Final Project Report

1. Introduction

This report provides a thorough overview of the algorithm we used to predict the position of the robot for 60 frames (2 seconds) after the end of each test case. In particular, we split the project into three main phases. First, we used the training data to infer a map of the world. Second, we implemented an ensemble learning (EL) algorithm that combines the predictions of Kalman Filter (KF) and k Nearest Neighbor (kNN). Third, we modeled the bouncing dynamics of the robot when it interacts with the world. This allowed us to predict the outgoing angle given an incoming angle. The purpose of the report is to demonstrate that the implementation of our algorithm is reasonably sophisticated for a graduate level CS course. Moreover, we aim to provide compelling justification for our algorithm selection.

2. World

Prior to running our EL algorithm, we mapped the physical world. This was important for three main reasons. First, this allowed us to accurately restrict the codomain of our predictions based on inferred boundary constraints. In this report we will refer to these constraints as walls. Second, this allowed us to localize the robot in our world based on its distance to walls. Third, this allowed us to model the dynamics of collision (e.g. trajectory of robot after hitting a wall). Note that the coordinates of the world is an input to our EL algorithm.

From the initial view of the training video we identified five walls: four walls of the outer rectangle and one wall of the inner circle. In this report we will refer to these objects as the rectangle and circle.

Finding the rectangle is trivial. First, we find the smallest and largest x values in the entire training data and denote them as $xMin$ and $xMax$ respectively. We do the same for y values to find $yMin$ and $yMax$. Second, we use these values to get the coordinates of the top left and bottom right vertices as denoted below. These two points uniquely define the rectangle.

$$TL = (xMin, yMin)$$
$$BR = (xMax, yMax)$$

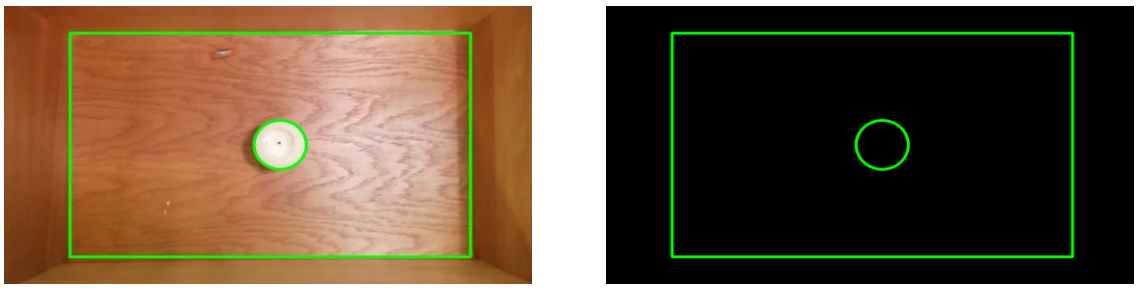

Figure 1: Map of the World (left) with Background Elimination (right)

Finding the circle is less trivial. We wrote an algorithm using histograms to find gaps in the data and subsequently infer the centroid of the inner circle. First, we constructed a 2 dimensional histogram with the training data. See Figure 2 below. Here, we have two parameters: the bin size of x and y. Second, we find the x and y bin intervals with the least density and compute the midpoint of the interval. We treat this midpoint coordinate as the center

of the inner circle. To infer the radius, we analyze the density of the neighboring bins to estimate the distance of the gap. Note that we did not use Computer Vision (CV) algorithms to find the rectangle and circle. With CV algorithms we could have used corner detection and Hough transform. However, we wanted to map the world with only the coordinates provided in the training data. This allowed us to be agnostic of the image properties in the video.

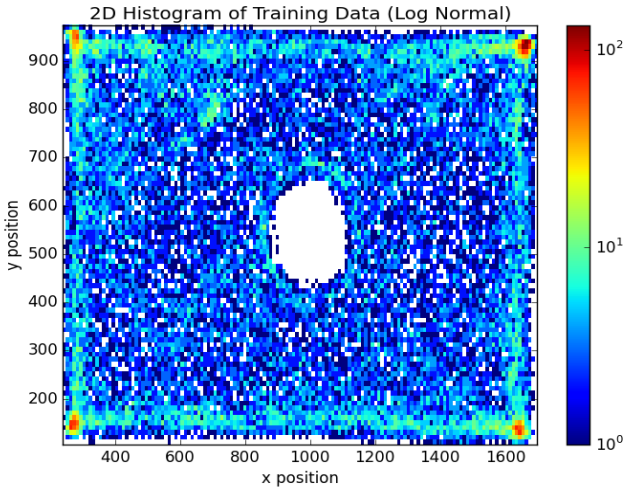

Figure 2: Visualization of Training Data with 2-dimensional Histogram

3. Ensemble Learning (EL) Algorithm

3.1 Kalman Filter

A. Rationale and Explanation

The robot is modeled by a 2D Kalman Filter of predicting $[x, y, \dot{x}, \dot{y}]$. The code is fairly similar to the Kalman Filter discussed in class. The reason that Kalman Filter was suitable for this problem was because four (4) unknown variables were observed, which were (x, y, vx, vy). Since the robot speed was observed to be constant in the videos, we chose not to model acceleration (ax, ay). Instead, we modeled the external forces that affect the robot when it comes in contact with the box border or center candle.

In this predictor, we chose to use a simple and naïve reflect/bounce angle model. Once the robot hits the wall, it reflects just as light reflects from mirror. The (x, y) was reflected to inside the box and (vx, vy) reversed depending on the direction it hits the wall.

B. Parameter Tuning

- dt set to 1 frame/sec
- Velocity measured as number of pixels per second
- External force modeled using a Bu matrix of: $\begin{bmatrix} \frac{dt^2}{2} * u \\ \frac{dt^2}{2} * u \\ dt * u \\ dt * u \end{bmatrix}$ $where\ u = 0.005$

3.2 Markovian Property and kNN

A. Rationale and Explanation

The robot's motion was assumed to behave like a Hidden Markov Model (HMM). In such a model, once a history of the last H steps is known then a prediction can be made on the behavior of the next step. We assumed the robot was followed by HMM of Level 5, which means that once we knew the last five (5) positions of the robot then we could predict its next position.

The training data was not complete but was still comprehensive and we were able to use kNN to match the last robot absolute position to the closest point seen to that point. Level 2 distance was used in order to assess "closeness" of the two patterns. This is justified because it measures the Gaussian noise of one pattern to another. Hence, by minimizing that, we maximize the expectation of the likelihood those two patterns are the same.

There are several ways to assess the "closeness" of two patterns:

- Normalize the points so that the oldest data point is at the origin
- Consider data is invariant by reflection and rotation
- Calculate the absolute difference

To keep our model simple, we chose not to normalize data to the origin and not to consider reflection/rotation. We leave this as a future enhancement to our algorithm.

Once the closest pattern has been identified, we calculate (dx, dy) of the next point in our observed history and apply the difference to calculate our prediction.

B. Parameter Tuning

We tested multiple values of k and multiple values of the length of the history. It turned out that since we combined multiple predictors together, using k = 1 worked well. The length of 5 for the history gave us the most reasonable result. If we had use longer history, we would over fit the data and impacted our runtime negatively. On the flip side, using shorter history would under fit the data.

3.3 Flip Detection

In one test case, we observed a flipped robot that stays at that location for a significant period of time. In order to detect and handle this event, we used standard deviation for both x and y. If both of those values fall below a certain threshold (50) for 3 seconds, then the robot is predicted to be flipped upside down and the predicted position is set to the mean of the last 3 seconds.

3.4 Ensemble Approach and Combining Results

Since each predictor has its shortcoming, we chose to combine them using an ensemble approach. Since we had four (4) potential predictors, we simply took the mean of the three initially. We had to down select to two (2) predictors after assessing our performance with the grading.py file. We also tried to use RANSAC, which took the two methods that agreed with each other the most and then took the mean. Those results were unpredictable due to flip flopping of results between the predictors.

3.5 Time Optimization

In order to keep it under the time constraint, we realized that we don't need to track the entire history of 1800 frames. We simply started tracking the robot using Kalman Filter at frame 1700 and started kNN right after frame 1800. This helped us to keep the time per test case under 6 seconds.

3.6 Future Development

There were a few modifications discussed that were not implemented in our algorithm due to limited time and scope of this project. The algorithm is also constrained in runtime for grading purposes. The following is a list of updates that we believe would improve our results:

a. Add more predictors
b. Incorporate bouncing angles of the box and the center candle into Kalman Filter
c. Add normalizing, reflection and rotation into kNN to see if we get better result
d. Avoid predicting robot at impossible location after taking the average of all predictors
e. Add confident metric, so that we remove weak predictor if the confident level is low
f. Remodel the robot movements even when it's flipped. We can smooth those positions out to see the trend of where the robot is shifting to.

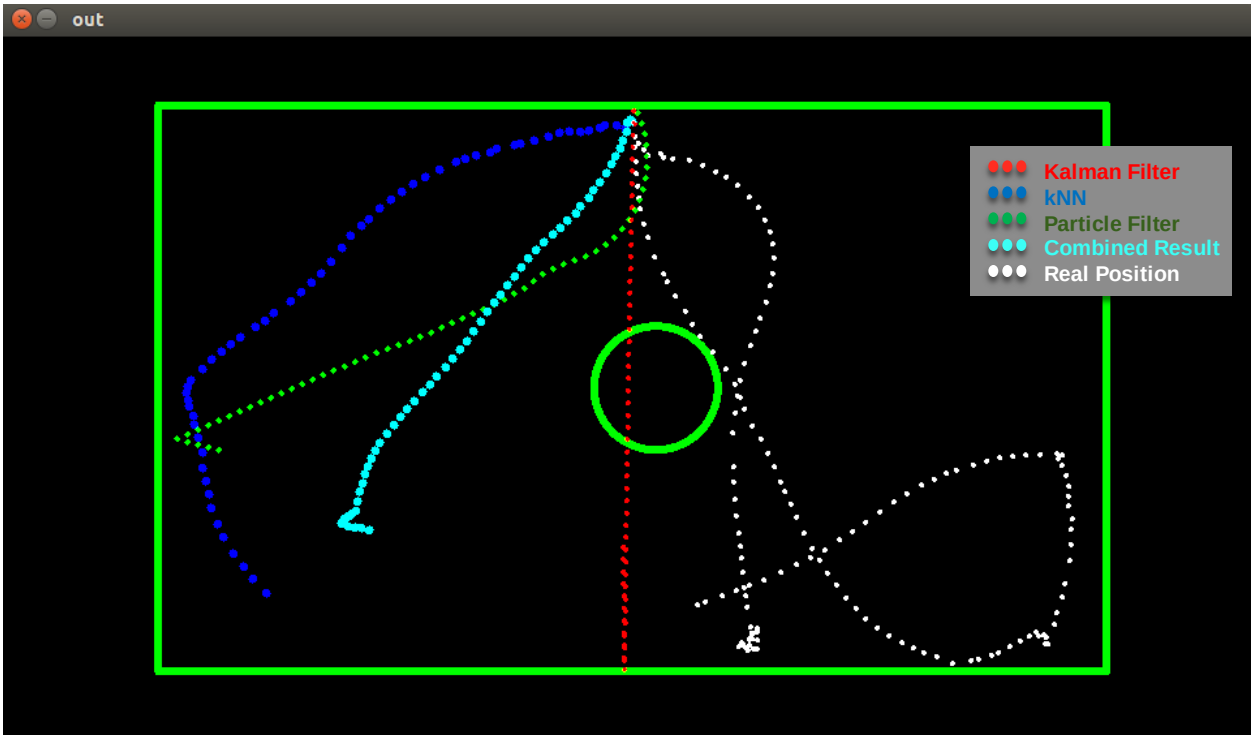

Figure 3: Example EL Output # 1

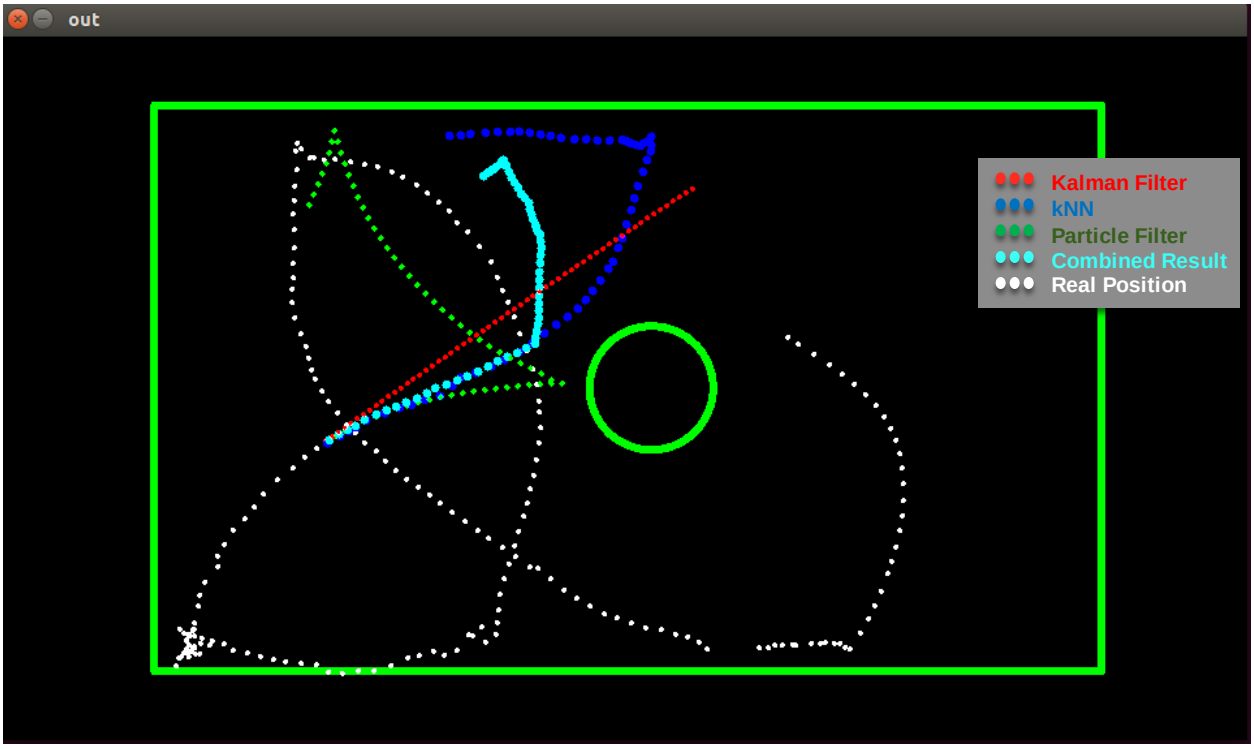

Figure 4: Example EL Output # 2

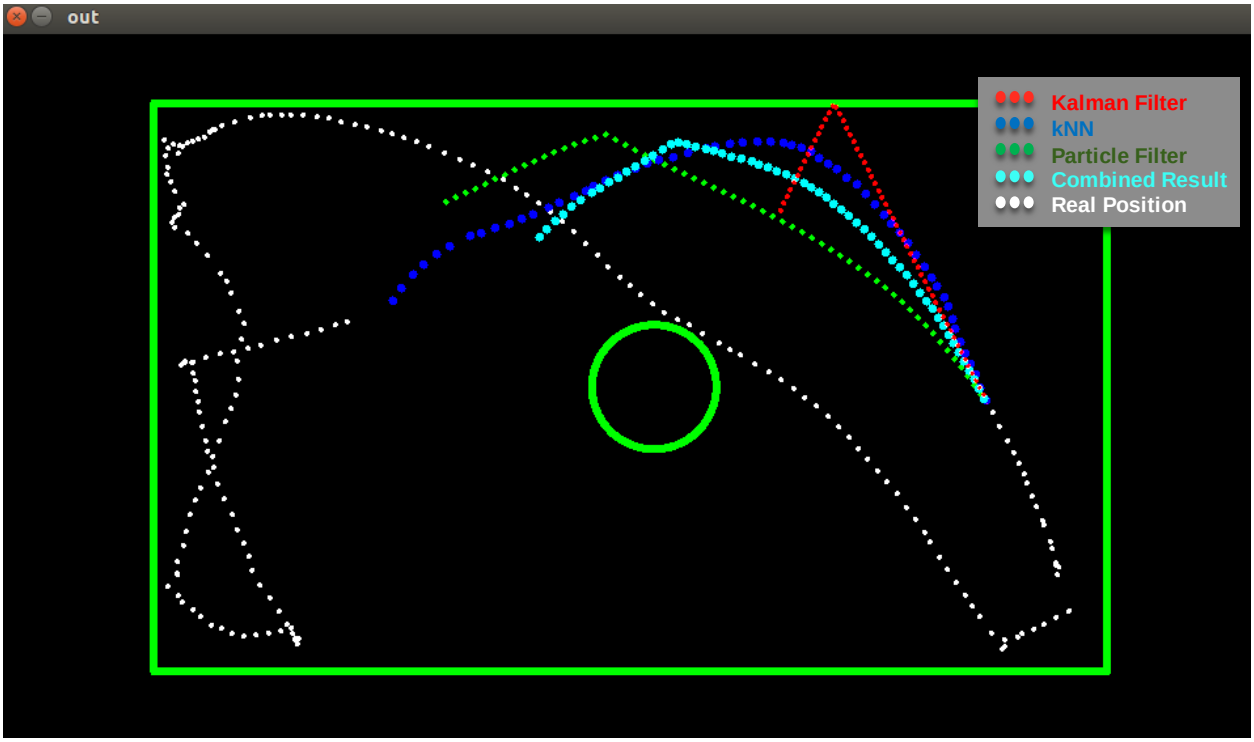

Figure 5: Example EL Output # 3

4. Alternative Approaches

This section contains alternative approaches that were investigated but did not become part of the final EL algorithm.

4.1 Locally Weighted Regression

Early in the planning process, a locally weighted regression (LWR) method was mentioned as a basis of comparison to the algorithms discussed in this course. The reason for a locally weighted regression is due to the non-linear trajectory of the robot requiring newer measurements to be weighted more than older ones when predicting the motion. A Gaussian kernel was used to apply weights to the velocity components (vx, vy) of data points near the last measurement. This allowed us to calculate a velocity vector to then add to the position vector (last measurement) to predict the next position of the robot.

4.2 Particle Filter

Particle Filters (PF) estimate the posterior density of N dimensional state space given an observed prior by using a discrete set of vectors (particles) and their weights to represent probability distributions. Here, we refer to the observed prior as the measurement. The mean of the posterior is used for the prediction. This is a weighted average of the particles in the set.

For our PF we used 2,000 particles and a 3 dimensional state space: speed, heading angle, and turning angle. Speed is the distance traveled for each unit of time. Here, our unit of time is frames. Heading angle is the current direction that the robot is facing. Turning angle is the angle of rotation. See Figure 6 below. Note that we truncate angles between -180 and 180 degrees where positive angles rotate downwards and negative angles rotate upwards. This is the convention used throughout the paper. The high number of particles was necessary to represent the density of a moderately high dimensional state space.

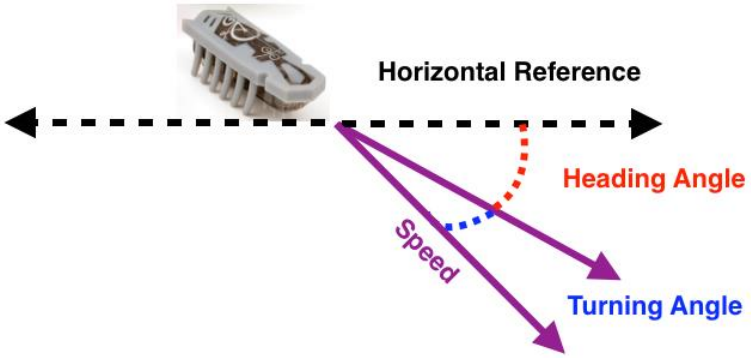

Figure 6: Heading and Turning Angle Convention

For each dimension we specify a sigma parameter as the standard deviation of that variable. These sigma parameters are used in the diffusion step to resample particles from a normal distribution with these standard deviations. For each dimension we also specify a range parameter to restrict the lower and upper bounds of the domain. These range parameters are used to generate the initial set of particles and to completely resample our set of particles in cases where we need to reset our prior belief. We'll discuss these cases in a later section.

Lastly, we specify a separate sigma parameter for the standard deviation of the Euclidean distance computed between the measurement and candidate predictions from each particle. The weight assigned to each particle is computed based on the probability of this distance given a prior normal distribution with mean 0 and this standard deviation. These sigma parameters allow us to perturb our probability distributions and minimally inject stochastic behavior in each time step. We treat this as a form of regularization to prevent overfitting by curbing our belief in the observed measurements. As we'll demonstrate in later sections, the ability of PF to not overcommit to prior observations is a key strength.

In the initialization step, we generate 2,000 particles randomly drawn from a uniform distribution with the min and max values determined by the range parameter. During initialization we also input the coordinates of the world mentioned earlier and create variables for the distance to these world coordinates. A margin parameter is used to define the distance for which the robot touches the world. In the diffusion step, we perturb our set of particles by adding noise from a Gaussian with mean 0 and standard deviation from the sigma parameters. This is done for each dimension.

In the measurement step, we input the new measurement and calculate the weights of each particle based on the probability of distances between the measurement and candidate predictions from each particle. The probability is inferred by a Gaussian with mean 0 and standard deviation from the sigma parameters. Higher absolute distances are assigned a lower probability. Candidate predictions from each particle are calculated by the dynamics model below. Euclidean distances are used. Once the weighting is done, we normalize the vector of weights to sum to 1.

$$x = x + speed * cos(heading + turning)$$
$$y = y + speed * sin(heading + turning)$$

In the sample step, we resample our particles based on the distribution of weights. Particles with larger weights are more likely to be drawn.

In the prediction step, we compute the weighted average of the particles in the set to get a mean speed, heading angle, and turning angle. As mentioned earlier, this is the mean of the posterior. We use these values to predict the next x and y position based on the dynamics model above.

In the localize step, we compute the distance of the robot to the world coordinates: four walls of the outer rectangle and one wall of the inner circle. This allows us to determine in each frame which wall the robot is closest to and whether the robot is touching that wall based on the margin parameter. Here, we used a margin value of 60. If the distance between the robot and the wall is less than 60, than we indicate that the robot is touching the wall. A margin distance of 60 was chosen after we analyzed the relationship between the minimum distance to walls and the turning direction. In Figure 7 below, we can see that the robot usually turns when it is within a distance of 100 to the closet wall. On average, it is around 60. As we'll show later, this information is crucially used in the scatter and bounce steps.

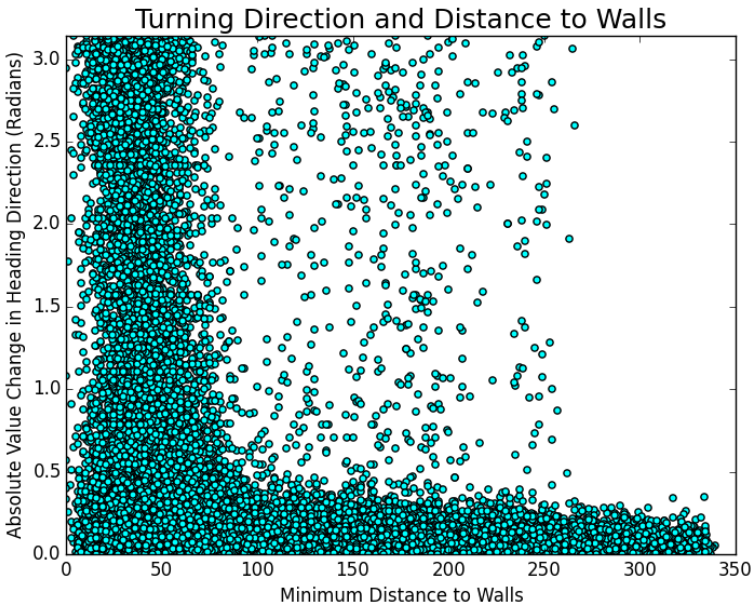

Figure 7: Relationship Between Distance to Wall and Change in Direction

In each iteration, we add two conditions: scatter and bounce. We scatter if a new measurement is available and bounce if a new measurement is not available.

In the scattering step, we reset our belief of the prior whenever the robot hits the wall. Again, the robot hits the wall if the distance to that wall is below the margin parameter. Due to the high noise of the robot, the angular rotation of the robot as it bounces off the wall is best handled by essentially starting over. We resample our particles from a uniform distribution with the range parameters. This is also quite effective in cases where the robot is stuck momentarily in a corner and needs to recover. See Figure 8 below for the density plot of turning direction in the training data. As we can see, the concentrated red regions in the corner represent cases where the robot gets stuck momentarily when it repeatedly turns. Note the red region in the top wall that is not in the corner. This is because the robot flips over at the 9 minute mark of the training video.

In the bounce step, we rely on both physical models and inference from the training data to predict the angular rotation of bounce after the robot hits the wall. When new measurements are not available, we cannot reset our prior belief. To predict 60 frames after the last measurement we need to infer changes in direction. For the physical model, we use the formula below. For the input angles we need the destination and source points for both the incoming and outgoing vectors. The normal vector is calculated using the destination point of the incoming vector (alternatively the source point of the outgoing vector) and the wall. For example, the normal vector to the left wall is 0 degrees. For the circle, we use the center and point of impact to calculate the normal vector. In a later section we will show how to infer the bounce angles from the training data.

$$out\ angle = 2 * normal\ angle - 180 - in\ angle$$

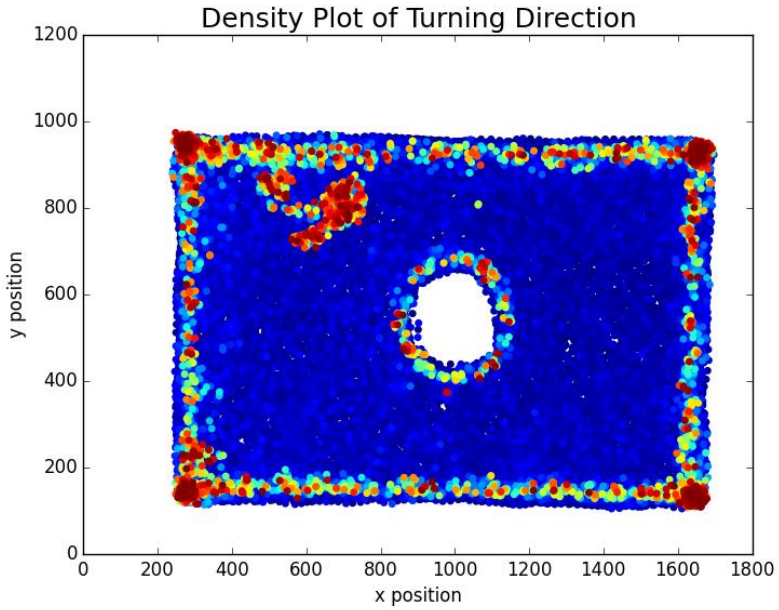

Figure 8: High (red) and Low (blue) Turn Angles

In summary, we decided not to use Particle Filter (PF) due to the time complexity (high run time) and the unpredictable in certain edges cases. Since PF injects a stochastic element to each prediction, it does not perform very well in corners or edges where the robot needs time to recover.

See below examples of the Particle Filter making predictions with the training data. Green dots are the actual measurement. Red dots are the predictions made while we still have new measurements. The blue dots are the predictions made when new measurements are not available. These images do not represent the final accuracy of our program. This is a demo that isolates the PF. Note that the PF is able to predict linearly and nonlinearly.

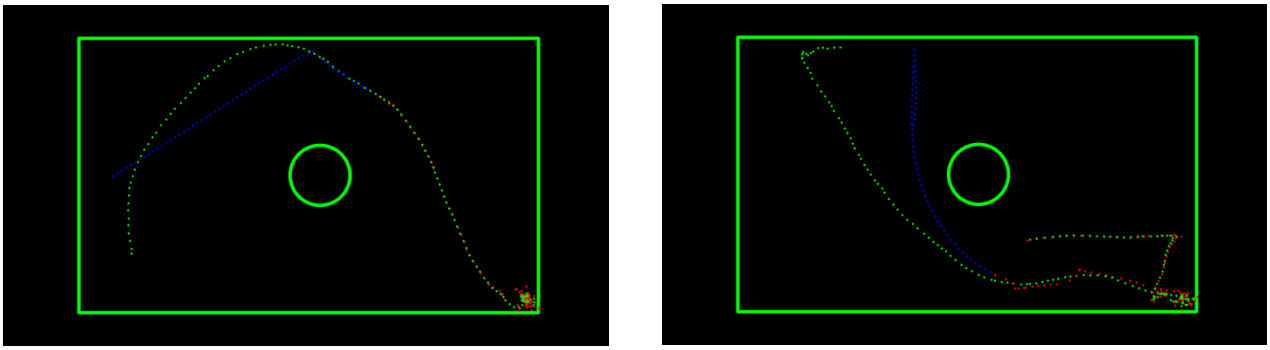

Figure 9: Demonstration of Particle Filter on Training Data

4.3 Bounce Angle

After examining training_data.mp4 video, we noticed that the robot's trajectory after a bounce is random. For this reason, we couldn't use a standard physical model for bouncing where the bounce angle is just the inward angle mirrored. We also noticed that after hitting the wall, the robot halts for couple frames before proceeding due to the rubber front are. Due to this "halt" phenomena, we couldn't rely on velocity and acceleration model. To mitigate the issue, we decided to find the corretion between inbound angle (IB) and bounce off angle (BO). For that, we collected a database of all the angles before and after the robot hits the wall.

First, we assumed that the box represents perfect rectangle with top left vertex at (240, 105) and the bottom right vertex at (1696, 974). Next, we used previously calculated margin to detect when the robot enters the collision area.

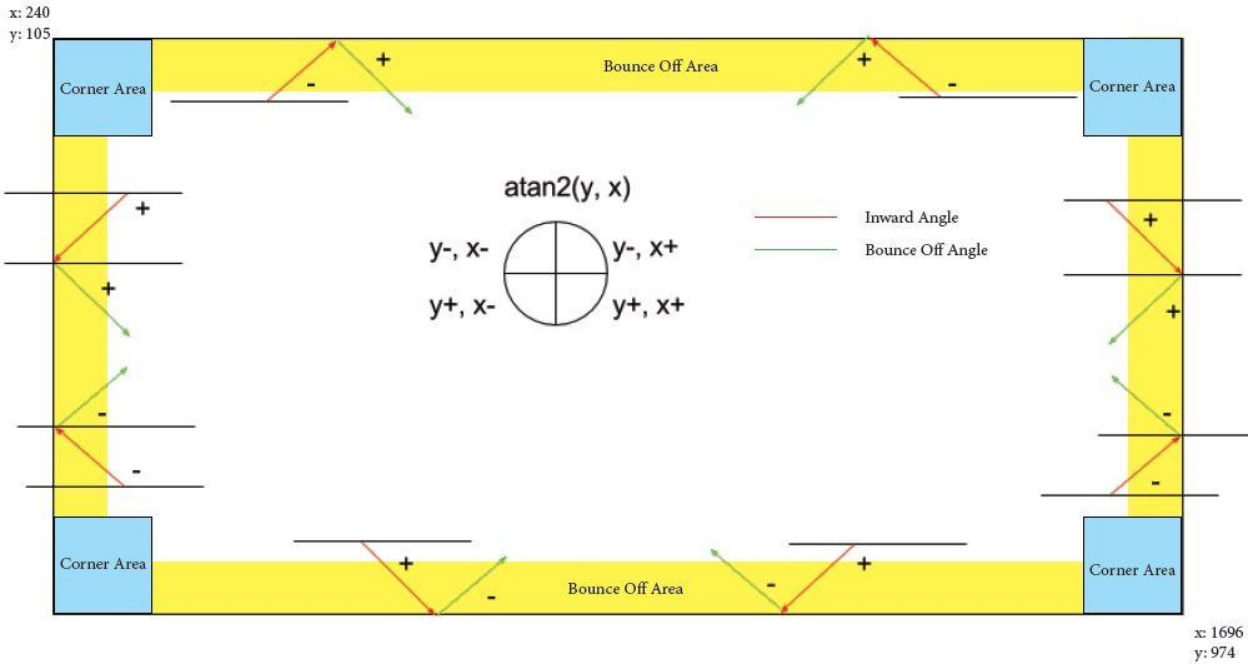

Figure 10: Collision Calculation Diagram

We created an algorithm that records all the position points when the robot enters a margin area ("hit the wall" area). Then when the robot leaves the margin area, we calculate the angles. Angles are calculated as followed:

1. Estimate the point at where robot bounced off the wall based on change of direction
2. Find the robot position one frame before it entered "hit the wall" area (P1)
3. Find the robot position one frame after it left "hit the wall" area (P2)
4. Using these 3 points, calculate the IBand BO angles
5. Add the angles to the database

After collecting all the IB and BO angles for each wall, we sorted them out into bins based on IB angles (-180° to +180°, 5° increment).

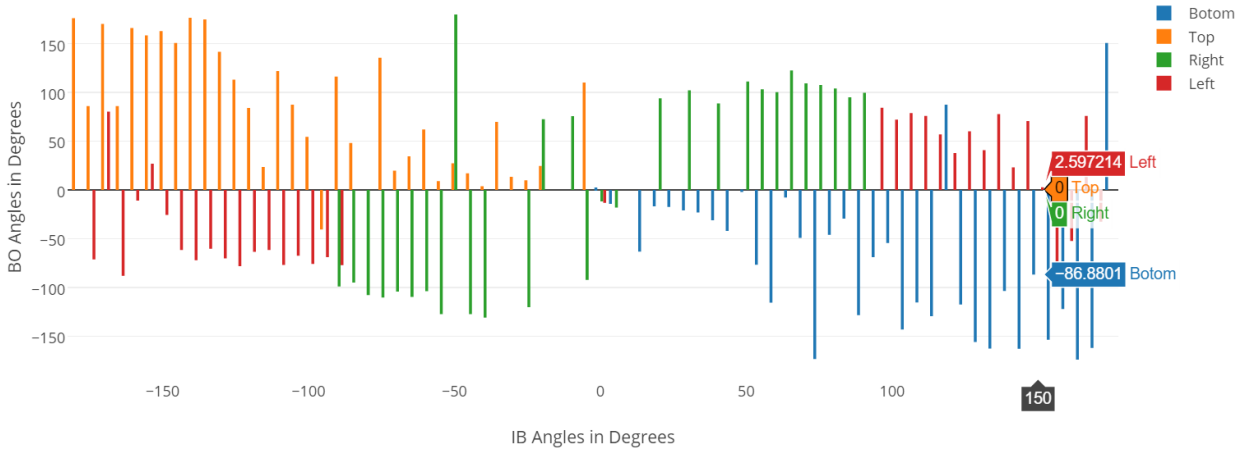

Figure 11: Inward Angles vs. Bounce-off Angles (Not Adjusted)

There where a few errors because of the robot's random movement. However, we found correlation between the IB/BO angles and were able to fit the data for our predict unknown values using 2nd order polynomial.

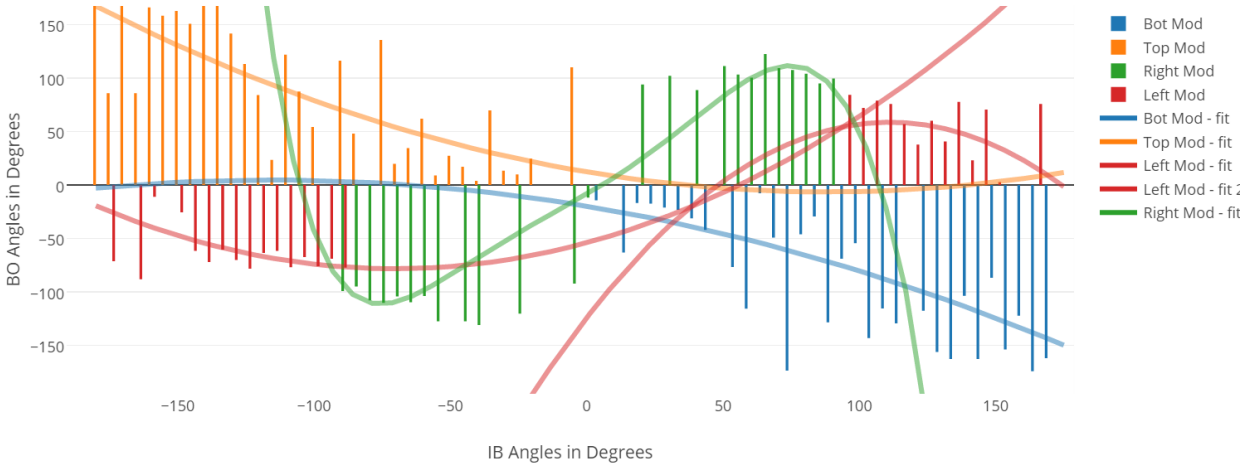

Figure 12: Inward Angles vs. Bounce-off Angles (With Fitting Lines)

Next, using these fitting lines and values of the neighbors, we predicted the angles for the unknown IB angles as shown in Figure 13.

For the corners, we couldn't find any correlation between IB and BO angles. However, we noticed that the angle at which the robot enters the corner area affects the duration of its presence there. Upon this discovery, we decided to create database of the robot's entering angle and the time spent there (TS). We created similar algorithm as before, one that records the points when the robot enters the corner area and performs the calculation when it leaves the area. The algorithm outputs time spent in the corner, based on the number of points, and the angle which the robot entered the area (using first 3 points). Next, we sorted out the data into bins of 5° angles and plotted TS as function of entering angle.

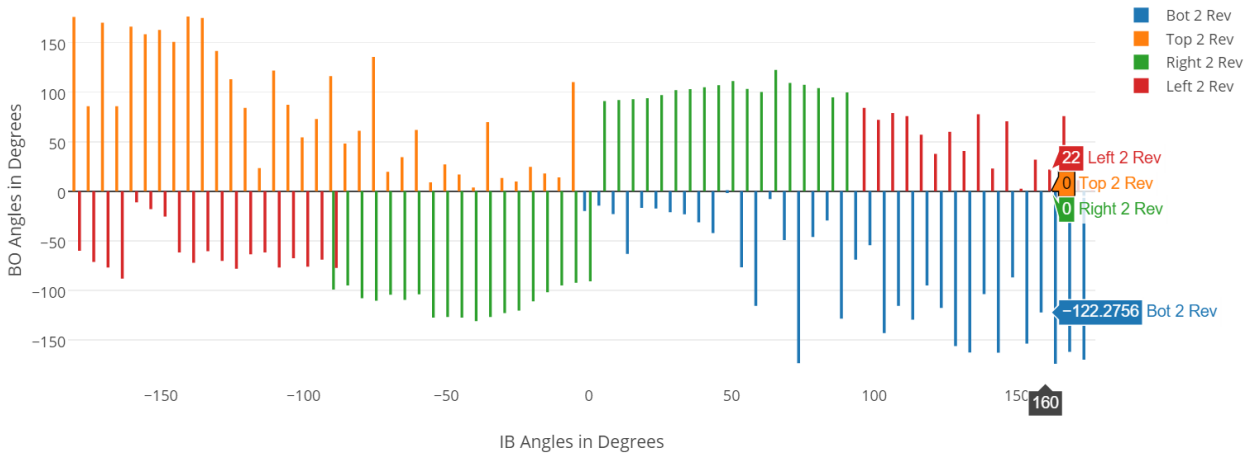

Figure 13: Inward Angles vs. Bounce-off Angles (Adjusted)

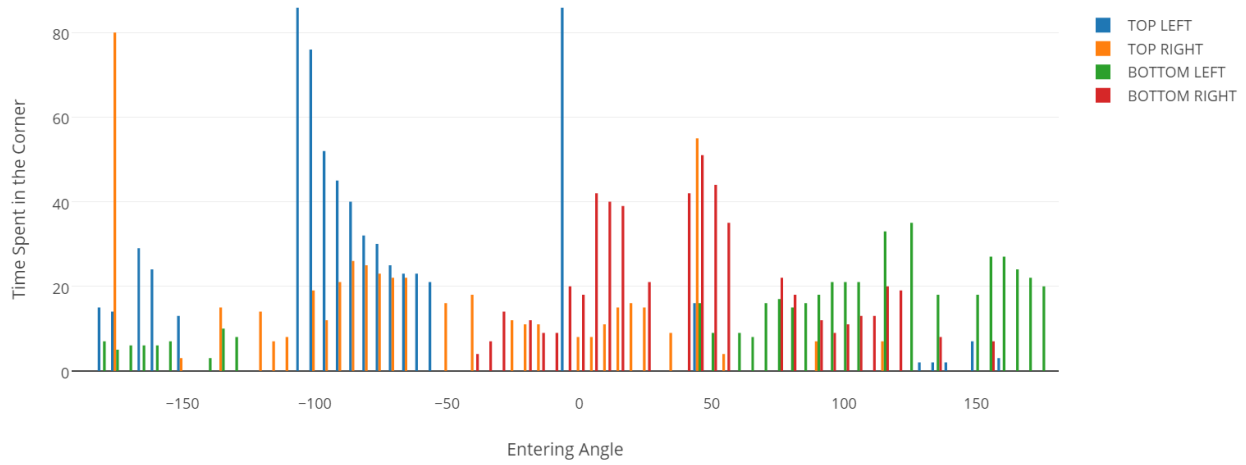

Figure 14: Entering Angle vs. Time Spent in the Corner

We expected that at 45 degrees entering angle, the robot will spend the most time in the corner area, bouncing from one corner wall to another. We indeed noticed such correlation at the Bottom-Let and Bottom-Right corners. However, with the top corners, the TS vales were too random to make the same conclusion.

5. Testing

To test our algorithm, we used the provided Virtual Machine (VM) to make sure it ran successfully and within the allotted runtime. We first tested the Particle Filter using grading.py and it had decent performance with an output error value of 2562 but took more than 3 minutes to execute. When the number of positions for tracking was reduced from 1800 to 100, PF executed in less than a minute but the error increased to 5564. The Kalman Filter was tested next and it produced an error of 1991 and executed in less than a minute. We continued with an

ensemble learning algorithm incorporating PF, KF and kNN. This test performed even better with an error of 1758 and a runtime of 57 seconds. Finally, we decided to run our ensemble learning algorithm without the Particle Filter. The updated EL algorithm scored an error of 1554 with a runtime of 44 seconds.

We were pleased with the results of the EL algorithm with KF and kNN. However, there were some concerns regarding the overfitting of data. To test for overfitting, we created sets of inputs and actual data using random cut sections from training data. Using this updated test case, the algorithm performed well and executed in less than 30 seconds with a score of 1096. We were also concerned about overfitting the data by training our algorithm on the same data we tested it on, so we removed the cut portion we cut of test points from the training data. The algorithm was executed yet again with this change and scored an error of 1435 with a 26 seconds runtime. From these results, it was decided to use the ensemble algorithm with Kalman Filter and k Nearest Neighbor as our solution to the given challenge.

6. Conclusion

In conclusion, an ensemble learning algorithm was developed as a solution to predicting the robot's motion for 60 frames (2 seconds) and for the given test cases. The EL algorithm combined the predictions of a Kalman Filter and k Nearest Neighbor to produce the best results out of four (4) potential candidates: Kalman Filter, k Nearest Neighbor, Particle Filter, Locally Weighted Regression. To measure our performance, the EL algorithm was tested using the provided grading.py (Python file) for various combinations. In the Virtual Machine (VM) environment, the final EL algorithm had an error 1554 and a runtime of 44 seconds using grading.py on the original inputs.