CS8803
Artificial Intelligence for Robotics

Final Project Report

Fall 2015

X.D Sheldon Gu	xgu60@gatech.edu
Lei Wang	leiwang@gatech.edu
Zhihao Zou	zzou@gatech.edu
Yanfang Li	yli818@gatech.edu

Table of Contents

1. Project Overview

In this project, we are given ten 60-second video clips of a robot (a hexbug) moving in a wooden box. Each video is recorded at 30 frames per second. We are asked to predict the position of the hexbug for the following 60 frames (2 seconds) after the end of each video.

In addition to each of the 10 test videos, we are provided with a text file that contains extracted centroid coordinates for that video that we should use as input to our algorithm. All given data are real-world data. We are also provided with a ~20 minute "training" video and corresponding centroid data. "Training" and "testing" videos and data will help us to make our program more accurate. These videos were shot using the same conditions as the ten 60-second video clips.

Our predictions will be compared to the actual video data after the end of the video. The predictions are expected to be as accurate as possible.

To predict the position of the robot for the last 2 seconds after the end of each video, we first applied Kalman filter to solve this problem. We found that Kalman filter worked well when the huxbug was moving linearly or with slight curves. However, when the little bug hit the wall or the obstacle in the center, its motions became unpredictable by Kalman filter.

Therefore, we added more functionalities to our initial algorithm in order to better predict the bug's movement when it hits something and change its moving direction. Basically what we did was to manually change the bug's heading when it is predicted by Kalman filter that the bug's next position will be outside of boundary. The change of heading follows a modified rule of reflection.

Later, we ran our algorithms with all centroid data of 60-second video clips. We found that prediction error cumulated with number of steps (assume one frame is one step) that the hexbug made increased. So we evaluated how good predictions would be if we only use last several steps to put into our algorithm. Our final decision was using the last 30 steps to predict the positions of the bug in 2 second after the end of each video.

1. Kalman Filter Overview

Kalman filtering is a linear quadratic estimation (LQE) algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based alone. Kalman Filter has common applications for guidance, navigation and control of vehicles. It is also one of the main topics in the field of robotic motion planning and control[1].

The Kalman Filter represents all distributions of the Gaussians and iterates two things: measurement updates and motion updates. When using Kalman filters, the measurement cycle is usually referred to as a measurement update, while the motion cycle is usually called prediction[2].

Kalman Filter allows us to model any linear system accurately[3]. For more complex systems, the Extended Kalman Filter (EKF) may do the job. In EKF, the state transition and observation models need not be linear functions of the state but may instead be nonlinear functions[1]. There is also an unscented Kalman filter that uses a deterministic sampling technique known as the unscented transform to predict highly nonlinear system[1].

In this project, we implemented basic Kalman Filter with multi-dimensions and with other assistant functionalities to predict motions of the hexbug.

2. Algorithm Overview

2.1. Basic Kalman Filter Implementation

We implemented a 6 X 6 kalman filter (x, y, dx, dy, ddx, ddy) to predict the robot's movement. x, y, dx, dy, ddx, ddy represent robot's x, y coordinates, velocities at x, y dimensions and accelerations at x, y dimensions. The details about the kalman filter implementation are similar to what we learned from the class. For simplicity, we assume the velocities of the robot remain constant within each step time (dt). Thus, the following equations apply:

$$x = x + dx$$
$$y = y + dy$$
$$dx = dx + ddx$$
$$dy = dy + ddy$$

At the very beginning, we implemented a visualization module in our program, thus we can directly compare the predictions to measurements. As shown in Figure 1, the blue line represents the measurements while the red line represents the KF predictions. The predictions are quite close to the measurements at the beginning. However, once the robot hits the wall, the KP predictions overshoot the wall and took quite long time to adjust the robot's orientation and move back into the box. It suggests that kalman filter is good at handling linear movements, but not the collisions when applied alone. To solve the problem, we implemented two simple bouncing models to the basic kalman filter.

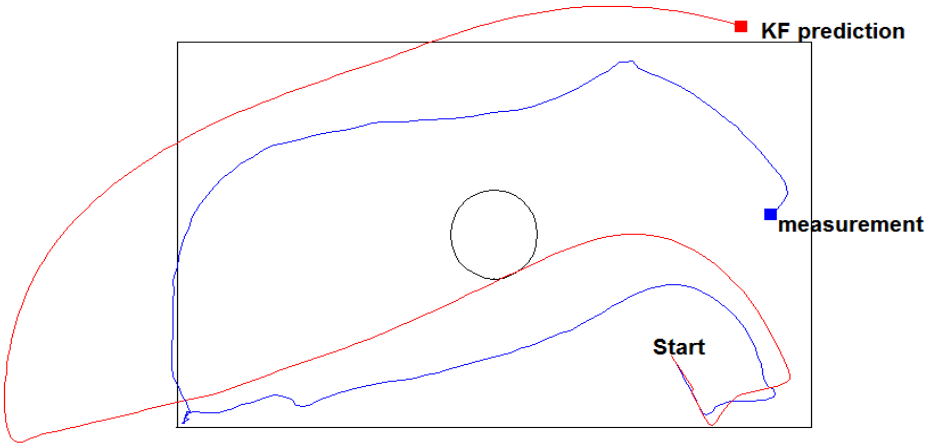


Figure 1. Using a basic kalman filter alone to predict robot's movement.

2.2. Bounce Modeling

We implemented two simple bouncing models in the basic kalman filter. The first one showed how the robot bounced at the walls of box (Figure 2), and the second one showed how the robot bounced at the center obstacle (Figure 3).

- Bounce on wall

The environment of the robot (x_min, x_max, y_min, y_max, center, radius) was easily obtained from training data. We examined the KF predicted position of robot using x = x + dx, y = y + dy, and if the position was out of the boundaries, instead of using kalman filter to update the robot's conditions, we updated its conditions manually. As shown in Figure 2. The robot's current position is (x, y) and the kalman filter is supposed to update its new position to (x+dx, y+dy) which is out of the x_min boundary. So we assumed the robot contacted the wall at a very short period of time and then moved at the same y direction but an opposite x direction.

Also, we have tried two different bounce models, one was a traditional perfect reflection, and the other is shown in Figure 2. It seemed that the bug was not bounced to the perfect reflection direction, but rather that the bug moved up a little bit and then bounced. This can be explained by the fact that the actual robot head is approximate hexagon shape and the extracted centroid coordinates can not represent the point of contact when hitting at different angles. The robot's next state (including the position) is then manually updated to (x_min, y+dy, -dx*α, dy) instead of (x+dx, y+dy), where α was a coefficient which represented the x dimension velocity loss due to the collision. In implementation, we found that to obtain a least L^2, the best value of α was 0.1.

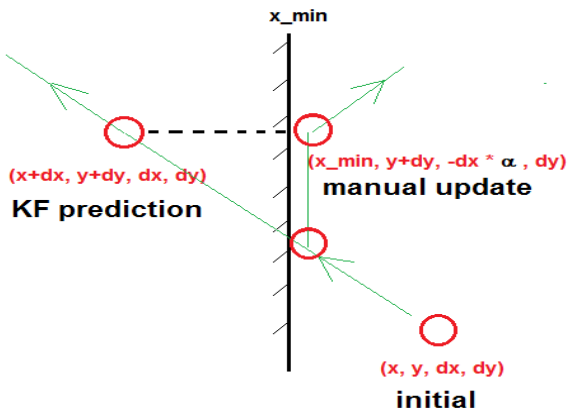


Figure 2. Illustration of robot bouncing at the wall

- Bounce on center obstacle

The bouncing at the center barrier is more complicated compared with the bouncing at the walls, and we implemented three help functions to solve this problem.

distance
takes two parameters (positions) and returns their distance.

interaction
identifies the touch point at the obstacle surface. Let's assume robot's current position as P1 (x, y), and KF predicted future position as P2 (x+dx, y+dy). If the distance between P2 and center is less than the radius of the obstacle, the robot will hit the surface of the obstacle and bounce back. To get the touch point, we tried a simple strategy by moving P1 to P2 at tiny steps 1/20 (dx, dy) until the distance between touch point and center close to radius.

collision
takes parameters as velocities at x, y dimensions, the touch point and the center of the obstacle, and return the new velocities at x, y dimensions through vector calculation assuming perfect reflection.

- Result

After implementing the additional bouncing models into the basic kalman filter, we observed significant improvements for the prediction of robot's movements. As shown in Figure 4, the blue line represents the robot's real movement and the red line represents KF prediction.

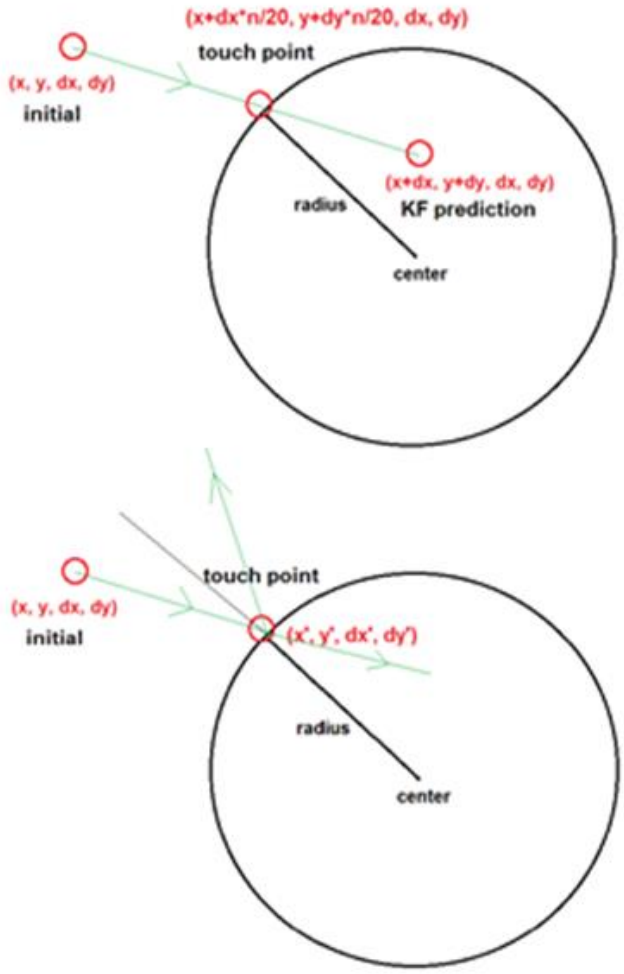


Figure 3. Illustration of robot bouncing at the center obstacle. (upper one) showing how to calculate the touch point, and (lower one) showing how to calculate the new velocity vector.

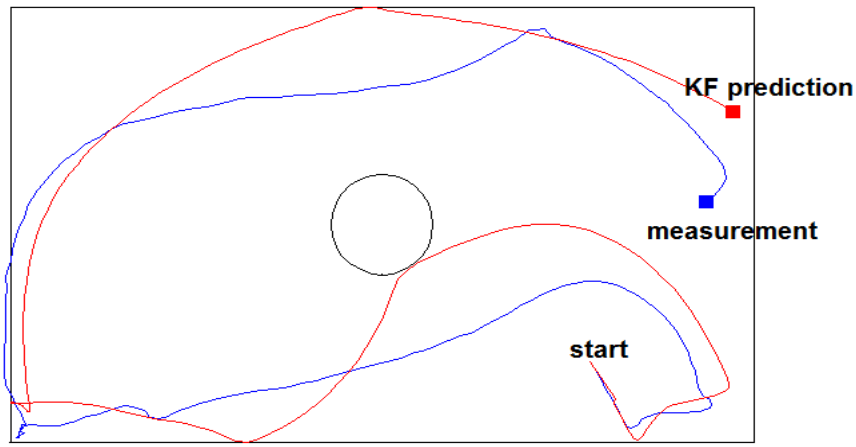


Figure 4. Using an updated kalman filter to predict robot's movement.

3. Validation

The accuracy of our program is examined by two different approaches.

1) Since we implemented a visualization module in our original program, we can check the accuracy through direct visual comparison. We pretended that we had no knowledge of the last sixty pairs of data, and used our program to predict the last sixty pairs of data. Then we visually compared the predicted movements with the actual movements.

2) Using grading.py to examine the accuracy of our program. we followed the instructions on piazza, created fake input files (removed last 60 pairs data) and fake actual files (60 pairs of data points transformed from original input files). Then we ran grading.py to generate the L^2 error of our program's predictions.

4. Optimization

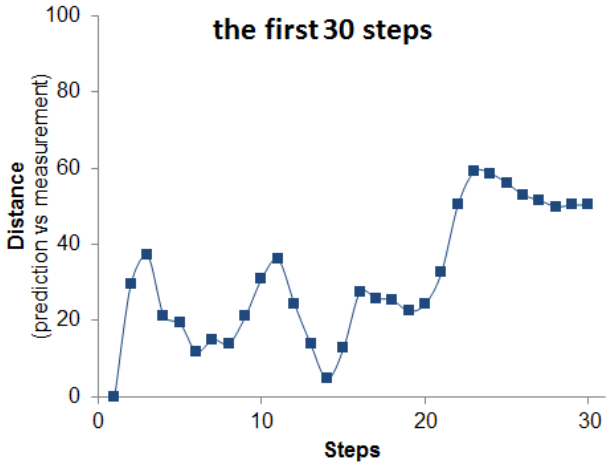
We finally used the last 30 steps to predict the future 60 steps. The rational to use the last 30 steps is as following:

At beginning, we used the given data as measurements, and implemented Kalman Filter to predict the bug's next position. Then we calculated the distance between each measurement and the corresponding prediction (Figure 5). We also visualized the first 400 measurements and prediction in Figure 6.

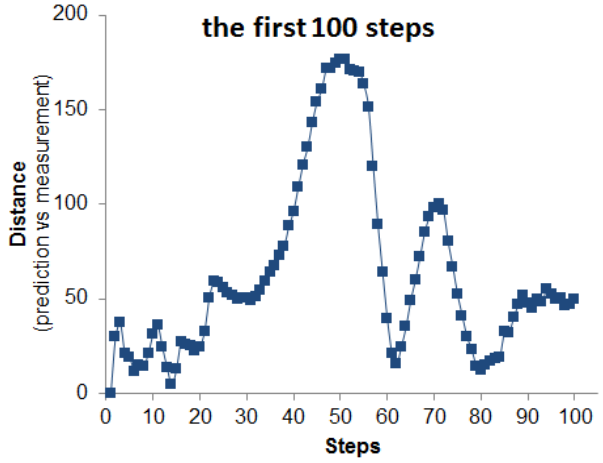
As shown in Figure 5c, the distance between the prediction and the actual position fluctuated with the moving steps and trended up. This might be caused by robot movement noise. For example, the robot may skid sometimes, which makes such movement unpredictable. As shown in Figure 5a, within the first 100 steps, there was a huge spike indicating the unpredictable movement of the robot. So it would not be wise to use more than last 100 steps to predict future steps.

We then took the number of last frames that was used to make predictions as a parameter to optimize the prediction. We had used the last 5, 10, 20, 30, 40, 60, 90 and 120 steps to make predictions, and resulted L^2 errors were showed in Figure 7. By comparing the L^2 errors, it's obviously that the last 30 steps is the best. The visualization of using the last 30 steps to predict the future 60 steps is shown in Figure 8.

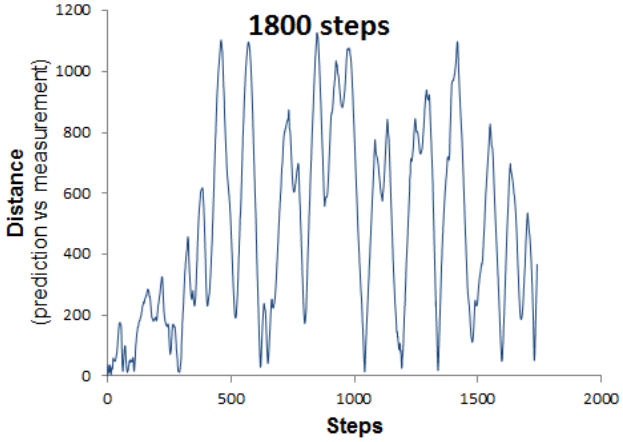
The reason behinds this might be that Kalman Filter is generally used in a situation where the running robot interferes with no other object. However, the robot in our case runs in a confined area with a center barrier, and bounces frequently on the walls and the barrier. When a large number of frames are used for prediction, an accurate bouncing model needs to be included. Otherwise, prediction based on a large number of frames makes L^2 errors increases over time.



(a) The first 30 steps



(b) The first 100 steps



(c) The first 1800 steps

Figure 5. Distances between the measurements and the predictions: (a) the first 30 steps; (b) the first 100 steps; (c) the first 1800 steps

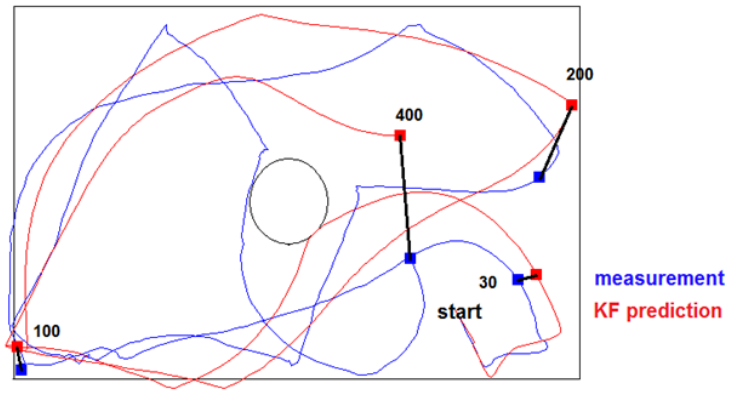


Figure 6. Illustration of distances between the measurements and KF predictions at 30, 100, 200, 400 steps

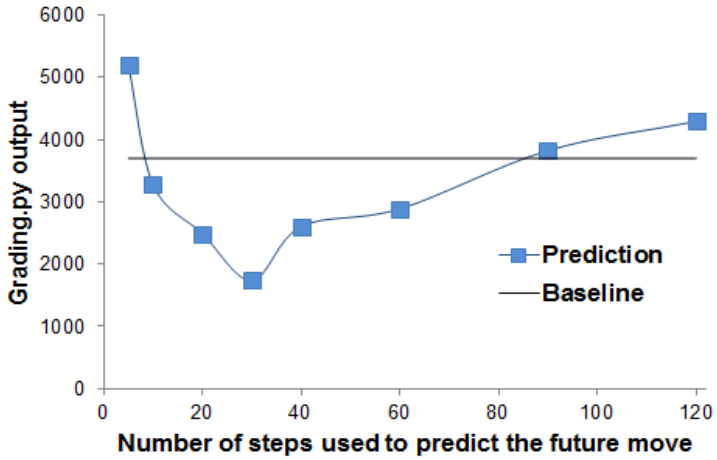


Figure 7. The L^2 errors vs. the numbers of steps used for prediction.

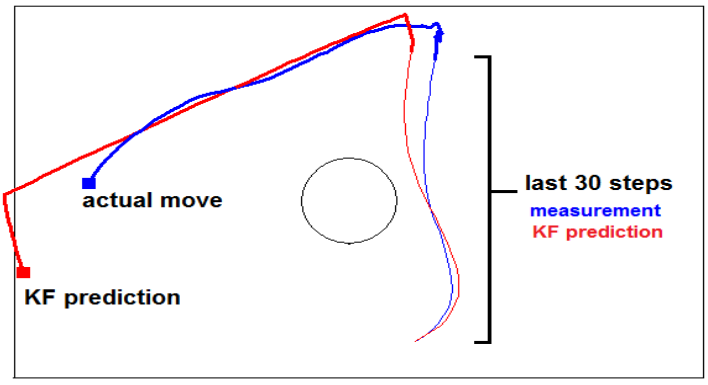


Figure 8. Visualization of using last 30 steps to predict the future 60 steps

5. Functionality Summary

Since the 10 test data are taken from the same setting, we used the training data to figure out where the edges of the box were, as well as the center and radius of the central barrier. We sorted the minimum and maximum values of x and y to locate the edges of the box. The coordinates of the center of the box were the average values of the minimum and maximum of x and y, respectively. We were assuming the center of the box and center of the barrier overlapped, so the radius of the round barrier was the minimum distance between the training data and the box center.

There were two types of robot's position predictions: non-collision prediction and collision prediction.

For the non-collision free movement, Kalman filter was used to track the position of the target robot. A 6-dimension matrix was fabricated including location, velocity and acceleration information. The prediction was based on previous measurements.

For prediction involving a collision, a modified bouncing ball model was used to model the movement of the robot hitting the wall or the center barrier. We updated our prediction of robot as shown in Figure 2 and Figure 3.

If basic Kalman filter is applied alone, the prediction is problematic when robot hits the wall or barrier (figure 1). With the updated Kalman filter including the bouncing model, our prediction improves significantly (figure 4).

The edge and center barrier information were used directly in the test. The test file was read into our program and the last 30 steps were used to predict the future 60 steps by Kalman filter with minimum errors, as shown in Figure 7.

We found that as the number of steps increasing, generally speaking the distance between the measurement and prediction increased too. So it's better to use a small number of last steps to predict the next position. However, if only a few steps are used to predict the future steps, big L^2 error would incur due to no enough information extracted from the given data. Visualization of using last 30 steps to predict the future 60 steps is shown in Figure 8.

6. Reference

[1] Kalman Filter, ***Wikipedia***, https://en.wikipedia.org/wiki/Kalman_filter
[2] Lesson 2: Kalman Filters, ***OMSCS 8803-O01 Lecture Slides***, Fall 2015
[3] How a Kalman Filter works, in pictures, ***Bzarg blog***, http://www.bzarg.com/p/how-a-kalman-filter-works-in-pictures/