COMP 4680/8650: Advanced topics in Statistical Machine Learning

Programming Assignment 2: Robot Localization

Manab Chetia, u5492350,

Vadim Solviev, u4573325

1. **Probabilistic Filtering on a Hidden Markov Model**
2. Given,

Yt

Y3

Y2

Y1

We use product rule to express the joint distribution of sequence of measurements

Assuming each of the conditional distributions on R.H.S is independent of all previous measurement except the recent, we obtain

From d-separation property, we see conditional distribution for measurement , given all of observation up to time is given by,

Now for each measurement , we have a latent variable and assuming latent variables form a Markov chain, which gives rise to a state space model. It satisfies the conditional independence property that and are independent given i.e. .

X0

X1

Xt

X3

X2

Yt

Y3

Y2

Y1

The joint distribution for this model is given by,

1. Using conditional independence in the graph we get,

Therefore,

Now,

Hence,

1. Using conditional independence in the graph we get,

Therefore,

Now,

Hence,

1. Bayesian filtering has 2 steps: Prediction and Correction. So each time we have a measurement we can improve our belief over the state at time t recursively. We define the belief over the state at time as
   1. Prediction
   2. Correction

The prediction and correction step can be obtained from derivation a) and b) respectively as proved above.

Combining prediction and correction steps we get,

Therefore we can derive recursively.

1. **Robot Localization: Particle Filter**
2. Motion model

Let,

distance to be moved in direction of new is represented by variables and respectively i.e.

For motion update, we compute the posterior over distribution one time step later and which is the convolution of transition probability, multiplied by prior i.e.

Here represent the set of particles. We take a random particle from and apply the motion model (+ noise model) to generate a new particle . Finally we get a new set of particles .

Measurement model

For the measurement update, we compute the posterior over state, given a measurement update, which is proportional to normalization of probability of measurement given the state and multiplied by i.e.

where,

1. I tried 2 ways to get the most likely trajectory:

**Method 1:**

To get the most likely trajectory, we save all waypoints that a particle goes through *(based on the initial to final odometer reading)* and we call each of them as . So, if there are 1000 , there are 1000 different for each particle. We then select that particle’s path which gives us the highest

where

**Method 2:**

Here, at each step we get the particles associated with the top 10 highest weights and we did a weighted average over them to get new .

**Most likely trajectories**

Easy

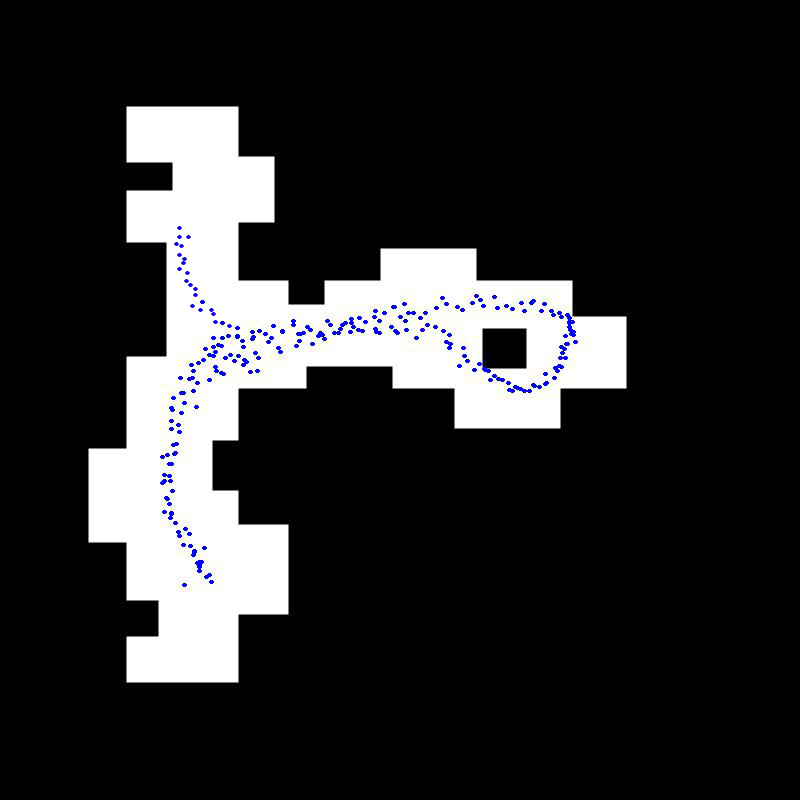


Figure 1: Easy map using method 1

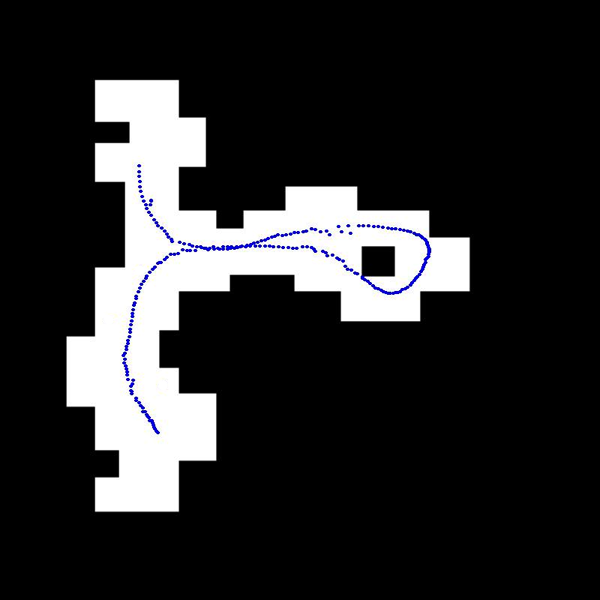


Figure 2: Easy map using method 2

Hard

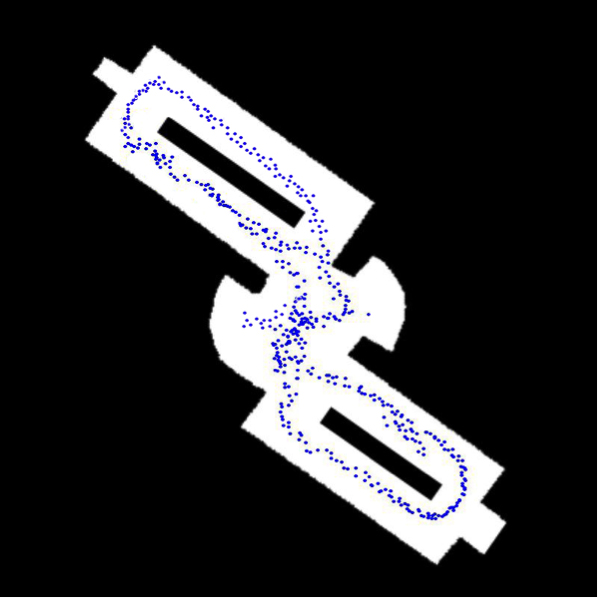


Figure 3: Hard map using method 1

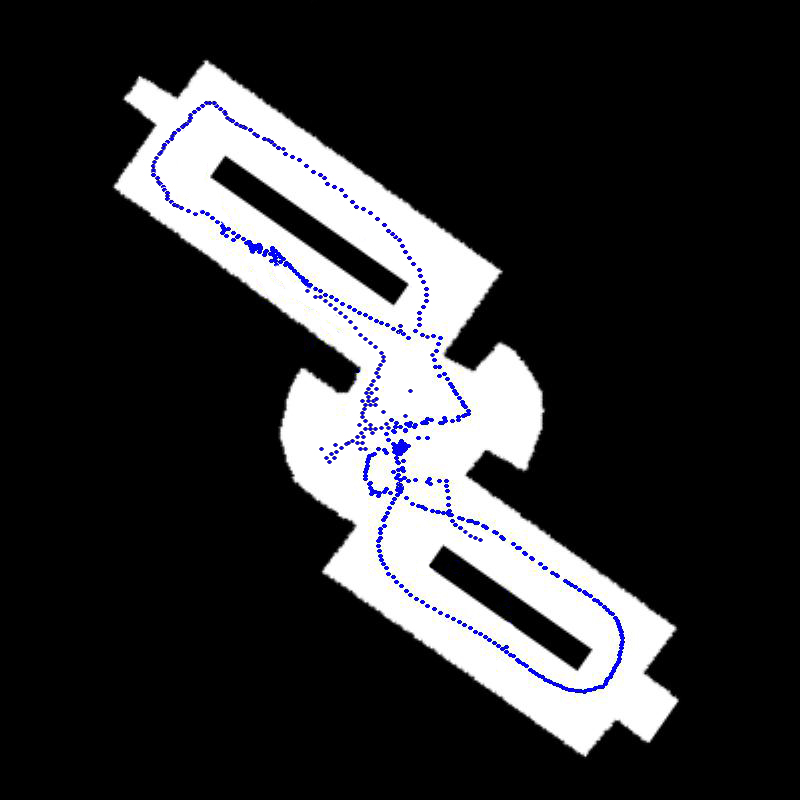


Figure 4: Hard map using method 2

1. The algorithm becomes most confused about the location of the robot in the places where there are symmetries in the map.

The estimate of the robot location becomes better with the increase in number of particles. Particle filters is an approximate method, therefore higher the number of samples, more accurate is our result. In our case, each particle is a discrete guess as to where the robot might be. Therefore, more the number of particles, more are the number of guesses where the robot might be and all guesses together generate an approximate representation for the posterior of the robot.

To avoid robot being confused, we needed around 3000 particles for the hard map.

1. **The Kidnapped Robot Problem**
2. For the kidnapped scenario, we add some randomly generated particles during the resampling step.

Code:

def **resample**(particles, weights, N):

particles3 = []

index = int( random.random() \* N )

beta = 0.0

mw = max(weights)

kidnapProb = 0.02

correctProb = 1-kidnapProb

for i in xrange( int(correctProb \* N) ):

beta += random.random() \* 2.0 \* mw

while beta > weights[index]:

beta -= weights[index]

index = (index + 1) % N

particles3.append( particles[index] )

*# Generating random particles*

for i in xrange( int(kidnapProb \* N) ):

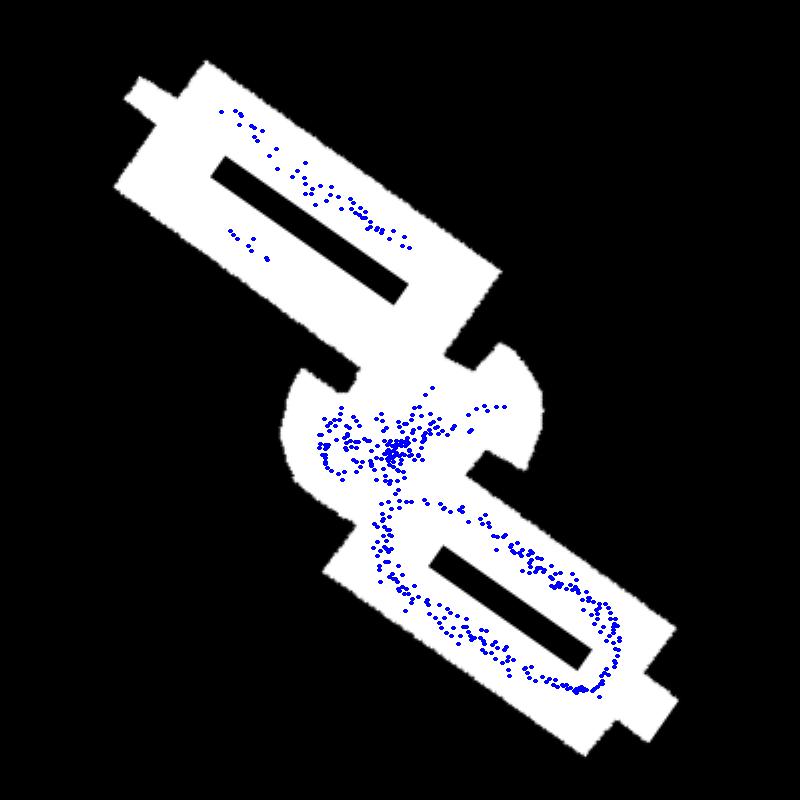
r = robot()

r.set\_noise(0.1, 0.1, 1.0)

particles3.append( r )

return particles3

1. Sd



1. asd