Lab3 Report - Particle Filter

Mohammad Rami Koujan

1 Introduction

The Particle Filter is an alternative nonparametric implementation of the Bayes filter. Just like histogram filters, particle filters approximate the posterior by a finite number of parameters. However, they differ in the way these parameters are generated, and in which they populate the state space. The key idea of the particle filter is to represent the posterior by a set of random state samples drawn from this posterior. Instead of representing the distribution by a parametric form (the exponential function that defines the density of a normal distribution), particle filters represent a distribution by a set of samples drawn from this distribution. Such a representation is approximate, but it is nonparametric, and therefore can represent a much broader space of distributions than, for example, Gaussians.
Thus, the main focus of this lab assignment is on Particle Filter or Montecarlo Localization (MCL) algorithm to localize a two dimensional robot (turtlebot) in a given map.

2 Algorithm analysis and implementation

This section presents and analyses the implementation of the Particle Filter with the different functions used. In fact, the particle_filter.py file contains the "ParticleFilter" class which has the three main functions (predict, weight, and resample) required to implement this filter. Those functions are called inside another class, LocalizationNode, which exists in node.py file.

Firstly, "predict" function is indeed the one responsible for computing the new position (x, y, θ) of the particles given an odometry measurement $(\Delta x, \Delta y, \Delta\theta)$ in the vehicle frame. In order to do so, it first adds the odometry measurement Δx and Δy to "self.p_xy" parameter after adding a Gaussian noise, with zero mean and "self.odom_lin_sigma" standard deviation, and transferring the result to the particle frame. The transformation is performed by the means of rotation by "self.p_ang" angles where the following rotation matrix is used:

$$Rot(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix}$$

Then, it also adds the odometry value of $\Delta\theta$ to "self.p_ang" parameter after adding a Gaussian noise with zero mean and "self.odom_ang_sigma" standard deviation. The last step in this function is to update the resampling flag which indicates that the robot has moved form its last position. In other words, this function implements the prediction step of the particle filter. However, the following two functions are in charge of the correction step of this filter.

Secondly, the "weight" function computes the weight of each particle by comparing the lines obtained from the measurements with the lines of the given map. To obtain the lines of the measurements, the "splitandmerge" node is used to extract those lines and return their first and last points' x and y coordinates. Three "for" loops, two of them are nested with the first one, are used to perform the main task of this function. The first one iterates on each each particle getting the polar (R and θ) coordinates of the map lines, inside a nested loop, from their Cartesian coordinates. The polar coordinates are, in fact, calculated with respect to the current particle of the first loop. After that, another loop starts to iterate through each measured line obtaining its polar coordinates and calculating the WR and Wθ weights based on the following Gaussian weight:

$$W = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] \tag{1}$$

where x represents the range or angle of the current measured line and μ the range or angle of each one of the map lines, when computing WR and Wθ respectively. Thus, WR and Wθ are vectors representing the range and angle weights of each measured line with respect to all map lines. The final weight of each particle is obtained by first calculating the maximum number of the vector that results from multiplying WR and Wθ for each measured line and then multiplying the resultant maximum numbers from all of the measured lines.

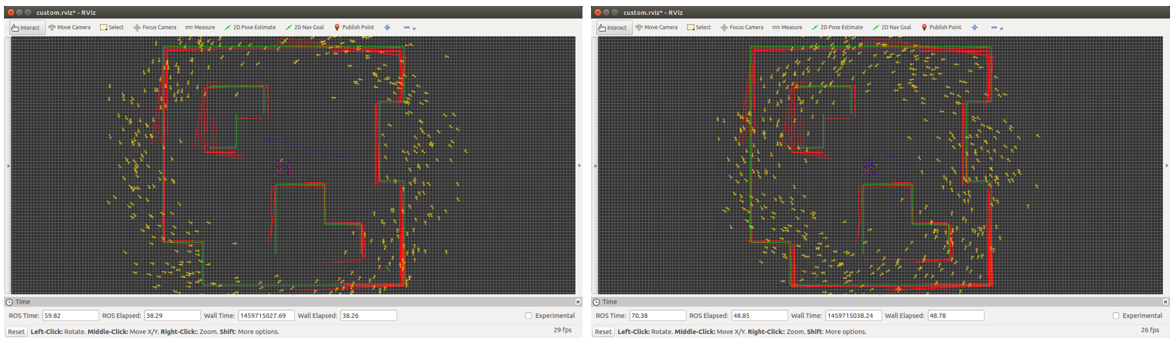
Finally, the resampling function implements the systematic low variance sampler. The main idea of this resampler is to generate initially a random number between zero and $\frac{1}{self.num}$, where self.num is the total number of particles, and then for each particle the resampling function increases the value of this random number by the index of the current particle divided by self.num and stores the result in another variable, say u. Thereafter, it compares u to the weight of the first particle. If it is bigger, it keeps aggregating the weights of the next particles till the aggregated number becomes greater than u. Then, it stores the x,y, and angle values of the last aggregated weight as the first resampled particle and keeps repeating the procedure for each particle. The final step after resampling is to set all the new particles weights to $\frac{1}{self.num}$. Figure 1 shows the pseudo algorithm of the systematic low variance resampling.

```
Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):
    $\bar{\mathcal{X}}_t = \emptyset$
    $r = \text{rand}(0; M^{-1})$
    $c = w_t^{[1]}$
    $i = 1$
    for $m = 1$ to $M$ do
        $u = r + (m - 1) \cdot M^{-1}$
        while $u > c$
            $i = i + 1$
            $c = c + w_t^{[i]}$
        endwhile
        add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
    endfor
    return $\bar{\mathcal{X}}_t$
```

Figure 1: Low variance resampling for the particle filter

3 Results

This section focuses on the results of running the previously explained algorithm on a turtlebot in ROS. The following figure shows the result of running the prediction part of the Particle Filter.



ROS Time: 59.82 ROS Elapsed: 38.29 Wall Time: 1459715027.69 Wall Elapsed: 38.26 29 fps

ROS Time: 70.38 ROS Elapsed: 48.85 Wall Time: 1459715038.24 Wall Elapsed: 48.78 26 fps

Figure 2: Prediction step of the Particle Filter

The preceding figure demonstrates that the particles lie on almost a circle and keep rotating and changing the radius of this circle according to the robot movement. This is an expected result since the particles weights are always the same in this case and there is no resampling step. Next figure presents the effect of including the prediction and correction steps on the particles. As it is shown, the particles now tend

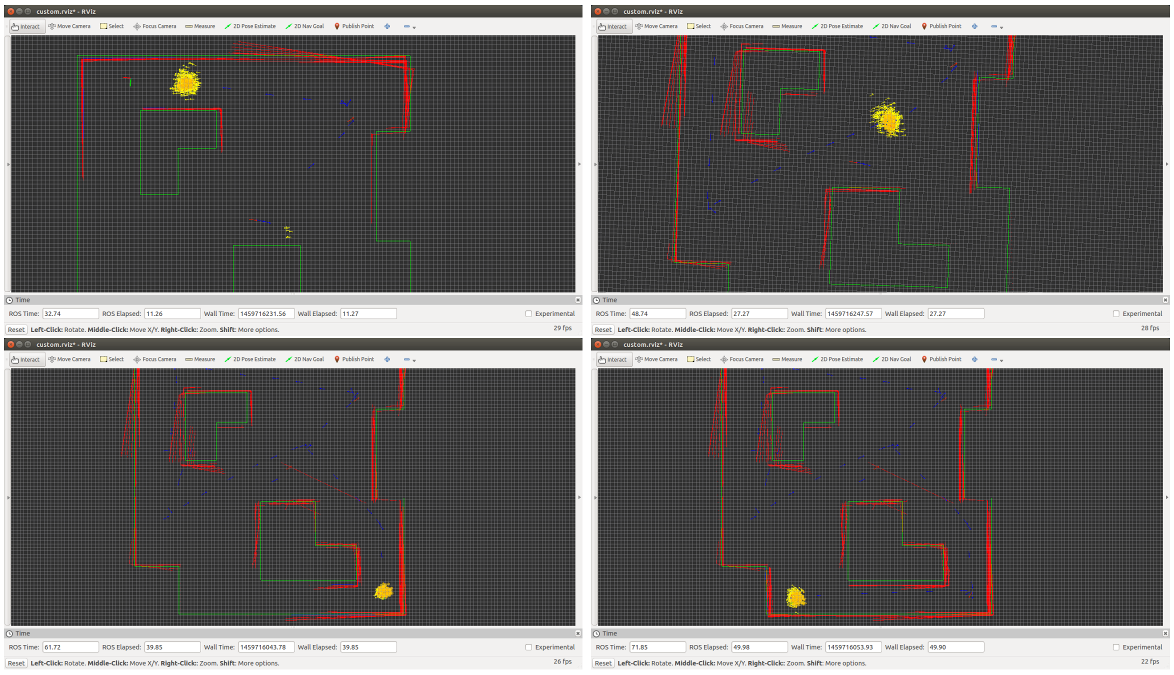


Figure 3: visual results of the Particle Filter

to follow the turtlebot as it moves and consequently can be used to localize the turtlebot in the shown map.

In the optional part of this assignment, an improvement for the localization is required by taking into account if the segments extracted from the measurements coincide with the segments in the given map. One possible approach to achieve this, is to compare the length of the map line with the measured one and if it is smaller, the weight of this particle with respect to this line is set to zero. In fact, this modification was implemented in only the "weight" function and the following figure shows the results of running the particle filter after incorporating this improvement.

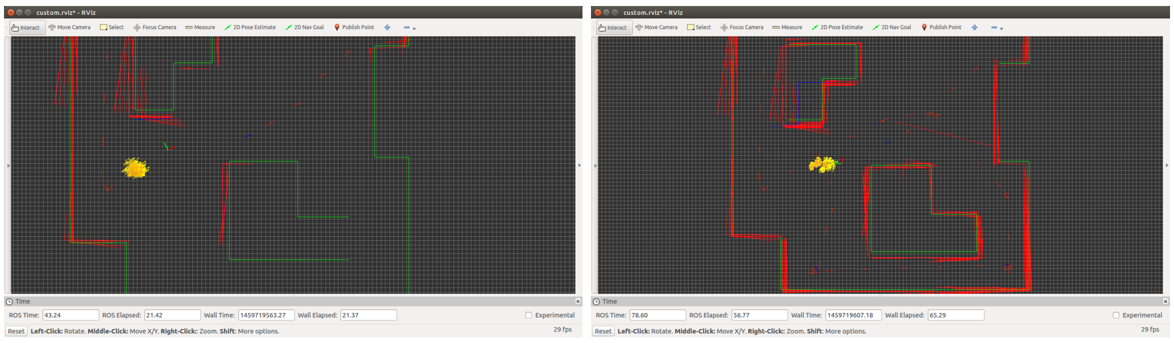


Figure 4: Prediction step of the Particle Filter

Actually, the effect of this improvement is that the particles now localize the turtlebot better and the percentage of algorithm convergence to the ideal solution gets higher with the same number of trails as before.

4 Conclusion

In this lab session, a Particle Filter algorithm based on lines extracted from the turtlebot map is designed and implemented successfully using python language. Inspecting the results signifies that odometry and some sensors information alone are not sufficient for giving precise details about the robot position which necessitates the use of some filtering algorithms like the Particle Filter in order to localize it more precisely even though the given measurements are noisy.