Homework 1: Robot Localization Using Particle Filters

16-833 Robot Localization and Mapping

Austin Windham

Introduction

Localization is a critical aspect of autonomous mobile robots, allowing them to determine their position within a known environment. In this assignment, I implemented the Monte Carlo Localization (MCL) algorithm, which employs a particle filter to estimate the robot's location using sensor data and a pre-defined map. The particle filter operates by sampling possible positions, updating them using a motion model, and refining them through a sensor model. Over time, the particles converge toward the robot's true position as the most probable states are resampled and maintained.

In this report, I will explain my approach to designing and implementing the particle filter, detailing the three key components: the motion model, sensor model, and resampling technique. I will also discuss the results of my implementation, highlighting the performance and challenges encountered.

Motion Model

I used the odometry model in the textbook *Probabilistic Robots* by Thrun, Burgard, and Fox. This model can be seen below and provides a probabilistic approach to account for the uncertainty in a robot's movement based on its previous and current odometry readings. The model assumes that the robot's motion consists of two rotational and one translational component: an initial rotation, followed by a translation, and then a final rotation. Each of these components is subject to noise, which is modeled using Gaussian distributions. The odometry data provides estimates of the robot's position and orientation at two consecutive time steps, and the motion model applies probabilistic noise to these estimates to predict the robot's new state. The four alpha parameters α_1, α_2, α_3, α_4 control the magnitude of the noises added to the rotation and translation. By incorporating this uncertainty into the robot's predicted movement, the model enables particle filters to better account for real-world imperfections in robot motion.

```
1:  **Algorithm sample_motion_model_odometry**($u_t, x_{t-1}$):
2:      $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3:      $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
4:      $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$
5:      $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}}^2 + \alpha_2 \delta_{\text{trans}}^2)$
6:      $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta_{\text{trans}}^2 + \alpha_4\ \delta_{\text{rot1}}^2 + \alpha_4\ \delta_{\text{rot2}}^2)$
7:      $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}}^2 + \alpha_2 \delta_{\text{trans}}^2)$
8:      $x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$
9:      $y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$
10:     $\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$
11:     *return* $x_t = (x', y', \theta')^T$
```

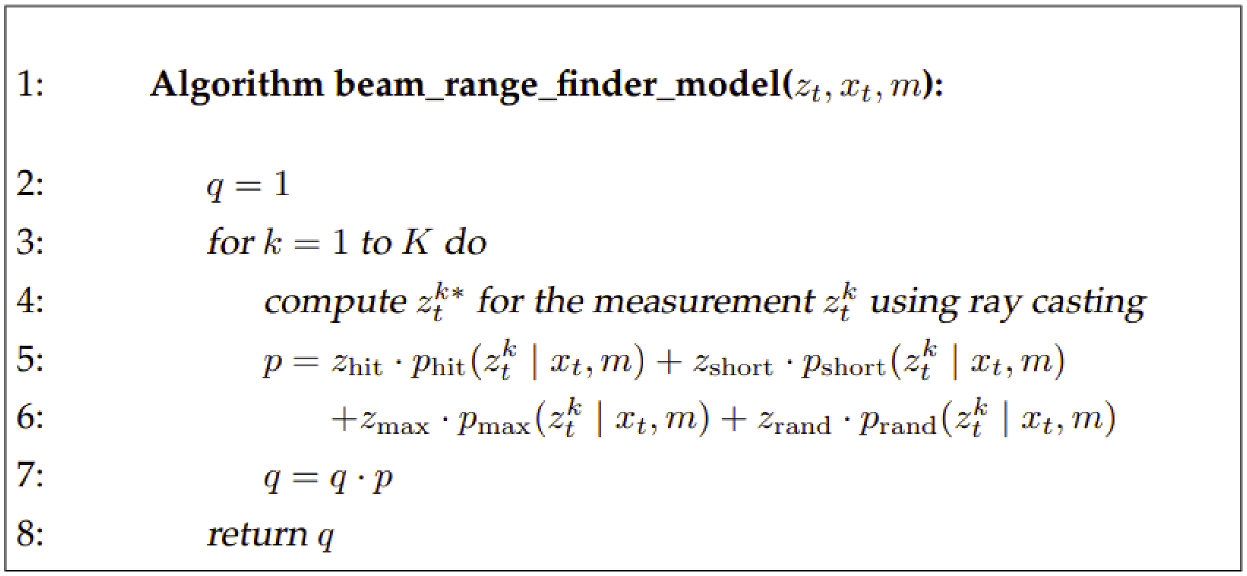
Figure 1: Motion Model

Sensor Model

The sensor model is very important in estimating the likelihood of each particle based on the sensor measurements. It consists of two main components: ray casting and the beam range finder model. Ray casting is used to predict the expected sensor readings for each particle by using the known occupancy map, and the beam range finder model calculates the probability of the measured sensor data given the particle's predicted measurements. Together, these components help refine the particle filter by weighing particles based on how well their predicted sensor readings match the actual observations.

Ray casting is a technique used to predict the expected range measurements for each particle by simulating laser scans based on the robot's current position and orientation. I sent multiple rays in different directions from each particle, simulating how the robot's laser range finder would perceive the environment. Each ray is incrementally extended in small steps from the particle's position until it hits an obstacle or reaches the maximum sensor range. The ray traverses the known occupancy map, where obstacles are identified based on a predefined probability threshold. If the ray hits an obstacle, the distance traveled becomes the predicted measurement for that direction. This process is repeated for all beams, providing a set of expected measurements for each particle. This allows us to simulate how the robot would perceive the environment, which is later compared to the actual sensor readings. I was able to test for ray casting by plotting one particle in the map with its rays to see if the rays reach the particle's surrounding obstacles, indicating that the correct ray distance was being calculated.

The beam range finder model was used to calculate the likelihood of each particle by comparing the actual sensor readings to the expected measurements generated by ray casting. This model can be seen below and was based on the same textbook as before *Probabilistic Robotics*. The model represents the sensor measurement distribution as a mixture of four probabilistic components: p_hit , p_short, p_max , and p_rand. Each component models the corresponding sensor behaviors: accurate readings, unexpected short readings, max-range readings, and random noise. We implemented the beam range finder model by calculating the probabilities for each of these based on the actual sensor reading and the predicted value from ray casting. The final likelihood for each particle is the product of the probabilities across all beams. This ensures that particles with predicted sensor readings closely matching the actual measurements receive higher weights, which improves the accuracy of the particle filter.



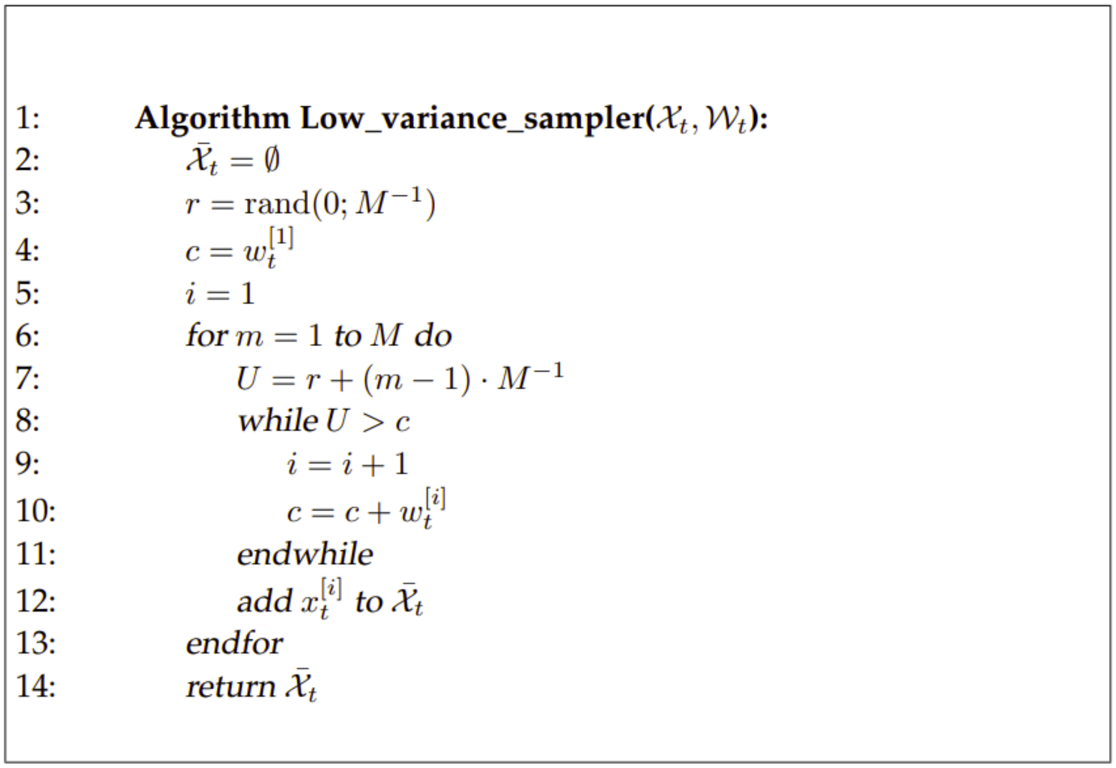
```
1:  Algorithm beam_range_finder_model($z_t, x_t, m$):
2:      $q = 1$
3:      for $k = 1$ to $K$ do
4:          compute $z_t^{k*}$ for the measurement $z_t^k$ using ray casting
5:          $p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$
6:              $+ z_{\text{max}} \cdot p_{\text{max}}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$
7:          $q = q \cdot p$
8:      return $q$
```

Figure 2: Sensor Model

The sensor model was computationally expensive, so I used three methods to improve its efficiency. I precomputed the ray cast values, so they can be directly read from a ray cast table. You will need to use the link at the end of the report to get this table. I also used log-likelihood to sum up the probabilities, and I subsampled once every five degrees for the range finder.

Resampling

I implemented the resampling step of the particle filter using the low-variance resampling method described in *Probabilistic Robotics*. This can be seen in the figure below. This approach helps mitigate the problem of particle depletion, where particles with low weights are gradually eliminated. In low-variance resampling, a single random number is drawn, and particles are systematically selected based on their weights, ensuring that particles with higher weights are more likely to be replicated. This method ensures that the most probable particles are preserved while maintaining diversity among particles with similar weights. By using this technique, I increased the likelihood that particles close to the robot's true position were kept and used in future iterations.



```
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 3: Resampling Method

Parameter Tuning

Tuning was necessary for both the motion model and the sensor model in this implementation. For the motion model, the four alpha parameters needed to be tuned to accurately show the robot's movement in the building. There was a gif of the robot's movement in the first log file that was given with this assignment, so I tuned the parameters with one particle plotted until the particle's movement looked accurate. These were my corresponding alpha values: .0001, .0001, .001, and 0.001.

To fine-tune the sensor model parameters, I simulated expected sensor readings and gradually introduced noise to adjust the values. First, I adjusted the Gaussian distribution coefficient z_hit by setting the measurement data equal to the true laser range at a specific point and finding the value of z_hit that maximized the probability to 1. Next, I tuned the coefficient for random noise z_rand by adding measurement noise, and the coefficient for unexpected short measurements z_short by reducing some of the readings. Finally, I introduced noise to estimate the value for z_max, which handles measurements at the maximum range. After determining these values, we normalized all the parameters and fine-tuned them based on real sensor data to improve the overall performance of the particle filter system. The corresponding parameters I used for z_hit, z_short, z_max, and z rand are 0.5, 0.025, 0.015, and 0.1. The parameters I used for sigma_hit and lambda_short were 100 and 0.1. I kept them as they originally were.

Performance

The particle filter is not perfect, but it is able to correctly localize on the robot most of the time. The link to the particle filter used with 500 particles on log 1 and 2 is here.

https://drive.google.com/drive/folders/1-zmBDazQV7IW6SxhTmfTCy09GJqkTAcZ?usp=sharing

This link also contains the ray cast table, which you will need to run the main.py file, and the first log is at 30 frames per second, and the second is at 50 frames per second. As you can see for both logs, the particles are able to eventually converge on the robot's location. It is tough to get concrete results, but I believe my particle filter works successfully around 80% of the time, so it does not have an extremely high degree of repeatability. The filter has issues in areas where there are many obstacles such as the main room and the bottom of the map. The filter also sometimes mistakes the particles for being in the long hall, which makes sense since the ray cast readings should be somewhat similar.

Future Work and Conclusion

Overall, this was a difficult assignment, but it helped me understand particle filters much better. I had some experience with particle filters from a previous robotics course, but this was more in depth. To improve my particle filter in the future, I think I need a better method for tuning the parameters in the sensor model. The textbook gives a method, but I was unable to implement the method correctly. Sometimes my particles also slightly extend into the obstacles, so it would probably be beneficial to have a way to remove particles that are slightly off the free space. Additionally, a somewhat reinforcement learning approach of randomly doing an exploration particle that is in a different location might help because it might realize that my clusters are not in the correct location. Overall, my particle filter worked well, and this assignment helped me understand particle filters better.

Link to videos and to download ray cast table: https://drive.google.com/drive/folders/1-zmBDazQV7IW6SxhTmfTCy09GJqkTAcZ?usp=sharing