Braden Bagby

Computational Physics

Final Project

Foundations of the Problem

Pool balls exhibit elastic collisions, which can be modeled using what is called hard sphere collision theory. This project demonstrates different pool ball collisions depending on different factors such as the starting point of the cue ball. It shows which starting point theoretically results in the pool break with the most spread, or where the balls exhibit the highest average distance from their starting point. I also simulated these collisions in their simplest form in 2D space and did not consider air drag, rotation, etc.

An elastic collision occurs when two objects collide, and the kinetic energy of the system is preserved. To detect collision between two circles, we check and see if the distance between them is less than the sum of their radii.

To calculate the velocity of the two colliding circles after the collision, we use simple conservation of momentum and kinetic energy physics, along with hard sphere collision theory to take into account the circular collision. I made the mass of every pool ball 1kg, calculating the new velocity is easy, as they just trade velocities because:

Vf1 = [(m1 – m2)* Vi1 + 2m2(vi2)]/(m1 + m2) simplifies to Vf1 = Vi2

And

Vf1 = [2*m1*Vi1 - (m1-m2)*Vi2]/(m1 + m2) simplifies to Vf2 = Vi2.

When m1 and m2 both equal 1

So first we calculate Vf2 and Vf1 in both x and y axis.

Next, we must take into account the sphere aspect of the balls and how that affects their collision.

Hard sphere collision theory says that "The component of the relative velocity, which is parallel to $\vec{d}$, instantaneously changes its sign. The perpendicular component remains unchanged (elasticity). " (http://www.shokhirev.com/nikolai/abc/physics/hard_spheres.html) where d is the distance vector between the two balls.

With this in mind, we update the velocity accordingly and have the correct final velocities for this collision. This idea can then be done across many different balls to simulate an entire pool ball table.

Text from the Program

For a single run consisting of a cue ball colliding with pool balls and then calculating the spread I programmed the following:

Setup

First, I created a matrix with the x and y starting positions of every ball. The cue ball is in index 1 and as you can see has coordinates of -4 and 0. The other balls are spread out like pool balls on a table, but separated slightly to keep a collision from extending instantaneously across multiple balls.

```
ballStartPositions = [
-4 0
0 0
1.2 0.6
1.2 -0.6
2.4 0
2.4 -1.2
2.4 1.2
3.6 0.6
3.6 -0.6
3.6 1.8
3.6 -1.8
4.8 0
4.8 1.2
4.8 -1.2
4.8 2.4
4.8 -2.4


];
```

I also have variables to store the velocities, ball count, mass, radius, and positions throughout.

```
ballCount = length(ballStartPositions);

ballVelocities = [
    5 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    0 0
    ];

ballMass = 1;
ballRadius = 0.5;
```

x and y are matrices that will store the x and y position of every ball for every iteration of time.

```
x = [];
y = [];
```

Loop

Here is the setup while loop for Eulers method that iterates over time.

```
time =0;
dt = 0.01;
runTime=5;
i = 1;
while time < runTime
```

Before detecting collisions, we use Eulers method to move the balls according to their velocity

```
    for u = 1:ballCount
```

```
        position = [x(u,i),y(u,i)];
        velocity = ballVelocities(u,:);



        velocity = ballVelocities(u,:);
        newPos = (position + (velocity * dt));
         x(u,i + 1) = newPos(1);
         y(u,i + 1) = newPos(2);




    end
```

After we have calculated the new position, we need to check for collisions.

The first loop loops through every ball (e.g. Ball1) and gets its position and velocity. The second loop loops through every ball indexed after Ball1. This is so we are not double checking collisions and only compare one object to every other object once.

```
    for u = 1:ballCount
        position = [x(u,i + 1),y(u,i + 1)];
        velocity = ballVelocities(u,:);
                %check collision
         for b = (u + 1):ballCount
```

Inside we get the position of Ball2 and check to see if the distance between Ball1 and Ball2 is the sum of their radii, or in this case twice the ballRadius because every ball has the same radius

```
  position2 = [x(b,i),y(b,i)];
            velocity2 = ballVelocities(b,:);

            %check if it is colliding
            distance = norm(position-position2, 2);

            %we have a collision if
            if distance < (ballRadius * 2)
```

If they are colliding, we get their starting velocities, their direction/distance vector, the normalization of this vector (just for ease of access) and the perpendicular vector to this normalized vector. We will use these vectors in calculating hard sphere collision as talked about above.

```
                %starting velocities
                vel1 = ballVelocities(u,:);
```

```
                vel2 = ballVelocities(b,:);


                d = (position-position2);
                normalized = d/distance;
                perpendicular = [normalized(2),-1 * normalized(1)];
```

Here we perform the calculation for hard sphere collisions

```
 %keep velocity thats perpendicular
                dot1 = dot(vel1,perpendicular * distance);
                perpV1 = dot1 * perpendicular;
                dot2 =  dot(vel2,perpendicular * distance);
                perpV2 = dot2 * perpendicular;

                %switch sign on velocity thats paralell
                dotParalell1 = dot(vel1,d);
                dotParalell2 = dot(vel2,d);
                parV1 = dotParalell1 * normalized;
                parV2 = dotParalell2 * normalized;
```

Then we assign the new velocities to the balls. Note, the formula I found on hard sphere collisions did not fully work. Through trial and error I made it work by instead of switching the sign of the velocity parallel to d, I just switched those across the balls.

```
        ballVelocities(u,:) = perpV1 + (parV2);
                 ballVelocities(b,:) = perpV2 + (parV1);
```

Now, since the collision is done we must make sure the balls are no longer overlapping incase the next iteration would accidentally detect them as colliding again. So we move them apart to be right next to each other.

```
                x(u,i + 1) = collisionPoint(1) + (direction(1) * (ballRadius));
                y(u,i + 1) = collisionPoint(2) + (direction(2) * (ballRadius));
                x(b,i + 1) = collisionPoint(1) + (direction(1) * (ballRadius) * -
1);
                y(b,i + 1) = collisionPoint(2) + (direction(2) * (ballRadius) * -
1);
```

Finally, we increase the index and time and go to the next loop

```

```

```
    i = i + 1;
    time = time + dt;
```

Calculate Spread

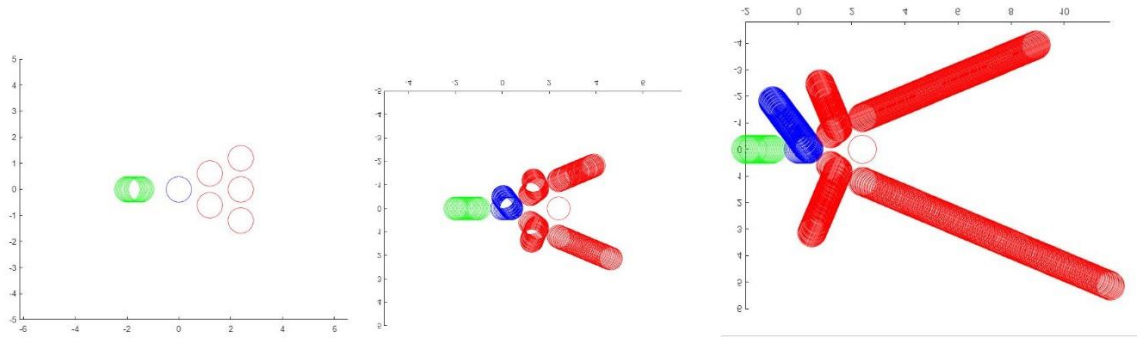
Calculating the average spread is easy. I just loop through all the balls, get their final positions distance from the start position, and average it.

```
totalDistance = 0;
   for u = 1:ballCount

       startPosition = ballStartPositions(u,:);
       currentPositionX = x(u,i);
       currentPositionY = y(u,i);

       distanceFromStart = abs((startPosition(2) - currentPositionY)/(startPositi
on(1) - currentPositionX));
       if isnan(distanceFromStart)
           "didnt move";
       else
             totalDistance = totalDistance + distanceFromStart;
       end

   end

   averageDistance = totalDistance/ballCount
```

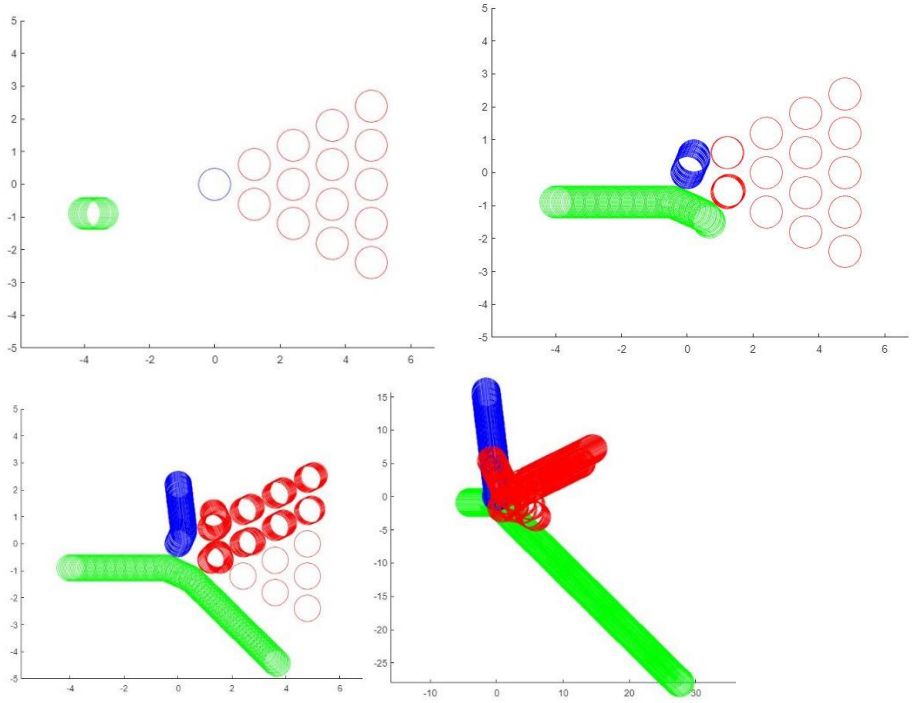
Example Runs

Here are some example runs. The plots show every position of the ball across time. Each circle is a ball. The cue ball is green. The first ball is blue, and every other ball is read. You can see the change of the balls position and the collisions happening as I provide a couple screenshots from different timestamps throughout the run

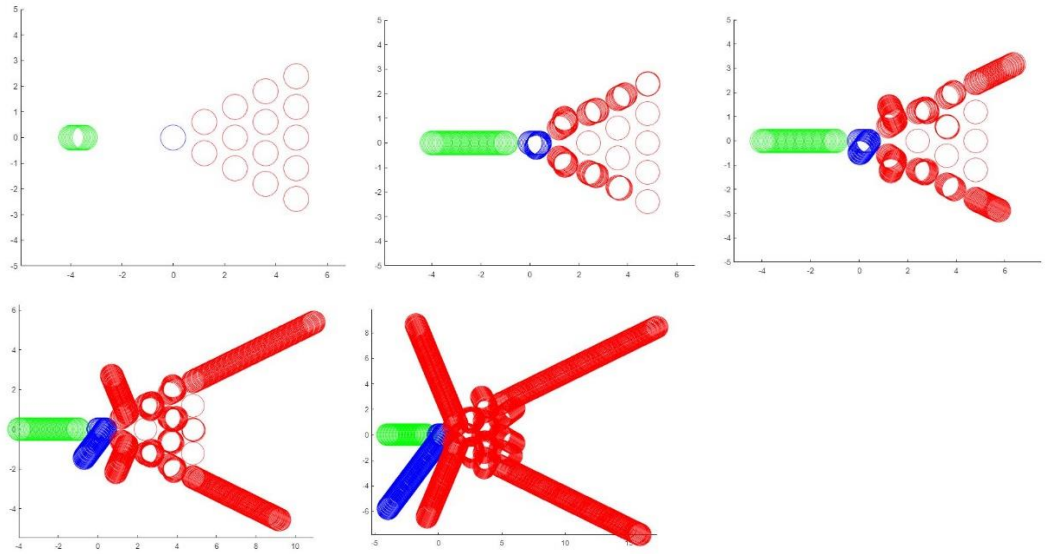
6 pool ball run



Full 15 ball run. Cue ball starting at -0.9



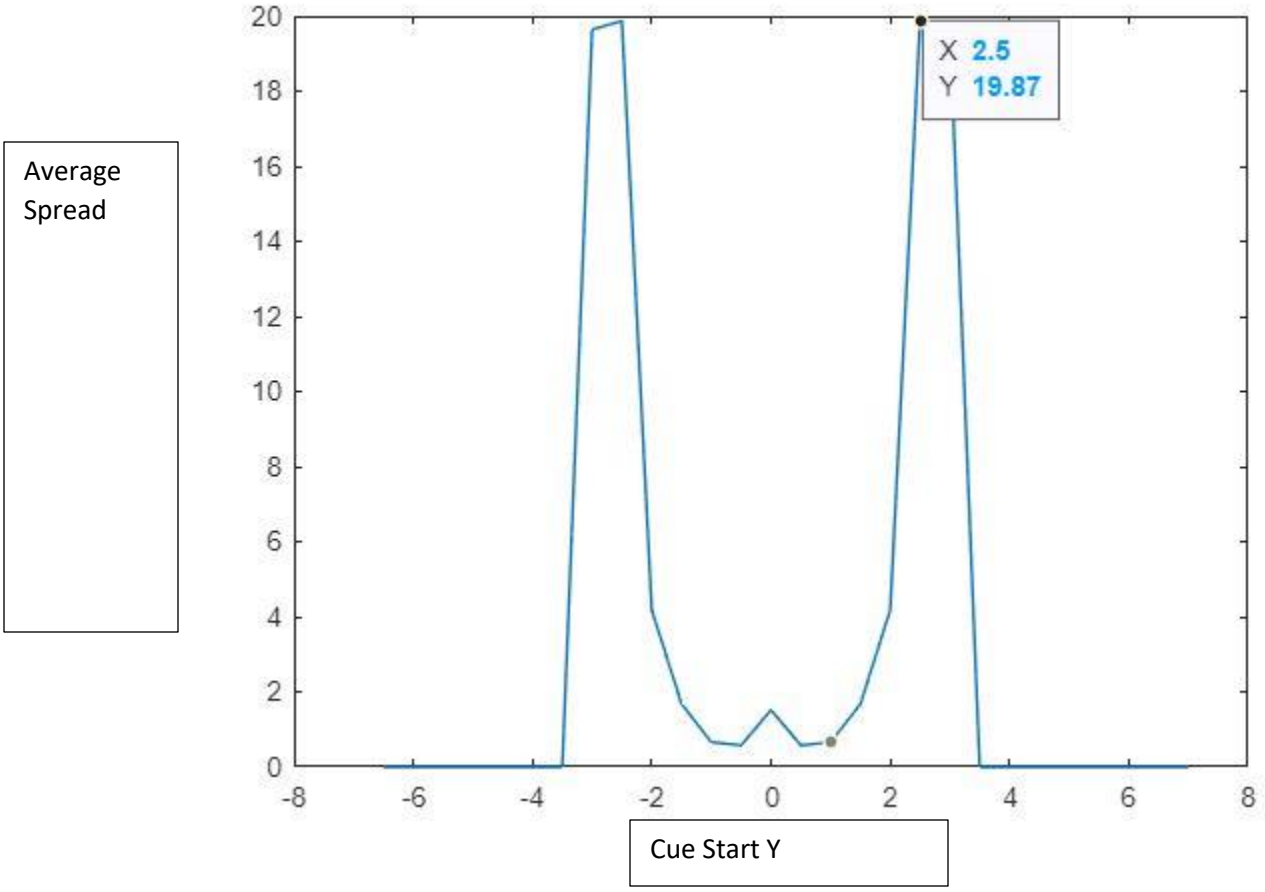
Full 15 ball run. Cue ball starting at (-4, 0)



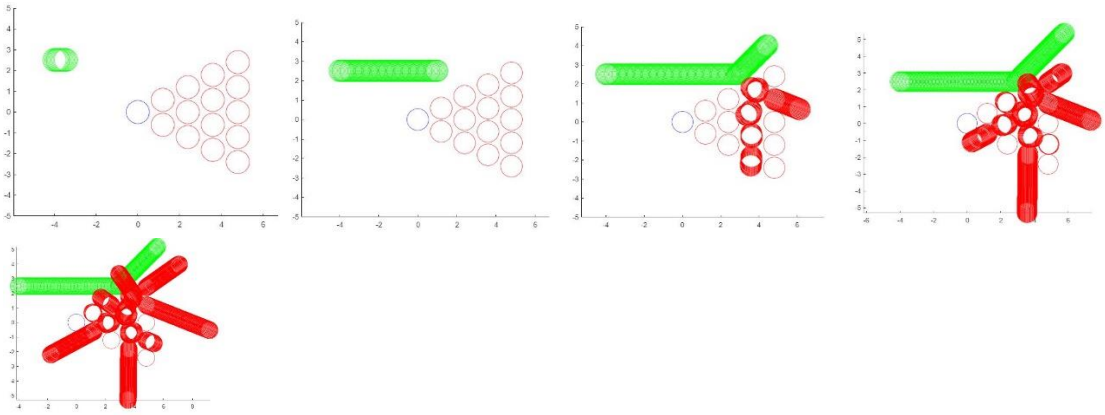
Calculating best starting cue spot

So far we have just discussed how to do one iteration of pool ball collisions where the cue ball started at one position. We can use this to run many different tests. One test is what is the best starting Y point for the cue ball. To do this, we embed our above program in a loop, changing the starting cue y position each time and keeping track of the spread. Finally, we graph CueY Vs Average spread to find out which

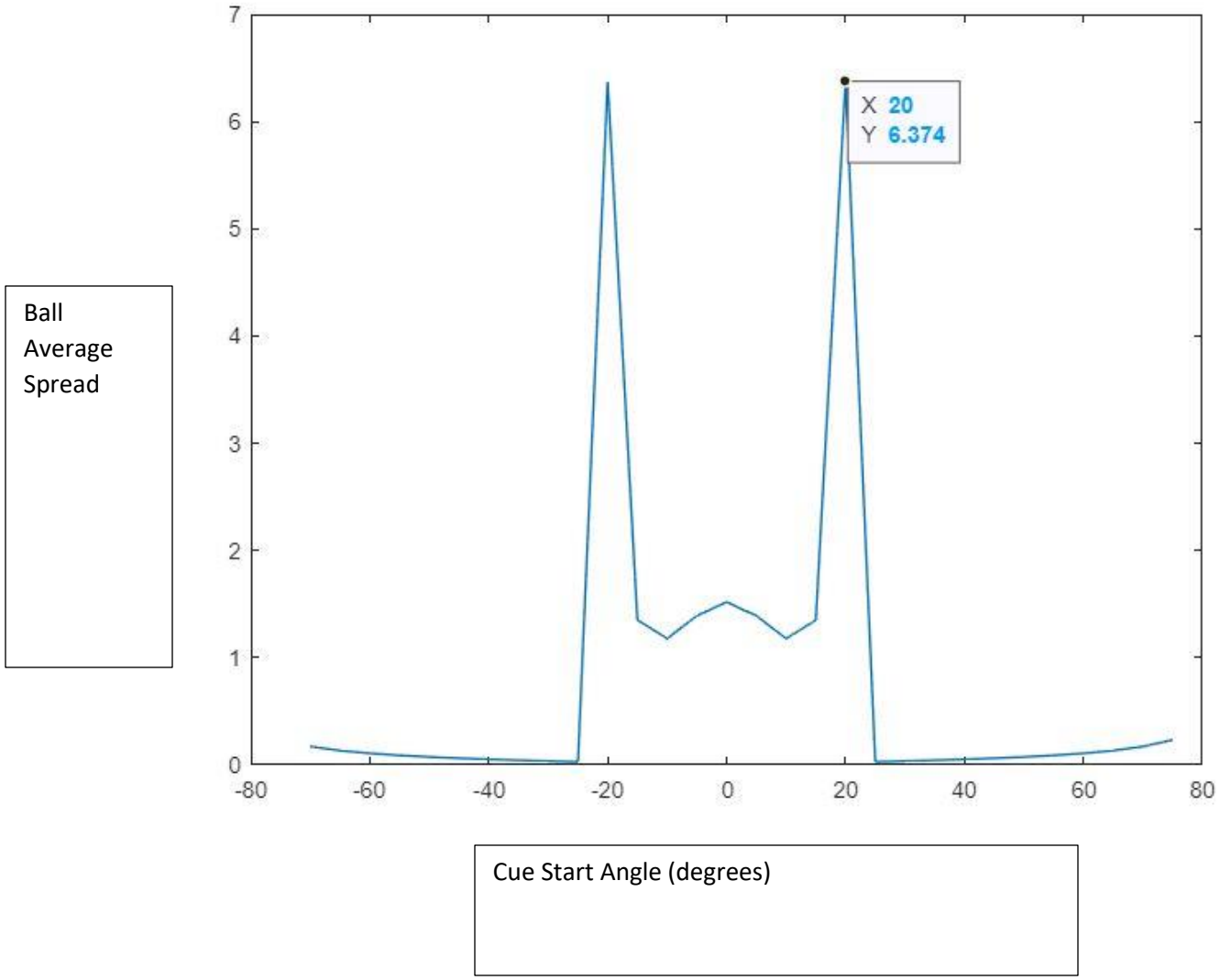
starting Y had the most spread. Here is the result.



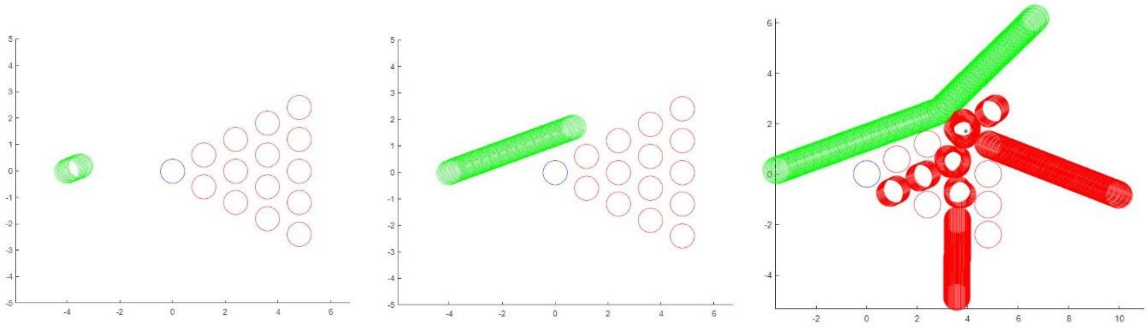
Notice how the graph is symmetric. This would make since as the pool balls also possesses symmetricity across the X axis. This shows that the furthest spread happened when the starting y position of the cue ball was 2.5 and -2.5. Here is the run with the greatest spread.



Ok, but what about starting angle. I changed the program to always start the cue ball at -4, 0, but change its starting velocity angle each time. Here are the results.



The results above indicate that a cue ball at 4 meters from the other pool balls would produce the most average spread aimed at 20 degrees. But, a plot of that specific run may surprise you.



As seen here, it looks like only 2 balls really went far. This must mean that these balls traveled so far that they still drastically increased the 'average' spread. Perhaps, if there were walls these balls would

bounce back and cause more collision and really help produce more visible spread. Overall, this angle of shot has the most impact on the pool balls.

Conclusion

I feel I learned a lot about simulating physics with programming, and I had fun doing it. If I were to do this again or do this project on a professional level, I would want to consider walls and other factors such as ball spin. This program can be used to calculate the possible best circumstances for a break in pool, and I showed that above by doing starting angle and starting y position.