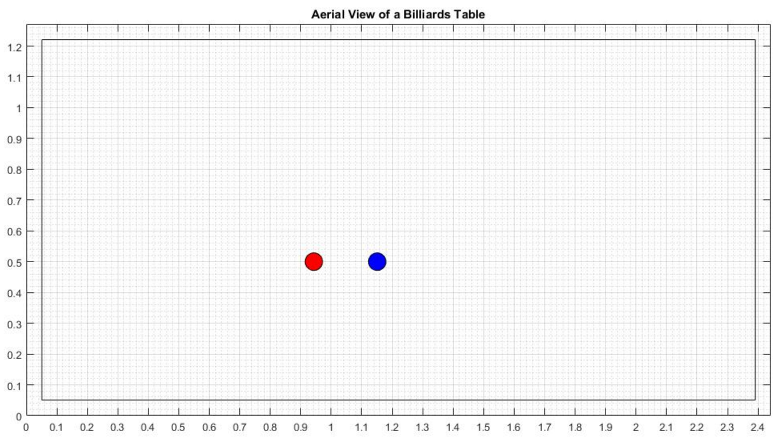
In `MatLab`, my simulation code is invoked at the prompt by calling the function `RunSimulation( sim, timeStep, simLength )` where *sim* is the simulation number, *timeStep* is the time interval between each calculation, and *simLength* is the real world time in seconds the simulation covers.

sim has three values.

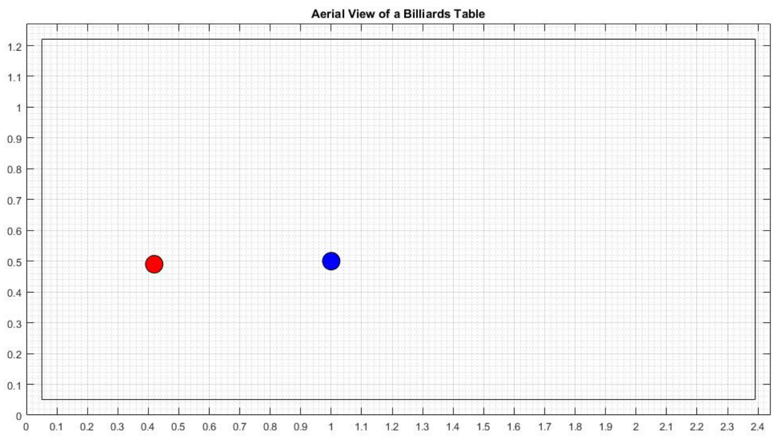
1. 1 corresponds to a head on collision between the read ball and the blue ball. The blue ball is initially at rest and the red ball is initially moving with a magnitude of 5. Here are pictures of sim 1. Initial position, right before collision, and right after respectively.

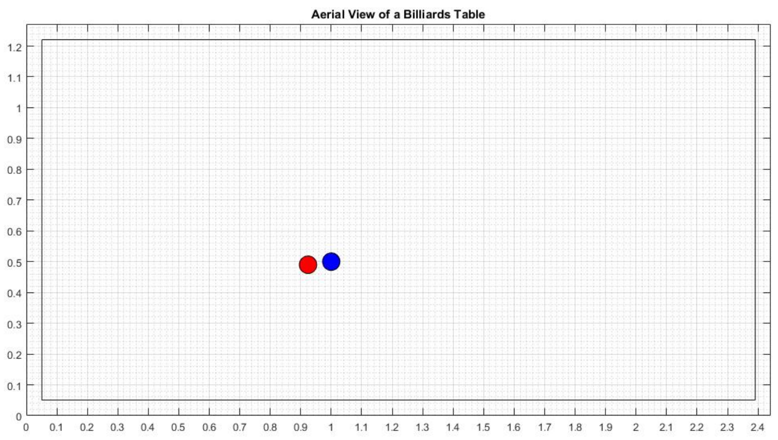


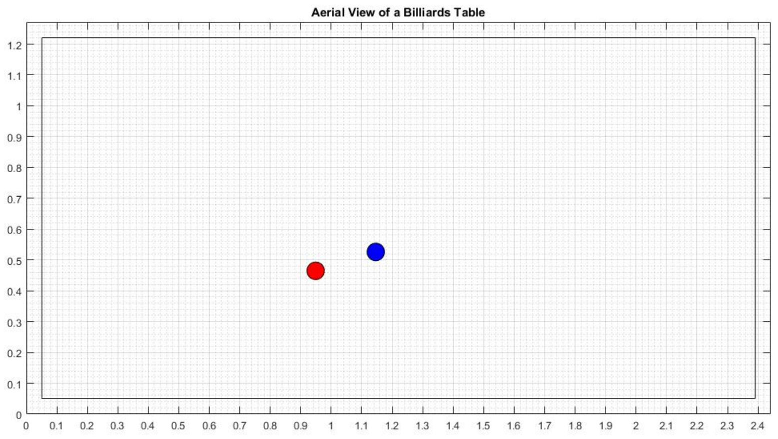




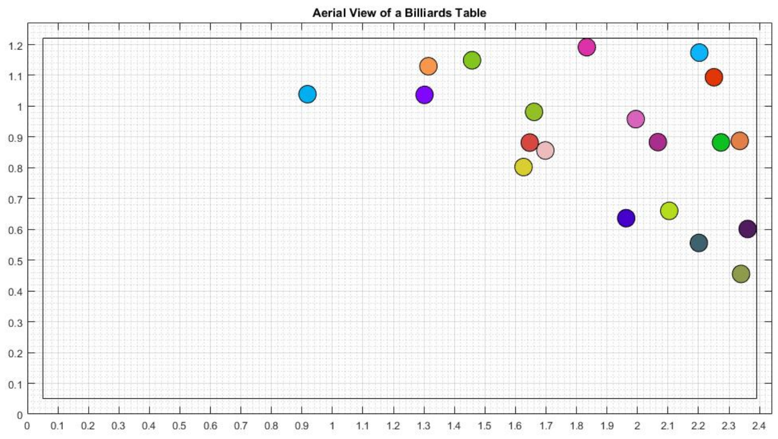
2. 2 corresponds to a glancing collision between the red ball and the blue ball. The blue ball is initially at rest and the red ball is initially moving with a magnitude of 5. Here are pictures of sim 2. Initial position, right before collision, and right after respectively.



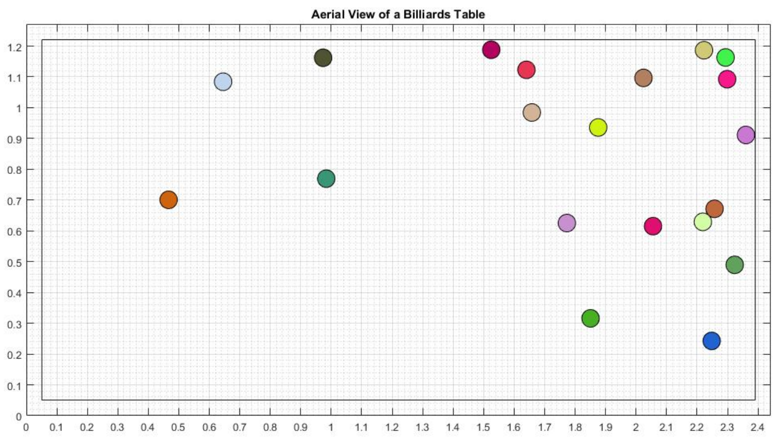




3. 3 corresponds to something more interesting. 20 randomly colored balls are created and they are given random legal positions and random velocities to start out with. Here are pictures of sim 3. The simulation is different everytime so it is just a snapshot of multiple



balls on a table at once.



4 Conclusion

My code works well, especially for a 2 ball system. However, there are several problems with my code. For example, if you create two balls and set their initial positions to where they are touching each other, there is a chance that when hit by a third ball, they pass over each other. Also my code does not account for the change in angular momentum during collisions. Furthermore my current collision handling process is an O(n**2) algorithm. Even with only 20 balls on the table, things can get very slow when there are lots of balls that are near collision. Also, I do not know whether or not my code can handle Zeno's Phenomenon. It would be safe to assume it can't because I have not set a limit on how low to let the velocity get before stopping a ball's movement. Lastly there are no comments in the code, as I was very busy and did not have time to brush things up, so even though the code is well tabbed and organized a lack of comments may leave readers lost. However, my code follows exactly the methods I described in the Models section so feel free to refer back to it. I wish I had more time to work on this project. It was very interesting and I was hoping to add pockets to the table for balls to fall in. On a final note, the code I believe is very powerful and the math and physics are sound. Improvements can definitely be made, but I am happy with where I got to with this simulation.

A Implementation in MatLab

The `MatLab` implementation with (very few) comments is given here:

```
function RunSimulation( sim, timeStep, simLength )

    if(sim==1)
        %CREATE THE TWO BILLIARD BALLS
        r = BilliardBall([.3,.5,0],[5,0,0],[0,0,0],[0,0,0],[0,0,0], true, [1 0 0]); %RED
        b = BilliardBall([1,.5,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0], false,[0 0 1]); %BLUE
        %CREATE BALL ARRAY FOR SIM
        ballArray = [r b];
        numBalls  = 2;

        BilliardSim(timeStep,ballArray(1:numBalls),numBalls,0,0,0,simLength,1200,600);
    end
    if(sim==2)
        %CREATE THE TWO BILLIARD BALLS
        r = BilliardBall([.3,.49,0],[5,0,0],[0,0,0],[0,0,0],[0,0,0], true, [1 0 0]); %RE
        b = BilliardBall([1,.5,0],[0,0,0],[0,0,0],[0,0,0],[0,0,0], false,[0 0 1]); %BLUE
        %CREATE BALL ARRAY FOR SIM
        ballArray = [r b];
        numBalls  = 2;

        BilliardSim(timeStep,ballArray(1:numBalls),numBalls,0,0,0,simLength,1200,600);
    end
    if(sim==3)
        %CREATE 20 BILLIARD BALLS
        for i=1:1:20
            ballArray(i) = BilliardBall([rand * 2.20,rand * 1.17,0],...
                                        [rand * 7, rand * 7, 0],[0,0,0],...
                                        [0,0,0],[0,0,0], false,[rand rand rand]);
        end
        numBalls = 20;
        BilliardSim(timeStep,ballArray(1:numBalls),numBalls,0,0,0,simLength,1200,600);
    end

end
```

```
function magnitude = magnitude1x3( vector )
%magnitude1x3 COMPUTES THE MAGNITUDE OF A VECTOR SZE 1 ROW x 3 COLUMNS
%   Input:
%       vector: THE VECTOR THAT NEEDS ITS MAGNITUDE CALCULATED
%   Output:
%       magnitude: A SCALAR QUANTITY THAT REPRESENTS THE MAGNITUDE OF vector
    magnitude = sqrt((vector(1)^2+vector(2)^2+vector(3)^2));

end




function newBallArray = updateVectors( timeStep, ballArray, numBalls, ...
                                        cueForce, ystrike, zstrike, ...
                                        bounds0, boundsMax )
%updateVectors UPDATES THE POSITION VELOCITY AND ACCELERATION VECTORS OF THE
%BILLIARDS BALLS ON THE TABLE
%   ARGUMENTS:
%       ballArray: IS AN ARRAY OF BILLIARD BALL OBJECTS
%   OUTPUT:
%       newBallArray: IS AN ARRAY OF BILLIARD BALLS WITH UPDATED VECTORS

    global FirstCollisionTime;

    ball1 = 0;
    ball2 = 0;
    CollisionType = 0;
    beyondLowerBounds = [0,0,0];
    beyondUpperBounds = [0,0,0];
    FirstBeyondLowerBounds = [0,0,0];
    FirstBeyondUpperBounds = [0,0,0];
    gravity = 9.81;

    if(dot(cueForce,cueForce))
        %IF ITS THE FIRST SHOT, FIND CUEBALL
        for i = 1: 1: numBalls
            if ballArray(i).isCueBall
                cueBall = i;
                break;
            end
        end
        %UPDATE CUEBALL VECTORS
        ballArray(cueBall).velocity = (cueForce / BilliardBall.MASS) * .01;
        ballArray(cueBall).position = ballArray(cueBall).position + ...
                                      .5 * (cueForce / BilliardBall.MASS) * ...
                                      .01^2;


        Rvector = [sqrt(BilliardBall.RADIUS^2-ystrike^2-zstrike^2), ystrike, ...
                                                                    zstrike];
        ballArray(cueBall).angularAcceleration = cross(Rvector,cueForce) ...
                                                 / BilliardBall.MOMENT_INERTIA;
        ballArray(cueBall).angularVelocity = ...
                               ballArray(cueBall).angularAcceleration * .01;
        ballArray(cueBall);
    end

    for i = 1: 1: numBalls
        tempArray(i) = ballArray(i);
    end

    %CREATE POSSIBLE RETURN VALUE
    for i = 1: 1: numBalls
        R = [0,0,-BilliardBall.RADIUS];
        vp = cross(tempArray(i).angularVelocity,R) + tempArray(i).velocity;
        mew = BilliardBall.KINETIC_FRICTION;
        if (magnitude1x3(vp) - BilliardBall.RADIUS * ...
                magnitude1x3(tempArray(i).angularVelocity)<.01)
            mew = BilliardBall.ROLLING_FRICTION;
        end
        if magnitude1x3(vp)
            friction = (-mew * BilliardBall.MASS * gravity / magnitude1x3(vp)) * vp;
        else
            friction = [0,0,0];
        end
        tempArray(i).acceleration = friction / BilliardBall.MASS;
        tempArray(i).position = tempArray(i).position ...
                               + tempArray(i).velocity * timeStep...
                               + .5 * tempArray(i).acceleration * timeStep^2;
        tempArray(i).velocity = tempArray(i).velocity ...
                               + tempArray(i).acceleration * timeStep;
        tempArray(i).angularAcceleration = cross(R,friction) ...
                                           / BilliardBall.MOMENT_INERTIA;
        tempArray(i).angularVelocity = tempArray(i).angularVelocity ...
                                 + tempArray(i).angularAcceleration * timeStep;

    end

    %CHECK FOR FIRST COLLISION
    for i = 1: 1: numBalls
        for j = i: 1: numBalls
            if j~=i
                if (dot((tempArray(i).position-tempArray(j).position), ...
                       (tempArray(i).position-tempArray(j).position)) ...
                       <= 4 * BilliardBall.RADIUS^2)

                   a = dot((ballArray(i).velocity-ballArray(j).velocity), ...
                           (ballArray(i).velocity-ballArray(j).velocity));
                   b = dot((ballArray(i).position-ballArray(j).position), ...
                           (ballArray(i).velocity-ballArray(j).velocity)) * 2;
                   c = dot((ballArray(i).position-ballArray(j).position), ...
                           (ballArray(i).position-ballArray(j).position)) ...
                         - 4 * BilliardBall.RADIUS^2;
                   CollisionTime1 = (-b + sqrt(b^2 - 4*a*c)) / (2 * a);
                   CollisionTime2 = (-b - sqrt(b^2 - 4*a*c)) / (2 * a);

                   if CollisionTime1 < 0
                       CollisionTime1 = 100;
                   end
                   if CollisionTime2 < 0
                       CollisionTime2 = 100;
                   end
                   if CollisionTime1 < CollisionTime2
                       CollisionTime = CollisionTime1;
                   else
                       CollisionTime = CollisionTime2;
                   end
                   if CollisionTime < FirstCollisionTime && CollisionTime > 0
                       FirstCollisionTime = CollisionTime;
                       CollisionType = 1;
                       ball1 = i;
                       ball2 = j;
                   end
                end
            end
```

```
end

beyondLowerBounds = ((tempArray(i).position - BilliardBall.RADIUS)...
                    <= bounds0);
beyondUpperBounds = ((tempArray(i).position + BilliardBall.RADIUS)...
                    >= boundsMax);
CollisionTime1    = 100;
CollisionTime2    = 100;
if (beyondLowerBounds * [1;1;0] > 0)
    if (beyondLowerBounds * [1;0;0]) == 1
        CollisionTime1 = (bounds0 * [1;0;0] + BilliardBall.RADIUS ...
                        - ballArray(i).position * [1;0;0]) ...
                        /(ballArray(i).velocity * [1;0;0]);
    end
    if (beyondLowerBounds * [0;1;0]) == 1
        CollisionTime2 = (bounds0 * [0;1;0] + BilliardBall.RADIUS ...
                        - ballArray(i).position * [0;1;0]) ...
                        /(ballArray(i).velocity * [0;1;0]);
    end
    if CollisionTime1 < CollisionTime2
        CollisionTime = CollisionTime1;
    else
        CollisionTime = CollisionTime2;
    end
    if CollisionTime < FirstCollisionTime
        FirstCollisionTime = CollisionTime;
        if CollisionTime1 < CollisionTime2
            FirstBeyondLowerBounds = [1,0,0];
        else
            FirstBeyondLowerBounds = [0,1,0];
        end
        CollisionType = 2;
        ball1 = i;
        ball2 = 0;
    end
end

if (beyondUpperBounds * [1;1;0] > 0)
    if (beyondUpperBounds * [1;0;0]) == 1
        CollisionTime1 = (boundsMax * [1;0;0] - BilliardBall.RADIUS ...
                        - ballArray(i).position * [1;0;0]) ...
                        /(ballArray(i).velocity * [1;0;0]);
            end
            if (beyondUpperBounds * [0;1;0]) == 1
                CollisionTime2 = (boundsMax * [0;1;0] - BilliardBall.RADIUS ...
                                - ballArray(i).position * [0;1;0]) ...
                                /(ballArray(i).velocity * [0;1;0]);
            end
            if CollisionTime1 < CollisionTime2
                CollisionTime = CollisionTime1;
            else
                CollisionTime = CollisionTime2;
            end
            if CollisionTime < FirstCollisionTime
                FirstCollisionTime = CollisionTime;
                if CollisionTime1 < CollisionTime2
                    FirstBeyondUpperBounds = [1,0,0];
                else
                    FirstBeyondUpperBounds = [0,1,0];
                end
                CollisionType = 3;
                ball1 = i;
                ball2 = 0;
            end
        end
    end

    % IF COLLISION, DEAL WITH COLLISION
    if (FirstCollisionTime <= timeStep)
        for i = 1: 1: numBalls
            tempArray(i) = ballArray(i);
        end
        %REDO tempArray WITH NEW TIMESTEP
        for i = 1: 1: numBalls
            R = [0,0,BilliardBall.RADIUS];
            vp = cross(tempArray(i).angularVelocity,R) + tempArray(i).velocity;
            mew = BilliardBall.KINETIC_FRICTION;
            if (magnitude1x3(tempArray(i).velocity) <= BilliardBall.RADIUS * ...
                              magnitude1x3(tempArray(i).angularVelocity))
                mew = BilliardBall.ROLLING_FRICTION;
            end
            if magnitude1x3(vp)
            friction = (-mew * BilliardBall.MASS * gravity / magnitude1x3(vp)) * vp;
            else
        friction = [0,0,0];
        end
        tempArray(i).acceleration = friction / BilliardBall.MASS;
        tempArray(i).position = tempArray(i).position ...
                              + tempArray(i).velocity...
                              * FirstCollisionTime...
                              + .5 * tempArray(i).acceleration ...
                              * FirstCollisionTime^2;
        tempArray(i).velocity = tempArray(i).velocity ...
                              + tempArray(i).acceleration ...
                              * FirstCollisionTime;
        tempArray(i).angularAcceleration = cross(R,friction) ...
                                         / BilliardBall.MOMENT_INERTIA;
        tempArray(i).angularVelocity = tempArray(i).angularVelocity ...
                                     + tempArray(i).angularAcceleration ...
                                     * FirstCollisionTime;
    end

    %HANDLE COLLISIONS
    if CollisionType == 1

        n = (tempArray(ball1).position - tempArray(ball2).position)...
          / (magnitude1x3((tempArray(ball1).position ...
                       - tempArray(ball2).position)));
        v_n1 = dot(tempArray(ball1).velocity,(-n)) * (-n);
        v_n2 = dot(tempArray(ball2).velocity,n) * n;
        v_t1 = tempArray(ball1).velocity - v_n1;
        v_t2 = tempArray(ball2).velocity-v_n2;

        tempArray(ball1).velocity = v_t1 + v_n2;
        tempArray(ball2).velocity = v_t2 + v_n1;
    end

    if CollisionType == 2
        if (FirstBeyondLowerBounds * [1;0;0]) == 1
            tempArray(ball1).velocity = tempArray(ball1).velocity * ...
                         [-BilliardBall.BUMPER_RESTITUTION,0,0;0,1,0;0,0,1];
        end
        if (FirstBeyondLowerBounds * [0;1;0]) == 1
            tempArray(ball1).velocity = tempArray(ball1).velocity * ...
                         [1,0,0;0,-BilliardBall.BUMPER_RESTITUTION,0;0,0,1];
        end
        end

        if CollisionType == 3
            if (FirstBeyondUpperBounds * [1;0;0]) == 1
                tempArray(ball1).velocity = tempArray(ball1).velocity * ...
                             [-BilliardBall.BUMPER_RESTITUTION,0,0;0,1,0;0,0,1];
            end
            if (FirstBeyondUpperBounds * [0;1;0]) == 1
                tempArray(ball1).velocity = tempArray(ball1).velocity * ...
                             [1,0,0;0,-BilliardBall.BUMPER_RESTITUTION,0;0,0,1];
            end
        end
    end

        for i = 1: 1: numBalls
            newBallArray(i) = tempArray(i);
        end

end


function BilliardSim( timeStep, ballArray, numBalls, cueForce, ystrike, ...
                        zstrike, simLength, windowWidth, windowHeight )
%BilliardsSim Runs the simulation of billiards
    global FirstCollisionTime;
    %SET TIME VARIABLE TO 0. TO BE USED TO SIMULATE PROGRAM FOR DESIRED
    t = 0;                                                      %LENGTH OF TIME
    tm= 0;
    f = 1;
    %tableWidth AND tableLength FOR A STANDARD 8 FOOT POOL TABLE IN ACCORDANCE
    % WITH THE World Pool-Billiard Association(WPA) IN METERS. playWidth AND
    % playLength DEFINE THE PLAYABLE AREA ( INSIDE OF THE TABLES BUMPERS ) IN
    % ACCORDANCE WITH THE WPA IN METERS.
    tableWidth  = 1.2716;
    tableLength = 2.4416;
    playWidth   = 1.17;
    playLength  = 2.34;

    FirstCollisionTime = timeStep +1;

    ballArray = updateVectors(timeStep, ballArray(1:numBalls), numBalls, cueForce, ...
                                ystrike, zstrike, [.0508,.0508,0], ...
                                [playLength + .0508,playWidth + .0508,0]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DRAWING FIRST FRAME %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    figure('position', [250, 50, windowWidth, windowHeight]);
    title('Aerial View of a Billiards Table');

    %CREATING AXES
    axis([0 tableLength 0 tableWidth]);
    axis('on');
    grid on;
    grid minor;

    xticks([0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8...
            1.9 2 2.1 2.2 2.3 2.4]);
    yticks([0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2]);

    rectangle('Position',[.0508,.0508,playLength,playWidth]);

    while t < simLength

        %TIME COUNTER FOR LOOP ITERATION

        FirstCollisionTime = timeStep +1;
        ballArray = updateVectors(timeStep, ballArray(1:numBalls), numBalls, 0, ...
                                ystrike, zstrike, [.0508,.0508,0], ...
                                [playLength + .0508,playWidth + .0508,0]);

        clf;
        for i = 1: 1: numBalls
            x = ballArray(i).position * [1;0;0];
            y = ballArray(i).position * [0;1;0];
            n = 100;
            for q=1:n
                theta=q*(2*pi/n);
                X(q)=x+BilliardBall.RADIUS*cos(theta);
                Y(q)=y+BilliardBall.RADIUS*sin(theta);
            end
            fill(X,Y,ballArray(i).RGBTriplet);hold on;
        end

        title('Aerial View of a Billiards Table');
        %CREATING AXES
        axis([0 tableLength 0 tableWidth]);
        axis('on');
        grid on;
        grid minor;

        xticks([0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8...
            1.9 2 2.1 2.2 2.3 2.4]);
        yticks([0 .1 .2 .3 .4 .5 .6 .7 .8 .9 1 1.1 1.2]);

        rectangle('Position',[.0508,.0508,playLength,playWidth]);

        pause(.02);
        if(FirstCollisionTime < timeStep)
            t = t + FirstCollisionTime;
            tm = tm + FirstCollisionTime;
        else
            t = t + timeStep;
            tm = tm + timeStep;
        end
        if (tm >= .03333)
            M(f) = getframe;
            f = f + 1;
            tm = 0;
        end

    end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    v = VideoWriter('BilliardsMovie.avi');
    open(v);
    writeVideo(v,M(1:f-1))
    close(v);
end



classdef BilliardBall
    %BilliardBall creates an object that represents a biliard ball
    %   The radius and weight of the ball are in accordance with the
    %   World Pool-Billiard Association(WPA) regulations and are constants since
    %   all balls should have the same weight and radius.
    properties
        position
        velocity
        acceleration
        angularVelocity
        angularAcceleration
        isCueBall
        RGBTriplet
    end

    properties(Constant)
        MASS              = .160
        RADIUS            = .028575
        KINETIC_FRICTION  = .2
        ROLLING_FRICTION  = .01
        BUMPER_RESTITUTION = .7
        MOMENT_INERTIA    = .4 * BilliardBall.MASS * BilliardBall.RADIUS^2
    end

    methods
        %CONSTRUCTOR SETS INITIAL POSITION, VELOCITY, AND ACCELERATION OF BALL
        function obj = BilliardBall(p,v,a, omega, alpha, C, color)
            obj.position            = p;
            obj.velocity            = v;
            obj.acceleration        = a;
            obj.angularVelocity     = omega;
            obj.angularAcceleration = alpha;
            obj.isCueBall           = C;
            obj.RGBTriplet          = color;
        end
    end

end
```