Homework 1 - Robot Localization using Particle Filters
16-833: Robot Localization and Mapping, Spring 2019

Rishi Madhok (rmadhok), Syed Ashar Javed (sajaved), Talha Ahmad Siddiqui (tsiddiqu)

1 Introduction

In this assignment, we address the problem of localizing a lost robot within a known map, given its odometry and laser rangefinder data. We implement a particle filter that (1) initializes N particles, (2) uses odometry measurements as principal input to the robot's motion model, (3) uses laser ray casting to assess particle quality with respect to real world laser measurements, (4) resamples the particles when the weight variance rises about a threshold. Particle filters are great for situations in which we have sensor and motion data from a known map but no ground truth. We observe good localization performance with a choice of $500 - 800$ particles and noise parameters of $\alpha_1 = .0001, \alpha_2 = .0001, \alpha_3 = .01, \alpha_4 = .01$ for the motion model. As our videos demonstrate, our implementation worked successfully on the first and second data logs.

2 Implementation

We implement our entire system in Python.

2.1 Particle Initialization

Given that the dataset depicts a robot traveling through a building, we can assume that the robot was always inside either a room or the hallway. Thus, we generate our initial samples as a uniform distribution over all pixels that are known but unoccupied with a probability of one (see Fig. 1).

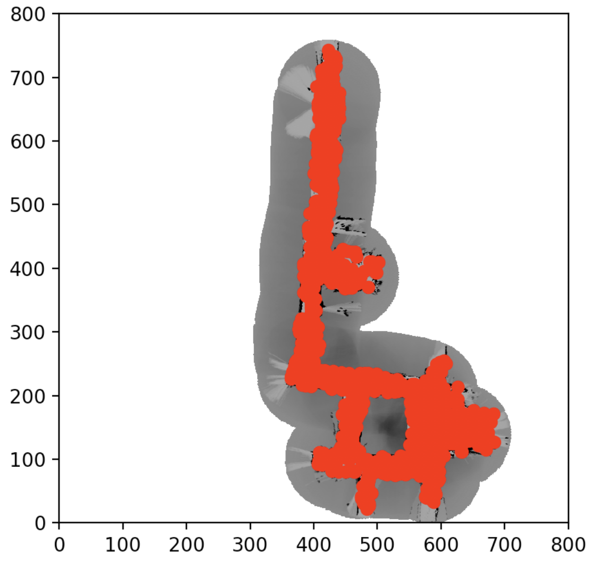


Figure 1: Particle Initialization: Map with initialization of 500 particles.

2.2 Motion Model

We use the motion model described in Thrun et al. [1] in this assignment. Our motion model uses the odometry data only to estimate the motion relative to the current particle frames. We also incorporate Gaussian noise to model the presence of inaccuracies in the odometry sensing.
To get motion estimates from one odometry configuration $(\tilde{x}, \tilde{y}, \tilde{\theta})$ to another $(\tilde{x}', \tilde{y}', \tilde{\theta}')$ on a two dimensional plane, the robot needs to perform a rotation to align with the direction of travel, a translation along that direction, and a rotation to adjust itself to the final orientation (Figure 2).

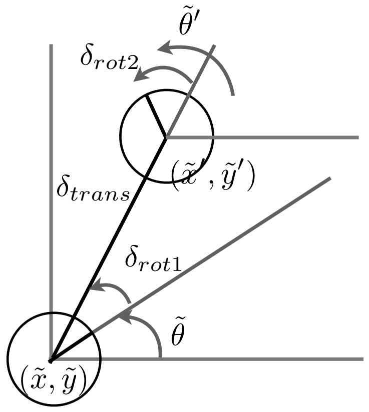


Figure 2: Motion model for a robot moving from configuration $(\tilde{x}, \tilde{y}, \tilde{\theta})$ to $(\tilde{x}', \tilde{y}', \tilde{\theta}')$

The initial change in rotation can be specified as:

$$\delta_{rot1} = \text{atan2}(\tilde{y}' - \tilde{y}, \tilde{x}' - \tilde{x}) - \tilde{\theta}$$

The change in translation is the distance travelled between the initial and final odometry reading:

$$\delta_{trans} = \sqrt{(\tilde{x}' - \tilde{x})^2 + (\tilde{y}' - \tilde{y})^2}$$

The final change in rotation is given as

$$\delta_{rot2} = \tilde{\theta}' - \tilde{\theta} - \delta_{rot1}$$

This motion model contains four tunable parameters α_1,α_2,α_3,α_4 that govern the variance in the model. The values for these parameters were selected on the basis of the quantities they affect. We allowed lesser noise in the rotation coordinates, and greater noise in translation. We observed that same amount of noise in both translation and rotation can change the rotation (in radians) by a large amount. After playing with the values, we kept $\alpha_1 = .0001, \alpha_2 = .0001, \alpha_3 = .01, \alpha_4 = .01$.

2.2.1 Incorporating Noise

We assume that the actual values of rotation and translation can be described by removing independent noise ϵ_b with zero mean and variance b:

$$\hat{\delta}_{rot1} = \delta_{rot1} - \epsilon_{\alpha_1 \delta_{rot1}^2 + \alpha_2 \delta_{trans}^2}$$

$$\hat{\delta}_{trans} = \delta_{trans} - \epsilon_{\alpha_3 \delta_{trans}^2 + \alpha_4 (\delta_{rot1}^2 + \delta_{rot2}^2)}$$

$$\hat{\delta}_{rot2} = \delta_{rot2} - \epsilon_{\alpha_1 \delta_{rot2}^2 + \alpha_2 \delta_{trans}^2}$$

The updates to the translation and rotation are given by:

$$x' = x + \hat{\delta}_{trans} \cos(\theta + \hat{\delta}_{rot1})$$

$$y' = y + \hat{\delta}_{trans} \sin(\theta + \hat{\delta}_{rot1})$$

$$\theta' = \theta + \hat{\delta}_{rot1} + \hat{\delta}_{rot2}$$

2.2.2 Sampling Distribution

We obtain all our samples from a normal distribution centered around the quantity of interest:

$$x \sim \mathcal{N}(0, \sqrt{b})$$

To obtain a random sample according to the normal distribution, we use Gaussian noise with mean 0 and variance b, where b is obtained by the equations stated above.

2.2.3 Validation

We tested the motion model individually before integration. First we generated a single particle and ran the the motion model without noise. Its motion mimicked the motion from the odometry data as expected although it's not the same as the final trajectory obtained as the odometry is noisy. Next, we added noise to the motion model, and increased the number of particles. When the particles were initialized at the same position and the motion model ran with noise, we obtained a banana-shaped spread.

2.3 Sensor Model

We use the sensor model as described in [1] which consists of 4 different distributions as shown in figure 3.

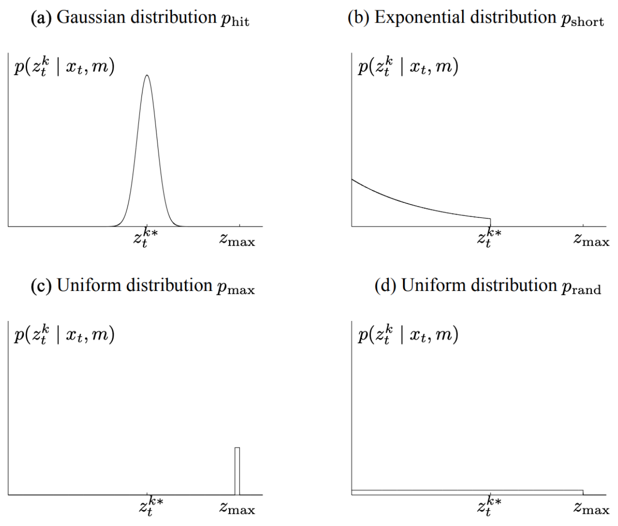


Figure 3: Sensor model is the sum of 4 different distributions

The Gaussian models the sensor measurement with its mean as the true sensor value obtained through ray casting assuming the map is known. The exponential is used for modelling close obstructions near the sensor due to the non-static nature of the map which often prevent the true distance from being measured. In the case of the given logs, we have footsteps in front of the robot which can be explained by this distribution. The dirac-delta like function models the maximum range which can be measured by the sensor and takes care of the cases where the laser does not receive the signal back and assigns the maximum value the measurement. Finally, the low uniform distribution models random and unexplained readings and prevents assignment of 0 probability to such particles.

We also imposed a condition to allocate very low weight to the particles which wandered to obstacle occupied positions without any ray tracing. This helped us not worry about corner cases in ray tracing when the particle is on top of obstacles or in outside the map. This also significantly reduced drift towards non-movable regions. The maximum expected value for a laser measurement was always set to 8192 cm.

The parameters for the sensor model were carefully selected by running multiple iterations across different logs. We found out that giving higher weight to random and Gaussian probabilities helps in localizing much better. Our parameter values are of scales $z_{phit} = 10^3$, $z_{prand} = 10^4$, $z_{pshort} = 10^{-2}$ and $z_{pmax} = 10^{-2}$.

2.3.1 Ray tracing Model

The weight for a particle is computed from the difference between the laser measurement L (captured in the real world by the robot) and an *ideal* measurement $\hat{L}$ taken from the position of the particle in the map. The process goes as follows: (1) we ray cast laser beams from the laser position $[x\ y\ \theta]$ of a particle, then (2) we find the location where the rays hit for the first time an obstacle in the map, and (3) finally use the distances from the the laser position to the hit locations as the components of the ideal laser measurement.

We process all laser beams i, with $i = 0 \dots 179$, as detailed below:

```
$beamAngle_i = \theta + i * \pi/180 - \pi/2$          // beams start from right to left
$orientation_i = [cos(beamAngle_i), sin(beamAngle_i)]$
$starPoint = [laser\_x, laser\_y]$
$endPoint = startPoint$
$curr\_step = step$
while $endPoint < [map_w, map_h]$ :
    $startPoint += curr\_step * orientation_i$
    $endPoint = int(startPoint)$
    $o_i = map(round(endPoint * scale))$
    if $o_i > H$ :
        $break$
    endif
end
$distance_i = norm_2(startPoint - endPoint)$
```

The variables *scale*, *step*, $[map_w, map_h]$, and H represent the map scale, the step to move forward along the beam,[1] the map dimensions and the occupied probability threshold, respectively. The values $distance_i$ compose the ideal laser measurement $\hat{L} = [distance_0\ \dots\ distance_{179}]$.
To obtain a value for the weight of a particle we compute probabilities under a Normal distribution with mean as the true distance from ray cast and variance σ_{beam} which is a hyperparameter we tune. Following [1], we also adopt the 3 other distributions to model the sensor measurement as described earlier. Finally, we sum all probabilities in log scale and then exponentiate.

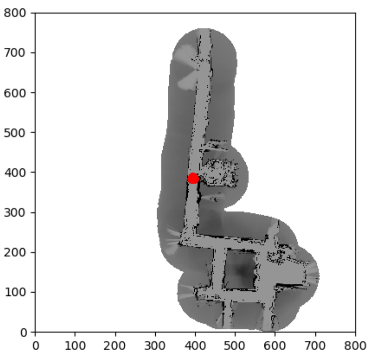
2.4 Resampling Step

We use low variance resampling of particles [1] to update our particle set. This helps us ensure that particles with a high enough weight will not be discarded during the resampling. We do not decay the number of particles over time, nor do we discard particles whose weights fall below a certain threshold. We let our sensor model evaluation and resampling take care of the latter.

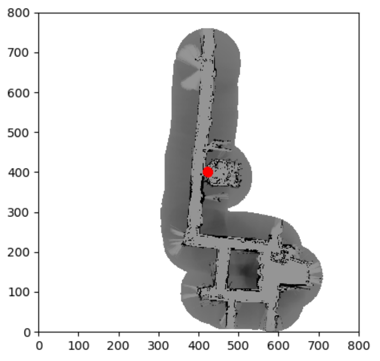
3 Results

As the accompanying videos demonstrate, our implementation worked successfully on the logs 'robotdata1.log' and 'robotdata2.log'.

[1]The step is chosen small enough so that no cell in the occupancy map is skipped along the ray.



(a)



(b)

Figure 4: Plots of the approximate robot's final location (a) For 'robotdata1.log' and (b) For 'robotdata2.log'

Note: to execute our accompanying code, use the script:

```
main.py
```

As the movies show, our particle filter algorithm converges to the correct global position within around few iterations. We depict the particles as red dots. For both the data logs, we used 500 particles. We found that noise parameters $\alpha_1 = .0001, \alpha_2 = .0001, \alpha_3 = .01, \alpha_4 = .01$ work well on both the logs. Additionally, we choose to keep a high weight for the random distribution in the sensor model so as to prevent initial dying of particles which often occurs due to lack of motion and unreliable measurements in the first few timesteps. Once subsequent sensor measurements start coming in, the Gaussian distribution starts dictating the total particle probability, and resampling ensures quick convergence to multiple modes within the first few seconds. These modes too dissolve soon and the particles converge to the right position. We found that anything greater than a 2:1 ratio of random-to-Gaussian weight works well.

The videos are for logs 1 and 2. These have also been submitted with the our code and report.

1. Log 1: https://youtu.be/bpoWfCj5AKc
2. Log 2: https://youtu.be/htfcB3Ue4U0

4 Future Work and Improvements

The limitations of our implementation revolve around the speed of the processing. Ray casting is done in every sensor iteration and this can be easily done once and stored in memory.
Also, the parameters can be better tuned as we sometimes found slight improvements on certain logs with parameters different than what we used.
Another improvement is the experimenting with the level of granularity of the 180 sensor measurements and seeing how the best set of them can be chosen to get the best probability value. Including all of them is noisy and time consuming while sampling them at equal intervals can lose important and reliable information. A better way can be explored.

5 Additional tasks (Extra Credit)

For adaptive resampling, there are three considerations:

1. When to add or subtract number of alive particles?

2. Where to initialize new particles in case of an addition?
3. Which particles to kill in case of a reduction?

The number of particles alive in the map at any time should be a function of how high/low the absolute probability values from the sensor model are. The tricky thing is how to interpret the full set of all particle values to make this decision. Intuitively, we should be adding more particles when the absolute probabilities of the most confident particle has gone below a certain value. This value too should be adaptive in the sense that it needs to come from that particular robot in that particular map. The number of particles can be reduced when all particles have very similar poses and they aren't adding new information over some timesteps. As for the other two questions, we can initialize new particles to areas near the position where the robot was previously localized with high likelihood. Alternatively, in a kidnapped robot setting, the new initializations should be done in random spaces. Similarly, for the case of reduction in number of particles, particles with very similar poses can be killed.

References

[1] Thrun, S., Burgard, W., and Fox, D. *Probabilistic Robotics*, The MIT Press, 2005.