16833-Robot Localization and Mapping-HW1

Parth Nilesh Shah
Shashwat Chavla

2nd October 2024

$$AndrewId - pnshah, shashwac \tag{1}$$

Contents

Motion Model

The motion model in a Particle Filter predicts the next state of each particle based on the current state and control inputs, incorporating randomness to account for uncertainty. This is typically done using a probabilistic approach where noise is added to simulate real-world imperfections in movement. We implemented the odometry motion model as outlined in Table 5.6 on Page 136 of to calculate the posterior distributions of poses based on odometry measurements. The sampling function we used by us was the $np.random.normal$ from Numpy which draws random samples form a normal distribution.

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: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$

4: $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5: $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \mathbf{sample}(\alpha_1 \delta_{\text{rot1}} + \alpha_2 \delta_{\text{trans}})$

6: $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \mathbf{sample}(\alpha_3 \ \delta_{\text{trans}} + \alpha_4 (\delta_{\text{rot1}} + \delta_{\text{rot2}}))$

7: $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \mathbf{sample}(\alpha_1 \delta_{\text{rot2}} + \alpha_2 \delta_{\text{trans}})$

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 Algotithm

In the odometry motion model algorithm, we see 4 tunable parameters – $\alpha_1, \alpha_2, \alpha_3, \alpha_4$. These values model the uncertainity of the robot motion and are crucial for accurately capturing the noise characteristics of the robot's movement. Specifically,

α_1 – relates to the noise in rotation due to rotational motion
α_2 – relates to the noise in rotation due to translational motion
α_3 – relates to the noise in translation due to rotational motion
α_4 – relates to the noise in rotation due to translational motion

Sensor Model

This model evaluates how well each particle's hypothesized state matches the actual sensor measurements, thereby influencing the probability distribution over possible states. The sensor model updates the weigths of particles based on the likelihood of the observed sensor data given each particle's predicted state. The sensor model incoporates probabilistic techniques to account for measurement noise and inaccuracies present in the sensor, as well as the inaccuracy of the map it uses to localize itself. Our use-case involves a laser range finder, and thus our sensor model would compare the predicted distance measurements from each particle's state to the actual distances observed. We implemented the range finder sensor model following Table 6.1 at Page 158 in [1], namely $beammodel$, to replicate the physical model of our sensor. The beam model accounts for four primary types of errors: small measurement noise, which is typically modeled with Gaussian distributions to capture minor inaccuracies in sensor readings; errors due to unexpected objects, where dynamic elements not present in static maps cause deviations, often modeled using exponential distributions; failures to detect objects, leading to maximum range measurements when obstacles are missed, represented by a point-mass distribution; and random unexplained noise, which accounts for sporadic, inexplicable sensor readings using a uniform distribution over the sensor's range. By combining these error models through a mixture of densities, the beam sensor model provides a robust approach to handling the inherent uncertainties and variabilities in range measurements.

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: $\quad + z_{\max} \cdot p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

7: $q = q \cdot p$

8: *return* q

Figure 2: Beam Range-Finder Model

One major change that we incorporated in implementing this model was in $line_7$. Instead of multiplying all the beam probabilities to compute the final probability of a particle, we summed the logarithms of these probabilities. By using the sum of logs, we maintain numerical stability and ensure that the computations remain within a manageable range, especially when dealing with a large number of beams. The equations for computing p_hit, p_short, p_max and p_rand is directly taken from page 125 of [1].

$$p_{\text{hit}}(z_t^k \mid x_t, m) = \begin{cases} \eta\, \mathcal{N}(z_t^k; z_t^{k*}, \sigma_{\text{hit}}^2) & \text{if } 0 \le z_t^k \le z_{\max} \\ 0 & \text{otherwise} \end{cases}$$

$$p_{\text{short}}(z_t^k \mid x_t, m) = \begin{cases} \eta\, \lambda_{\text{short}}\, e^{-\lambda_{\text{short}} z_t^k} & \text{if } 0 \le z_t^k \le z_t^{k*} \\ 0 & \text{otherwise} \end{cases}$$

$$p_{\text{rand}}(z_t^k \mid x_t, m) = \begin{cases} \frac{1}{z_{\max}} & \text{if } 0 \le z_t^k < z_{\max} \\ 0 & \text{otherwise} \end{cases}$$

$$p_{\max}(z_t^k \mid x_t, m) = I(z = z_{\max}) = \begin{cases} 1 & \text{if } z = z_{\max} \\ 0 & \text{otherwise} \end{cases}$$

Figure 3: Measurement Probability Models

The main-tunable paramenters here are z_hit, z_short, z_max, z_rand. These are weights that determine the contribution of each error component to the overall probability calculation. There are 3 more parameters which are related to the probability models – σ_hit and λ_short and $subsampling$.

Resampling Process

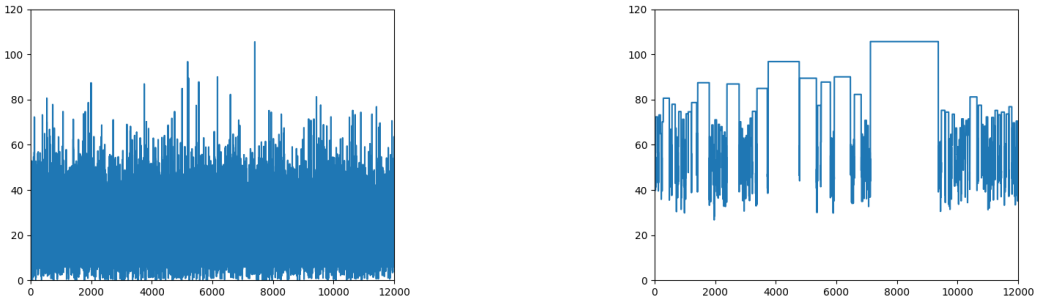
the resampling process is a crucial step that helps maintain a diverse and representative set of particles by focusing computational resources on the most promising hypotheses. As particles evolve over time, some may accumulate higher weights due to better alignment with observed data, while others may have negligible weights and contribute little to the overall state estimation. Resampling addresses this by duplicating high-weight particles and eliminating low-weight ones, effectively redistributing the particle set to better represent the underlying probability distribution.

We implemented the low variance resampling following Table 4.4 at Page 110 in [1] to resample our particles. The low-variance resampling method, specifically, is an efficient technique that reduces the variance introduced during resampling. It works by systematically selecting particles based on their cumulative weights, ensuring that each particle has a chance of being selected proportionally to its weight. This method minimizes randomness in selection and helps maintain a stable and accurate estimate of the state over time.

1: **Algorithm Low_variance_sampler**$(\mathcal{X}_t, \mathcal{W}_t)$**:**
2: $\bar{\mathcal{X}}_t = \emptyset$
3: $r = \text{rand}(0; M^{-1})$
4: $c = w_t^{[1]}$
5: $i = 1$
6: for $m = 1$ to M do
7: $u = r + (m-1) \cdot M^{-1}$
8: while $u > c$
9: $i = i + 1$
10: $c = c + w_t^{[i]}$
11: endwhile
12: add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
13: endfor
14: return $\bar{\mathcal{X}}_t$

Figure 4: Low Variance Resampling

The figure below showcases the plot of probabilities of all particles. We can clearly see that over-time, the resampling function resamples the particle set to include more particles with a higher particle-probability.



(a) 1st Sensor Reading

(b) 10th Sensor Reading

Figure 5: Plot of probability of particles

Parameter Tuning Discussion

We used the logs $robotdata1.log$ and $robotdata5.log$ to verify our implementation and the demonstration video indicates our particle filter can converge and generate correct localization results for both logs.

These are all the tunable parameters and our values in Sensor Model-

_z_hit	_z_short	_z_max	_z_rand	_sigma_hit	_lambda_short	_max_range
11	0	1	800	3	0.01	800

Here is the reasoning behind why chose these values -

- **max_range:** Based on analysis of the data logs and map, most range values were within 800 units, as the majority of beams hit narrow corridor walls. Therefore, we set the `max_range` to 800.
- **z_rand:** The base probability for all particles is determined by p_{rand}, which is $\frac{1}{\texttt{max\_range}}$. To ensure that the sum of logs remains stable and probabilities do not drop below 1 (since $\log(1) = 0$), we set `z_rand` to 800. This ensures that if no beam hits a target, the cumulative probability q remains at 1.
- **sigma_hit:** This parameter represents the standard deviation of the Gaussian distribution for p_{hit}. Observations showed noisy edges with deviations around 3-4 units from true edges, so we set $\sigma_{\text{hit}} = 3$.
- **z_hit:** With $\sigma_{\text{hit}} = 3$, the peak value of the Gaussian p_{hit} is approximately 0.133. To ensure that $z_{\text{hit}} \times p_{\text{hit}}$ exceeds 1, we set `z_hit` to 11.
- **lambda_short and z_short:** The parameter λ_{short} controls the exponential decay for obstacle noise. Given that obstacles were rare in our data logs, primarily limited to controller legs affecting a small number of beams, we set `z_short` to 0, effectively ignoring p_{short}. Attempts to include p_{short} were problematic due to working in map resolution, which flattened the exponential decay curve over large readings. Converting measurements to meters could resolve this but would require retuning other parameters, which was not feasible within our time constraints.

The figure below show's our values for different probabilities over a measurement range of 0 to max_range and the $expected_measurement$ of 400

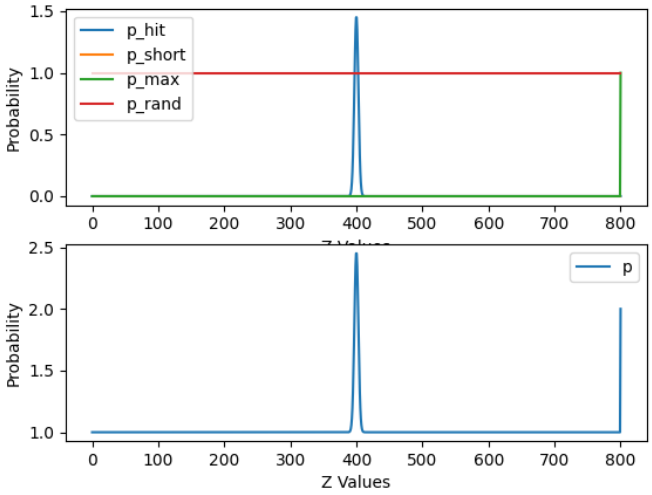


Figure 6: Motion Model Spread

Tuning the motion model parameters, one can adjust how sensitive the model is to different types of motion-induced errors, which in turn influences the accuracy of pose estimation. These are the values we ended up with after some tuning -

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} = \begin{bmatrix} 0.0005 \\ 0.0005 \\ 0.005 \\ 0.005 \end{bmatrix}$$

This is how the particles spread over 160 datalines of logfile $robotdata1.log$. The number of particles is 50 and all are initialized (0, 0). Each color represents a different time-instance.

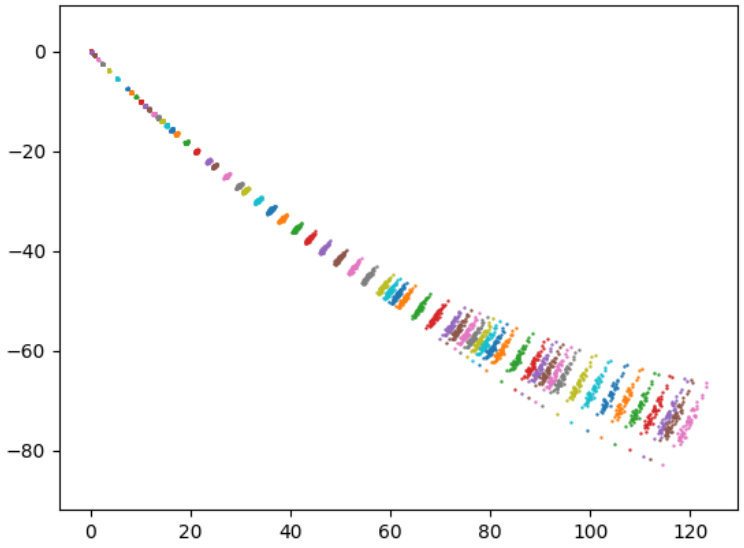


Figure 7: Motion Model Spread

Results

Run on Datalog1 - $https://youtu.be/O6IbpQzpses$

Run on Datalog2 - $https://youtu.be/XHd5tbLGV9w$

Run on Datalog3 - $https://youtu.be/iQq\_KA1hse4$

Kidnapped Robot on Datalog1 - $https://youtu.be/mNL02VW6ZWg$

Performance of our Implementation

Overall Performance
The implementation demonstrates a high level of repeatability, successfully localizing the robot in 8 out of 10 trials for datalogs 1, 2, and 3. The two instances of failure were attributed to the particles not initializing with the appropriate spread, which led to poor localization. To address this, we introduced a fixed random seed value, ensuring consistent initialization across trials. This adjustment improved our success rate to 10 out of 10, highlighting the importance of proper particle distribution in achieving reliable performance.

Number of Particles
Optimal results were achieved using 12,000 particles evenly distributed across the free space. This configuration provided a robust representation of the state space, enhancing localization accuracy. While reducing the number of particles to 5,000 still yielded decent results, the success rate dropped to 5 out of 10 trials. This suggests that while fewer particles can reduce computational load, they may also compromise the filter's ability to accurately capture the robot's pose under varying conditions.

Time Taken
Our implementation is highly efficient due to vectorization and the use of a hashmap for retrieving expected measurement values. Running at a frequency of 43 Hz, it processes datalog1 in approximately 75 seconds, even though the actual duration is 135 seconds. This efficiency ensures that the system can operate in real-time environments without significant delays, making it suitable for dynamic applications where timely updates are crucial. Overall, these performance metrics demonstrate that with careful tuning and optimization, our Particle Filter implementation is both effective and efficient in real-world scenarios.

Vectorization and Look-Up Table

Vectorization was extensively used to improve the performance of both the motion and sensor models. In the motion model, operations like calculating delta changes, applying noise, and updating particle positions were performed on entire arrays of particles simultaneously, rather than iterating through each particle individually. This allows for efficient use of NumPy's optimized array operations. In the sensor model, vectorization was applied to transform particle positions, calculate expected measurements, and compute likelihoods for all particles at once. This approach significantly reduces the computational overhead compared to processing particles one by one.

To further optimize the sensor model, a precomputed look-up table was implemented. This table, called $sensor\_map$, stores pre-calculated expected laser measurements for all possible particle positions and orientations in the map. The $create\_sensor\_map$ function generates this look-up table by ray-casting from each free space cell in the occupancy map for all possible angles. When evaluating particles, instead of performing expensive ray-casting operations in real-time, the algorithm simply fetches the pre-computed measurements from the $sensor\_map$ using the particle's position and orientation as indices. This approach dramatically reduces the computational cost of the sensor model, as it replaces complex geometric calculations with fast array lookups.

Kidnapped Robot Problem

Video Link - https://youtu.be/mNL02VW6ZWg

The implementation for the Kidnapped Robot problem in this code is a clever approach to detect and recover from sudden, unexpected changes in the robot's position. Here's a summary of the solution: The code calculates the mean and maximum of particle weights, then computes a mean-to-max ratio. This ratio serves as an indicator of the particle distribution's health. If this ratio falls below a certain threshold (0.1 in this case), it suggests that the particles have experienced a sudden change in sensor readings, which is interpreted as a potential robot kidnapping event.

When such an event is detected, the system prints a warning message and initiates a recovery procedure. This procedure involves resampling the particles across the entire free space of the environment, effectively resetting the robot's belief about its position. This resampling is done using the $init\_particles\_freespace$ method, which presumably distributes particles uniformly in areas not occupied by obstacles.

This approach allows the system to recover from situations where the robot's actual position has changed dramatically and unexpectedly, such as when it has been physically moved to a new location. By reinitializing the particle distribution, the system gives itself the best chance to re-localize the robot quickly in its new environment.

It's worth noting that this method is particularly useful in scenarios where global localization is necessary, and it provides a robust way to handle significant localization failures or robot transportation events.

Adaptive Number of Particles

All the above runs have adaptive particles, here's a link with the run explicitly stating the number of particles.

Adaptive Particles Datalog1 - $https://youtu.be/O6IbpQzpses$

The implementation of an Adaptive Number of Particles in this code is based on the Effective Sample Size (ESS) concept, drawing inspiration from UC Berkeley material [2]. This approach dynamically adjusts the number of particles based on the variance of particle weights, which serves as a measure of the filter's uncertainty. The code calculates the variance of normalized particle weights and compares it to a predefined threshold ($low\_variance\_threshold$). When the variance is low, indicating high confidence in the current estimate, the number of particles is reduced by a factor ($\_reduction\_factor$) – 5%, but not below a minimum threshold ($min\_particles$) – 1000. Conversely, when the variance is high, suggesting increased uncertainty, the number of particles is increased by a factor ($\_increase\_factor$) – 20%, up to a maximum limit ($max\_particles$) – 10000. This adaptive approach allows the filter to maintain efficiency when the robot's position is well-estimated while increasing robustness during periods of higher uncertainty. The implementation then uses a low variance sampler to resample the new number of particles, ensuring a fair representation of the belief distribution while adapting to the current estimation quality.