# Probabilistic Robotics Lab Report
# PARTICLE FILTER

Raabid Hussain

April 3, 2016

## 1 Overview

The objective of this lab was to implement a particle based filter called Monte-Carlo localization algorithm for localization of the turtlebot. The program was to be tested using turtlebot simulator. A bag file containing the environment was provided along with the robot instructions for moving. We had to predict the current position of the robot using particle filters. For the sake of computational cost, the number of particles was fixed to a maximum of 500. The walls in the map were represented as lines using the code for the previous lab in which split and merge technique was used to define geometry of the walls using straight lines. These lines were used as input to the particle filter.

## 2 Methodology

Monte-Carlo localization (MCL) is a form of particle filter localization, based on Markov localiztion problem. Provided with a map of the environment, the MCL algorithm estimates the position of the robot as it is moving through the map with the help of particles placed throughout the map. The MCL is an update of the Bayes filter. It uses the filter to represent the distribution of the likely states which the robot might be in. The algorithm starts by randomly placing all the particles throughout the map. The particles are assigned equal weights at start. As the robot moves, the particles are moved in same direction and the weights of these particles are changed according to

the likeliness of the state being represented by the particle. Whenever the robot senses something, the particles are resampled following the recursive Bayesian estimation approach.
The algorithm can be divided into three main steps:-

## 2.1 Prediction Step

The first step in the algorithm is to predict the state of the robot based on the motion parameters. We were to predict the x,y and theta parameters of the robots state. The input from the odometry sensors was used to move the particles with respect to the motion of the robot. However, in practice the movement is never perfect and there are many factors that lead to incorrect motion models. So in prediction step, we always need to add gaussian noise to the state. This noise has to be different for each particle, otherwise the results are wrong.
The main thing to do in prediction step was to add the noisy odometry measures to the current state of the robot. However, when I directly ran the code like this, the particles were behaving strangely. It was later found that the current state of the particles and the odometry measures were in different coordinate systems. So, a transformation was required to convert odometry measures into the world coordinate system.

## 2.2 Weighting Step

After the prediction of the new state of each particle, it was necessary to assign weights to each particle depending on the likeliness of the robot to be in that particles state. This weight was computed by comparing the lines in the given map with the lines obtained from the measurement.
In order for the lines, to be compared they were transformed into parametric form in which they are represented by their length (perpendicular distance to the origin) and their angle in the map. A inbuilt function 'getpolarline' was used for this. The lines form the map were transformed into the coordinate system of the measured lines. Then separate weights were calculated for distance and angle using the formula stated in the lab description file. The two weights were multiplied with each other for all the lines and their maximum was taken and multiplied to the previous weight of the particle to get the new weight of the particle.
As an optional step, the lengths of the two lines were compared to see if they

actually corresponded. If the correspondence was not found then that particles weight was assigned zero value to eliminate its effect. These weight were assigned to the particles after normalizing them. Then an efficient number was computed using the equation in lecture notes to decide whether to go for re-sampling or not.

### 2.3 Resampling Step

After the weighting step, the algorithm calculates the efficient number to decide whether to resample or not. If resampling is required, then the algorithm generates a random number and starts with the weight of the first particle to run the resampling algorithm as mentioned in the lecture notes. A problem encountered here was that the random number being generated was giving me an empty ([]) number. For this, I changed the function being used to generate the random number.

## 3 Results

After writing the code for all the three stages, it was time to test them. It was observed that the particles initially moved in random directions and after some resamplings, they all converged to the location of the robot. However, since the algorithm is based on probability, the algorithm was observed to behave correctly approximately 8/10 times. A sample output is shown in the following figure:-

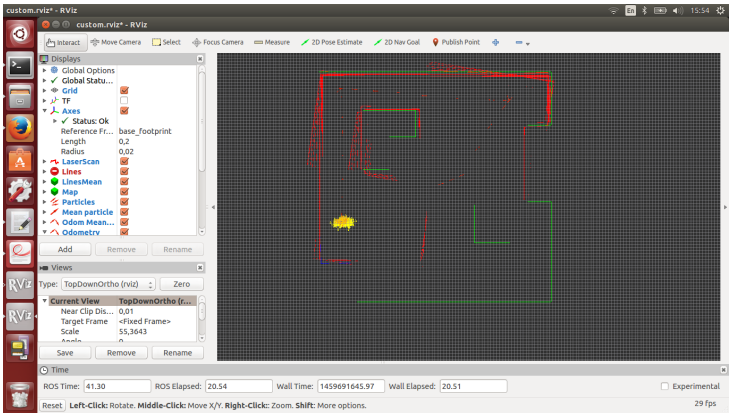


Figure 1: Output of the algorithm at a particular moment in time

# 4 Conclusion

In this lab work, a particle based filter was implemented using ROS. The algorithm was divided into three functions, 1 each for its main steps. A maximum of 500 particles were used for the prediction of the robot state. Weights were assigned to these particles according to the measurements obtained. The algorithm was found to behave correctly approximately 8 out of ten times owing to the probabilistic nature of the algorithm. This time more than 4 days were given for the lab, which proved helpful in completing the tasks.