COMPUTER VISION, LOCALIZATION AND NAVIGATION

A. MORIARTY, K. SVENDSEN, W. OKANSKI

1. Question 1: Vision & Localization

The first part of this question was to create a program capable of finding red blobs. The code would take an image captured by the webcam. It was then run through a series of filters. The first filter was a Gaussian filter that would blur the image. The next filters the image based on hue, and intensity of the image removing any non-red objects. Next the filtered image runs through another Gaussian filter. The final two filters are dilate and erode. The former caused the image to stretch while the latter decreases the resolution. This creates a white mask designates where red blobs are found. Next we determine the largest red blob as the blob we are looking for based on the assumption that all other red blobs are minor background noise.

We trained a Support Vector Machine with a Linear Kernel with the pixel coordinates as input data and motor values as labels. After training we then combined the sensor values from the camera with our predicted values from our motion model using a kalman filter. Between the predict and update stages of the kalman filter algorithm, we move the motors. An added while loop waits for the vision data.

```
# predict
X, S = kf_predict(X, S, d*U)
# move
m_1.turn( d*20,120,False)
m_2.turn(-20,120,False)
z = None
while z == None:
z = v.get_local()
Z = clf.predict(z)
# update
X, S = kf_update(X, S, Z)
```

Date: November 21, 2012.

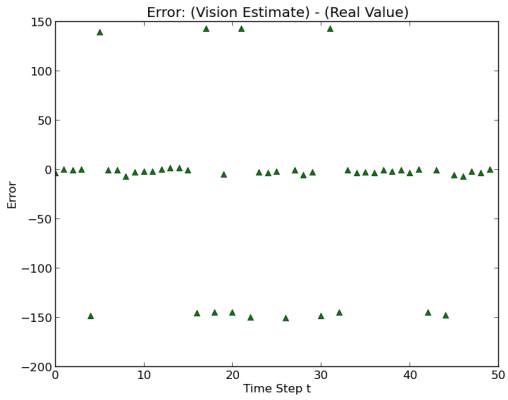

FIGURE 1. Vision vs. Real

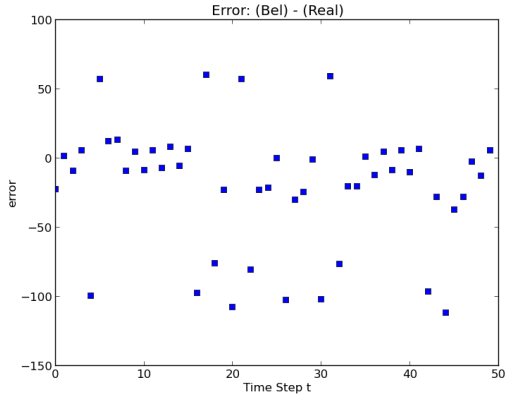

FIGURE 2. Difference in Bel(x) and Real

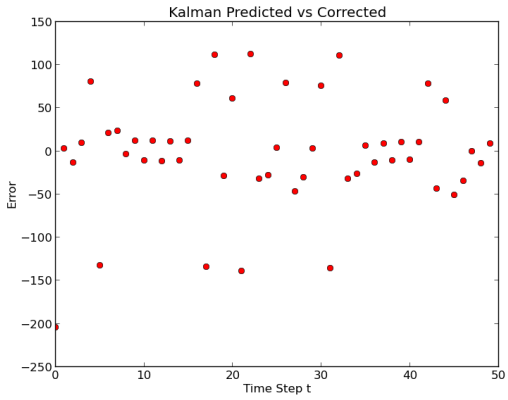

FIGURE 3. Difference in Bel(x) between Predict and Update

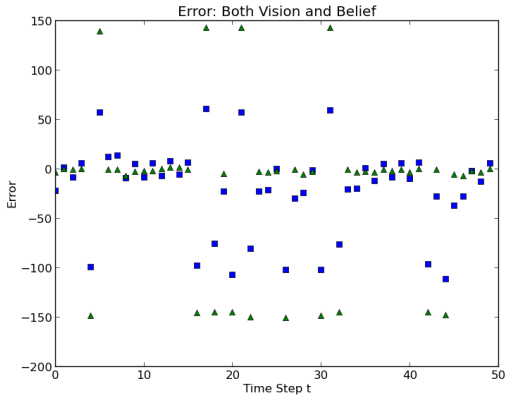

FIGURE 4. Error in belief (Blue) and vision (Green)

2. Question 2: Navigation

This exercise required that the robot arm navigate around an obstacle to reach an endpoint, from a start point. By describing the space around the robot arm with a matrix (alpha x beta) of a 5% resolution (i.e. the resolution of our grid was divided by a factor of 5 to increase speed at the consequence of accuracy, we were able to assign rewards and penalties to the endpoint and the obstacle(s)/boundaries of the space respectively. One of the first problems we encountered was in our code. When implemented, the robot arm seemed to move only partially to its destination. A visualization of the matrix showed that the reward set at some point (alpha, beta) was only propagating to points in the beta direction but not the alpha direction. This was found to be a technical problem where the iterations of the policy in the alpha direction (up/down) were not based on the calculations of the value function but instead based on values equal to zero due to the resetting of the loop. Further complications occurred with the setting of certain constants such as the resolution of the space, reward/punishment values and the discount factor of our value function. We discovered a difficulty between the two motors of the robot arm, being that if the second portion of the arm couldnt move towards the reward destination without intersecting with the punishment region of the space, it would get caught in a loop in the policy instead of the first portion of the arm adjusting to allow the space required. We overcame this issue with a couple methods. The first was effective but it relied on increasing the speed of the motors and sampling time of the position of the arms. This lead to consistent overshooting of the obstacle and problematic space mentioned previously but ultimately resulted in constant overshooting of the final destination. The second and more successful method was to fine tune a number of the parameters including increasing the punishment of the obstacle and the reward of the destination, increasing the discount factor to enhance immediate reward than later reward and decreasing the resolution to add noise to the space. The following graphs show the occupied space and the path the robot arm took to reach the desired location. The desired location was a hard coded value that was arbitrary set and the starting location was a random point that the arm was left at. The program would have some difficulty when there was a limited range of motion without interception with the block as demonstrated around angle 5,0.

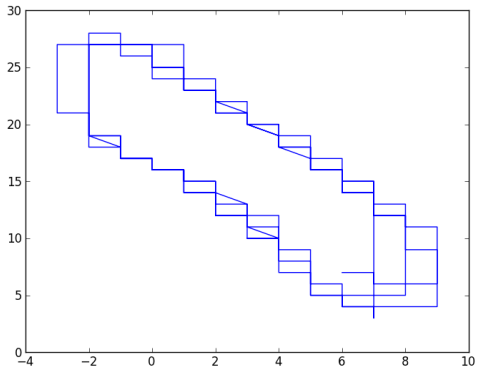

FIGURE 5. Objects outline in space grid map

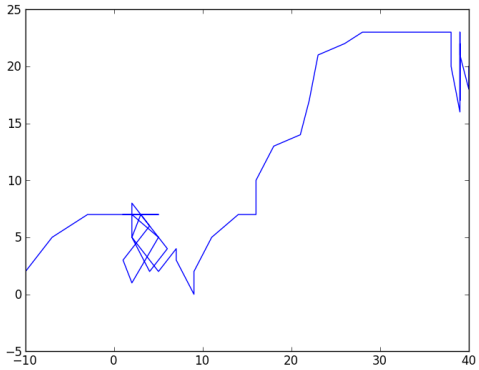

FIGURE 6. Path of Robot Arm to get to desired location

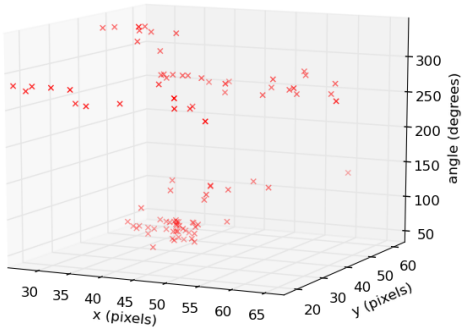

FIGURE 7. Sample of Training Data

3. APPENDIX

Code from Kalman Filter. In higher dimensions mutliplication would need to be matrix dot product. Constants set as parametres for easier testing.

```
def kf_predict(X, S, U):
x = A*X + B*U
s = A*S*A + R
return x, s

def kf_update(X, S, Z):
k = S*C/(C*S*C+Q)
x = X+k*(Z-C*X)
s = (1-C*k)*S
return x, s
```

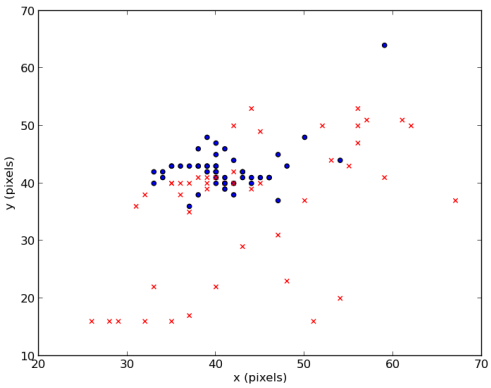

FIGURE 8. Labelled Training Data