CS 8803: Final Project

Craig Abernethy, Glenn Abernethy, Swapnil Ralhan
Summer, 2015

Abstract

We present below our strategy, algorithm and implementation details for the CS 8803: AI for Robotics Final Project. With the goal of predicting the behavior of the robot, we use the extended kalman filter to accurately model its behavior. Then we further apply the model of the world generated from the training data to predict collisions and demonstrate the different parameters we tuned to get optimal results.

Algorithm Overview

Extended Kalman Filter

In order to find a reasonable strategy to predict the robot's movement, we analyzed the input data and videos very closely. Since the input data was very minimal (only containing x,y coordinates for each frame of camera data) and the robot's movements were very sporadic and unpredictable, we decided to implement an extended kalman filter. The benefits to utilizing an extended kalman filter helped us develop a transition function from one point to the next to accurately predict the most likely position in the next time step while considering other variables such as measurement noise and external motion. What we found was that the kalman filter could accurately predict the robot's position except on the extreme cases where the robot reflects off a wall/object unpredictably or turns randomly but it was able to get back on track with minimal steps to continue predicting accurately after about 2-3 steps. This meant that the

kalman filter was going to be very useful in predicting the robot's position when we ran out of input and had to solely rely on the previous state to predict a full 2 seconds worth of predictions.

Our extended kalman filter is comprised of several matrices used in two main steps to both predict and then update our measurements for our current state X and our covariance matrix P. While we took into consideration measurement noise and external motion (matrices R and U respectively), they didn't appear to have much of impact on the kinematics of the robot and thus are only included for robustness and testing. While the code is included in our kalman_filter.py, the general algorithm is as follows:

Predict

$$X = (F * X) + U$$ F is our transition function

$$P = F\ P\ F^T$$

Measurement Update

$$Y = Z - (H * X)$$ Z is our measurement and Y is our error

$$S = H\ P H^T + R$$ S is our system uncertainty

$$K = P H^T S^{-1}$$ K is our Kalman gain

$$X = X + (K * Y)$$

$$P = (I - (K * H)) * P$$

When this algorithm is applied to the measurements, it produces accurate predictions as shown here where green are the robot measurements and red are our predictions:

Figure 1

Closer inspection of the graph depict that along straight paths and predictable curves, the kalman filter does quite well in staying on target with the robot. When the robot collides or has an unpredictable turn, the kalman filter can be seen to deviate since it's prediction was wrong but immediately gets back on path and continues to predict with good precision.

Our main method for transitioning from our measurement to our prediction was utilized in our matrix F (state transition function). Since at small discrete steps in time the robot more or less is going in a straight line and the fact that we have accurate data to 30 frames per second, we were able to utilize geometry to predict $\hat{x} = x + distance * cos\Theta$ and $\hat{y} = y + distance * sin\Theta$ given the distance between our previous measurements, the angle between our previous measurements, and turning angle as calculated from our previous two angles to predict a straight line measurement. Given the straight line predictions and the actual measurement data, our predictions contained roughly 0-2% error on average with a few outliers indicating unpredictable collisions, bouncing off walls, and sharp turns yielding realistic results. Combined with collision detection, the hard to predict measurements had reasonable geometric trajectory which allowed predictions to have minimal error when our uncertainty is high.

Collision Detection and Handling

The 20 minutes of training data was analyzed thoroughly in order to build an accurate collision detection model. The data shows within the 20 minutes of video that the robot reached values of X between 240 and 1696 and Y between 105 and 974. We initially used values of (240,105) and (1696,974) as our boundaries. Further analysis proved the robot moved below 248 and over 1690 on the X-axis less than 10 times each, and below 111 and above 967 on the Y-axis less than 15 times each. The movement of the robot is such that it was getting closest to the walls when it was moving parallel to that wall. This made us tighten the boundaries for the collision model - there's little reason to worry about areas it can only reach by moving parallel to the boundary. The more severe angle the robot heading towards a boundary kept the robot from getting as close as it might due to the oblong shape of the robot. In fact, the shape of the robot kept the robot from reaching the extreme boundaries of the map (see Figure 2).

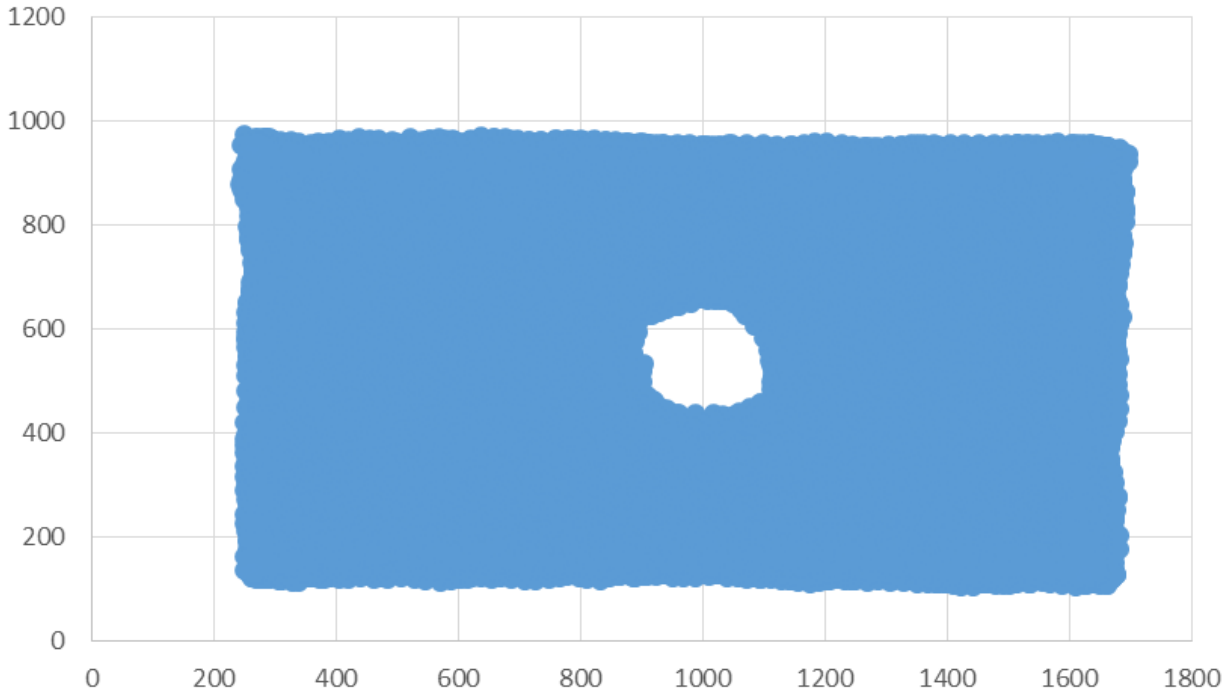


Figure 2

We also detected a circular region with a range on the X-axis from 902 to 1097 and 439 to 645 on the Y-axis. We gathered from the approximate 200 unit diameter on both axes that we should place a collidable region with radius 100 at the coordinate (1000, 542).

The collision model used the aforementioned boundaries to predict the reflection of the robot from impact. Any prediction that moved the robot outside the borders on the X-axis subtracted the current heading from π, and a prediction outside the borders on the Y-axis was negated. The center portion of the map was slightly more difficult to model. When the predicted movement was within 100 units of our detected center, we reflected the movement based on the robot heading and tangent to the circle at the moment of impact. See below :

```
dist = distance_between((x,y), OBSTACLE_CENTER)
if dist <= 100:
    s = sin(heading)
    c = cos(heading)
    # angle of the tangent
    ca = atan2(y-OBSTACLE_CENTER[1], x-OBSTACLE_CENTER[0])
    # reversed collision angle of the robot
    rca = atan2(-s, -c)
    # reflection angle of the robot
    heading = rca + (ca - rca) * 2
```

We detect the angle of the tangential line on the circle at the moment of impact as well as the collision angle of the robot. The arctangent of those returns the angle of the robot heading and the angle of incidence on the circle. The new heading is the collision angle plus twice the difference between the tangent's angle and the heading. This ensures a small deflection will occur when the robot glances off the side of the center and a severe collision will turn the robot around at a severe angle. Figure 3 demonstrates the model in test.

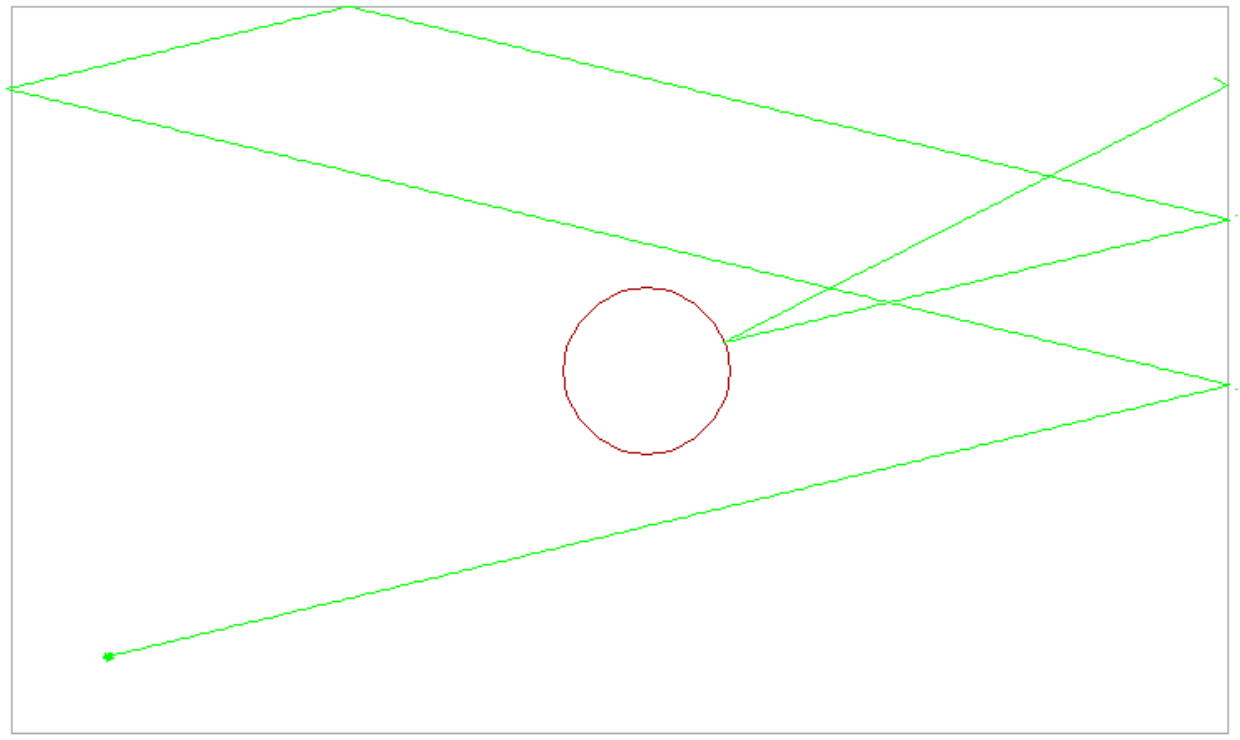


Figure 3

In practice, the billiard-ball type of collisions were accurate for many test cases, however was problematic in some scenarios. In a handful of cases, the robot hit a boundary at a perpendicular angle and did not turn but bounced backwards and eventually turned parallel to the boundary. This would give us a trajectory nearly 90° from our projection which would be the opposite direction of the preceding vector. In cases that the robot would hit near the corner of the contained box, the oblong robot would pivot to one side and hit the wall and return to the corner. Predicting when and how the robot could escape the corner when stuck was an exercise that we eliminated from the model.

Performance

We tested our code using the last 2 seconds of the provided test file. Our finalproject.py, if provided with the --test=True flag, will exclude the last 60 values from the test file. The code will output its prediction for the last 2 seconds, and we used the grading.py script to compare the error for these last 2 seconds' prediction with the actual test video.

Errors

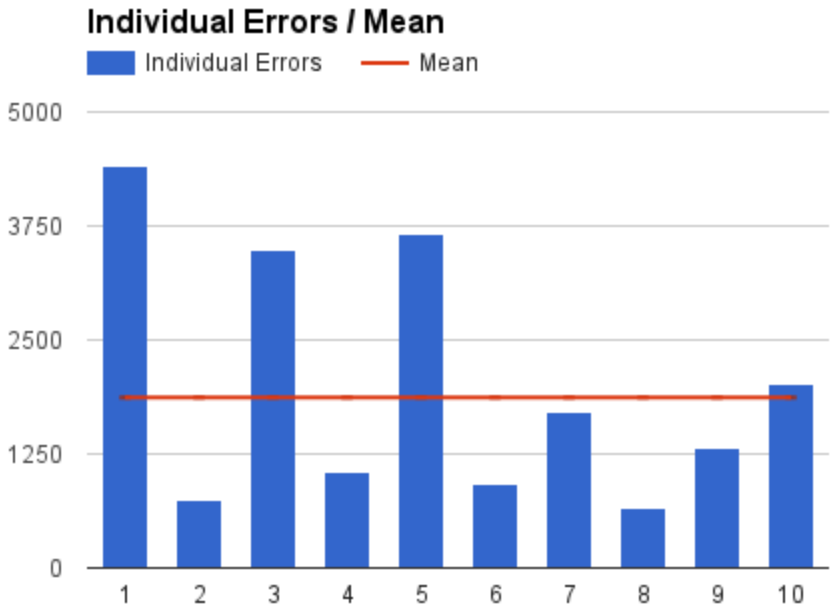


Figure 4

Figure 4 shows the performance for each test. We get a mean error of 1867.96, though as we see from the above results, we do much better in most of the tests, and worse in only 3 tests. We talk more about the parameter tuning we did to achieve best results with the given algorithm in the next section.

Time

The algorithm implementation is extremely fast, taking approximately 1 second per test. Doing a simple evaluation, we can see that it took around 10 seconds to run the code. Given this performance, we did not work on optimizing this further.

```
$ time python grading.py finalproject
1867.95705522

real  0m10.428s
user  0m9.740s
sys   0m0.332s
```

Parameter Tuning

We tweaked a number of parameters used by our algorithm, to achieve the optimal performance.

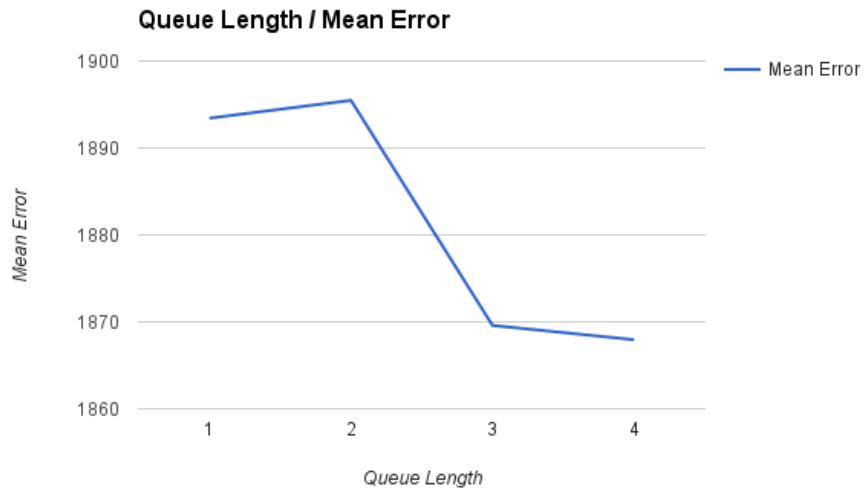
Historical Queue Size

The first parameter we tuned was the number of elements used for historical evaluation. This was closely evaluated in tandem with the next parameter(minimum number of elements present in the queue before we use the historical data as opposed to simply the kalman filter prediction). Here, we present the results of different values of the historical queue size(using the best value of the 'number of elements' parameter)

From the above analysis, we found the minimum error at a Queue Length of 5.

Number of Elements in the Queue

Next we used a second parameter to decide when to use the historical data. In our experiments we realized that we need a minimum number of historical points at which point the number of elements in the queue becomes useful. We did some experiments at each queue size above to generate the optimum length. Below we show the results for queue length 5:



The optimum Queue Length before we use the queue is 4, for a total queue length of 5.

Measurement Noise

We tried various values of measurement noise, and found that the relative effect of the measurement noise was negligible on the prediction. We settled on a value of 0.05, and did not spend too much time optimizing this as it did not affect the results.

Short Historical Data vs Long Historical Data

When analyzing the historical data queue and finding our optimum queue length of 5, it became apparent that while that combined vector overall was a good indication of where the robot should be going and how fast, it unfortunately wasn't always that good which became apparent in when the robot was trapped on it's side and would bounce back and forth making an illusion that the vector was an accurate depiction of where the robot would go. Ideally, we would predict the robot would stay roughly in the same space so we started to think of ideas on how to utilize a long history of data such that it would not impede on our results. We decided to try using a short term history of data and a long term history of data and creating two vectors that would represent where the robot was going in the short term and where the robot was going in the longer term (since the last collision). The average of these two vectors were then used and we were able to accurately predict where the robot would go in situations like the robot being on it's side. Unfortunately, overall it didn't end up improving our predictions so we didn't use it in our final build but we really liked the idea. We also ended up trying to use the short term data as a good estimate of the direction the robot would go and the long term data as a good estimation of how fast it was going to make a better vector but we still couldn't improve our results so we didn't end up using this in the final build.

Historical Weights

An idea that we explored was weighing newer measurements more heavily than older ones. We tried 2 weighing methods:

1. **Exponentially increasing -** Starting with the oldest historical point indexed as 0, we gave a weight of 2^i, to historical point index i. With this method, we got a mean error of 2270.14.
2. **Constantly increasing -** Starting with the oldest historical point indexed as 0, we gave a weight of i, to historical point index i. With this method, we got a mean error of 2157.87.

Finally we ended up with no weighting, and got the optimal result of 1867.96.

Alternatives Considered

Kalman Filter

We attempted to smooth out the unpredictable curves by implementing an 8 state kalman filter by introducing 3 new variables: rotationalVelocity, accelerationX, accelerationY. With rotational velocity, we were hoping to capture how fast we were turning on a curve to better fit the curve. Unfortunately due to the nature of the robot's randomness and the sometimes overestimation or underestimation of our prediction, we would have a more jagged curve that would be less precise than if we made our math more simple. While the robot never truly goes straight for long periods of time, it was a better estimate than predicting it to curve one way and it truly ends up curving the opposite way. For the acceleration, we were hoping that we could detect when the robot was speeding up or slowing down to allow us to predict where we were going more accurately. With 60 frames of data and variable speeds, the robot potentially could end up much further than we anticipate simply due to the fact that it was acceleration at the point the video was cut. Our initial testing demonstrated that this ended up being more complicated than our simpler model and was not as accurate in results so we decided to not focus on the acceleration or rotational velocity and to go back to our 5 state Kalman Filter which implemented x,y,angle, turning angle, and the distance.

Particle Filter

We briefly considered how we could make use of a particle filter for our predictions but since our Kalman Filter was utilizing a 5 state variable, we weren't concerned with it being too complex or computationally expensive to predict the next motion. Given our experience with kalman filters on the runaway robot problem, the kalman filter became a natural extension to this problem.

Conclusion

The project end result evolved significantly in the weeks leading up to submittal. The team's primary goal was to develop a Kalman Filter that would smooth the robot's movement and continue the predicted path for the next 60 frames. At some point a collision would occur (a few cases the collision was imminent), thus a model for handling collisions was developed and integrated. There was quite a bit of discussion about acceleration and deceleration of the robot and we concluded that the electric motor's acceleration was often mitigated by randomness in turning and obstacle collisions.

We had implemented rotational velocity into the projection as well. While we believed the robot should continue a turn during the prediction phase, in testing we found the prediction to be less reliable than maintaining the course. The unpredictability of the robot movement within the data was such that a straight course was unlikely though a left hand turn staying left was less likely. In our testing and tuning, keeping rotational velocity in our prediction phase resulted in a greater error rate and was subsequently removed.

While combining the kalman filter and collision detection module gave us the greatest results, we weren't quite pleased with our predictions and we became experimenting with a lot of different ideas to better handle realistic motion. Ultimately, we were able to accompany our current prediction with a historical vector that allowed us to modify where our robot was most likely facing and how fast it was going over an average of 5 points instead of the last point. This ended up producing much greater precision in our results and subsequent testing and data analysis show that it handles most situations in a realistic manner.

Overall this project has been a great learning experience for us and it allowed us to work well in hypothesizing how best to predict real world movements when sensors fail and measurements are unreliable. The techniques we learned in lecture allowed us to practice them in a realistic example which demonstrated how powerful the kalman filter can is and how it could be used.