16-833: Robot Localization and Mapping, Fall 2024

Homework 1
Robot Localization using Particle Filters

This homework may be submitted in groups of **(max) two** people.

Due: Friday Sep 27, 11:59pm, 2024

Your homework should be submitted as a **typeset PDF file** along with a folder including **code only (no data)**. The PDF and code must be submitted on **Gradescope**. If you have questions, please post them on Piazza or come to office hours. Please do not post solutions or code on Piazza. Discussions are allowed, but each group must write and submit their **own, original** solution. Note that you should list the name and Andrew IDs of each student you have discussed with on the first page of your PDF file. You have a total of 4 late days, use them wisely. As this is a group homework, every late day applies to all members of the group. This is a challenging assignment, ***so please start early***! Good luck and have fun!

1 Overview

The goal of this homework is to become familiar with robot localization using particle filters, also known as Monte Carlo Localization. In particular, you will be implementing a global localization filter for a lost indoor mobile robot (global meaning that you do not know the initial pose of the robot). Your lost robot is operating in Wean Hall with nothing but odometry and a laser rangefinder. Fortunately, you have a map of Wean Hall and a deep understanding of particle filtering to help it localize.

As you saw in class, particle filters are non-parametric variants of the recursive Bayes filter with a resampling step. The Prediction Step of the Bayes filter involves sampling particles from a proposal distribution, while the Correction Step computes importance weights for each particle as the ratio of target and proposal distributions. The Resampling Step resamples particles with probabilities proportional to their importance weights.

When applying particle filters for robot localization, each particle represents a robot pose hypothesis which for a 2D localization case includes the (x, y) position and orientation θ of the robot. The Prediction and Correction Steps are derived from robot motion and sensor models respectively. This can be summarized as an iterative process involving three major steps:

1. Prediction Step: Updating particle poses by sampling particles from the **motion model**, that is $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$. The proposal distribution here is the motion model, $p(x_t|u_t, x_{t-1})$.

2. Correction Step: Computing an importance weight $w_t^{[m]}$ for each particle as the ratio of target and proposal distributions. This reduces to computing weights using the **sensor model**, that is $w_t^{[m]} = p(z_t|x_t^{[m]}, \mathcal{M})$.

3. Resampling Step: Resampling particles for the next time step with probabilities proportial to their importance weights.

Here, m is the particle index, t is the current time step, and $\mathcal{M}$ is the occupancy map. $x_t^{[m]}$ is the robot pose of particle m at time t, and $w_t^{[m]}$ is the importance weight for particle m at time t.

2 Monte Carlo Localization

Monte Carlo Localization (MCL), a popular localization algorithm, is essentially the application of particle filtering for mobile robot localization. You can refer to **Section 4.3** of [1] for details on the MCL algorithm. Algorithm 1, taken from [1], describes the particle filter algorithm applied for robot localization.

Algorithm 1 Particle Filter for Robot Localization

$\bar{\mathcal{X}}_t = \mathcal{X}_t = \phi$
for $m = 1$ to M **do**
  sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$ ▷ Motion model
  $w_t^{[m]} = p(z_t \mid x_t^{[m]}, \mathcal{M})$ ▷ Sensor model
  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
end for
for $m = 1$ to M **do**
  draw i with probability $\propto w_t^{[i]}$ ▷ Resampling
  add $x_t^{[i]}$ to $\mathcal{X}_t$
end for
return $\mathcal{X}_t$

As you can see, the MCL algorithm requires knowledge of the robot motion and sensor models, and also of the resampling process to be used. You will need to implement these three components of the algorithm. We briefly describe these three components and point you to resources with more details and pseudo-codes.

Motion Model

The motion model $p(x_t|u_t, x_{t-1})$ is needed as part of the prediction step for updating particle poses from the previous time step using odometry readings. **Chapter 5** of [1] details two types of motion models, the Odometry Motion Model and the Velocity Motion Model. You can use either model for sampling particles according to $x_t^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$. The Odometry Motion Model might be more straightforward to implement since it uses odometry measurements directly as a basis for computing posteriors over the robot poses.

Sensor Model

The sensor model $p(z_t|x_t, m)$ is needed as part of the correction step for computing importance weights (proportional to observation likelihood) for each particle. Since

the robot is equipped with a laser range finder sensor, we'll be using a beam measurement model of range finders. **Section 6.3** of [1] details this beam measurement model $p(z_t|x_t, m)$ as a mixture of four probability densities, each modeling a different type of measurement error. You'll have to play around with parameters for these densities based on the sensor data logs that you have. You are also free to go beyond a mixture of these four probability densities and use a measurement model that you think describes the observed laser scans better.

Additionally, as part of this beam measurement model, you'll be performing ray-casting on the occupancy map so as to compute true range readings z_t^{k*} from individual particle positions (shifted to laser position).

Hint: The book specifies that the sensor model needs to be a normalized probability distribution, however, we have found in practice it is easier to debug when the mixture weights (and thus the distribution) are unnormalized as the particle weights are later normalized.

Resampling

As part of the resampling process, particles for the next time step are drawn based on their weights in the current time step. A straightforward resampling procedure would be sampling from a multinomial distribution constructed using importance weights of all particles. However, repetitive resampling using such a technique may cause the variance of the particle set (as an estimator of the true belief) to increase.

One strategy for reducing the variance in particle filtering is using a resampling process known as *low variance sampling*. Another strategy is to reduce the frequency at which resampling takes place. Refer to the Resampling subsection under **Section 4.3.4** of [1] for more details on variance reduction and using low variance resampling for particle filters.

3 Implementation

Resources

You may use any programming language for implementation. There is no requirement that your implementation run in real-time, although it is advisable to use something faster. Feel free to utilize the techniques that we have discussed in class as well as extensions discussed in [1] or elsewhere. You will be performing global localization for a lost indoor mobile robot in Wean Hall given a map, odometry readings and laser scans. The data directory that you received with this handout (courtesy of Mike Montemerlo) has the following files:

- `data/readme.txt` – Format description for the map and the data logs.
- `data/log/robotdataN.log` – Five data logs (odometry and laser data).
- `data/map/wean.dat` – Map of Wean Hall to use for localization.
- `data/map/bee-map.c` – Example map reader in C from BeeSoft that you may use. Note we also provide a Python map reader.
- `assets/wean.gif` – Image of map (for reference).
- `assets/robotmovie1.gif` – Animation of data log 1 (for reference).

We have also provided you with helper code (in Python) that reads in the occupancy map, parses robot sensor logs and implements the outer loop of the particle filter algorithm illustrated in Algorithm 1. The motion model, sensor model, and resampling implementations are left as an exercise for you.

- `main.py` – Parses sensor logs (`robotdata1.log`) and implements outer loop of the particle filter algorithm shown in Algorithm 1. Relies on SensorModel, MotionModel and Resampling classes for returning appropriate values.
- `map_reader.py` – Reads in the Wean Hall map (`wean.dat`) and computes and displays corresponding occupancy grid map.
- `motion_model.py, sensor_model.py, resampling.py` - Provides class interfaces for expected input/output arguments. Implementation of corresponding algorithms are left as an exercise for you.

You are free to use the helper code directly or purely for reference purposes. To utilize the framework, please start with a Python 3 enviroment. We recommend creating a virtual enviroment using *e.g.* `conda`, and `pip install -r requirements.txt` (located in the `code` directory) for the basic dependencies. A short tutorial for creating a virtual enviroment can be found at here.

Improving Efficiency

Although there is no requirement that your code run in real-time, the faster your code, the more particles you will be able to use feasibly and the faster your parameter tuning iterations will go. You'll most probably have to apply some implementation 'hacks' to improve performance, for instance,

- Intializing particles in completely unoccupied areas instead of uniformly everywhere on the map.
- Subsampling the laser scans to say, every 5 degrees, instead of considering all 180 range measurements.
- When computing importance weights based on the sensor model, be cognizant of numerical stability issues that may arise when multiplying together likelihood probabilities of all range measurements within a scan. You might want to numerically scale the weights or replace the multiplication of likelihood probabilities with a summation of log likelihoods.
- Since motion model and sensor model computations are independent for all particles, parallelizing your code would make it much faster.
- You'll observe that operations like ray-casting are one of the most computationally expensive operations. Think of approaches to make this faster, for instance using coarser discrete sampling along the ray or possibly even precomputing a look-up table for the raycasts.
- Lastly, if you use Python, apply vectorization as much as possible; if you're comfortable with C++, consider using the OpenMP backend (which is a one-liner) to accelerate.

Debugging

For easier debugging, ensure that you visualize and test individual modules like the motion model, sensor model or the resampling separately. Some ideas for doing that are:

- Test your motion model separately by using a single particle and plotting its trajectory on the occupancy map. The odometry will cause the particle position to drift over time globally, but locally the motion should still make sense when comparing with given animation of datalog 1 (`robotmovie1.gif`).
- Cross-check your sensor model mixture probability distribution by plotting the $p(z_t|z_t^*)$ graph for some set of values of z_t^*.
- Test your ray-casting algorithm outputs by drawing robot position, laser scan ranges and the ray casting outputs on the occupancy map for multiple time steps.

4 What to turn in

You should generate a visualization (video) of your robot localizing on `robotdata1.log` and another log of your choice. Don't worry—your implementation may not work **all** the time—but should perform most of the time for a reasonable number of particles. Hyperlinks to the videos must be in the report—we prefer unlisted Youtube videos or Google Drive links. Please ensure proper viewing permissions are enabled before sharing the links. Please speed-up videos to ensure each log is under 2 minutes, and mention the speed multiplier in the video or report. **The report must describe your approach, implementation, description of performance, robustness, repeatability, and results.** Make sure you describe your motion and sensor models (including your ray casting process), your resampling procedure, as well as the parameters you had to tune (and their values). Include some future work/improvement ideas in your report as well. Turn in your report and code on **Gradescope** by the due date. Do not upload the `data/` folder or any other data. Only one group member needs to submit, and should list all group members on the title page as well as via Gradescope (see instructions here).

Score breakdown

- (10 points) Motion Model: implementation correctness, description
- (20 points) Sensor Model: implementation correctness, description
- (10 points) Resampling Process: implementation correctness, description
- (10 points) Discussion of parameter tuning
- (30 points) Performance of your implementation
- (10 points) Discussion of performance and future work
- (10 points) Write-up quality, video quality, readability
- (Extra Credit: 10 points) Kidnapped robot problem

- (Extra Credit: 10 points) Adaptive number of particles
- (Extra Credit: 5 points) Vectorized Python or fast C++ implementation

5 Extra credit

Focus on getting your particle filter to work well before attacking the extra credit. Points will be given for an implementation of the kidnapped robot problem and adaptive number of particles. Please answer the corresponding questions below in your write up.

i. **Kidnapped robot problem:** The kidnapped robot problem commonly refers to a situation where an autonomous robot in operation is carried to an arbitrary location. You can simulate such a situation by either fusing two of the log files or removing a chunk of readings from one log. How would your localization algorithm deal with the uncertainity created in a kidnapped robot problem scenario? Can you make improvements to your algorithm to better address this problem?

ii. **Adaptive number of particles:** Can you think of a method that is more efficient to run, based on reducing the number of particles over timesteps? Describe the metric you use for choosing the number of particles at any time step.

You will also receive bonus credits provided your implementation is optimized, either with vectorization in Python or acceleration in C++.

6 Advice

The performance of your algorithm is dependent on (i) parameter tuning and (ii) number of particles. While increasing the number of particles gives you better performance, it also leads to increased computational time. An ideal implementation has a reasonable number of particles, while also not being terribly slow. Consider these factors while deciding your language of choice—e.g. choosing between a faster implementation in C++ or vectorized optimization in Python vs. using the raw Python skeleton code.

References

[1] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics*. MIT press, 2005.

7 Motion Model Implementation and Analysis

7.1 Implementation Overview

The motion model is the pivotal component of a particle filter that predicts the next state of each particle based on the previous state and control inputs. A robust motion model accounts for the noise and uncertainities, that are the charactersitcs in the robot's motion.

The design choice I personally made to go towards implementing Odometry Motion model for this particle filter (as described in section 5.3 of Thrun et al.'s "Probabilistic Robotcs") against the Velocity Motion model (another feasible option mentioned in the book) was primarily driven by the following factors:

1. **Modality of data in logs:** The log files provided contained direct odometry measurements complemented by the range finder data. The odoemtry data provides access to direct positional changes between time steps, which aligns with the Odometry Motion Model's input, while in contrast the Velocity Model would require deriving velocity and angular velocity from the provided odometry data with potential compounding error. Additionally velocity estimation at low speeds or quick direction changes (such as observed by quick rotational movements in log3) would have been challenging.

2. **Computational Efficiency:** The computational steps for the Odometry motion model, as described by Thrun et al, were less and required less error-handling as compared to v/ω that could occur in the Velocity Model when the angular velocity in very small (straightline).

3. **Error Modeling**: Odometry motion model does error modeling in terms of rotation and translation, which aligns with the available odometry data.

4. **Handling non-holonomic constraints:** The utility of change in position and orientation by the Odometry model implicitly incorporates non-holonomic constraints which is important for our use-case of a wheeled robot.

5. **Discrete Time Steps:** The discretized processing of data aligns with the Odometry's model which represents motion between two distinct poses.

7.2 Mathematical Formulation

The Odometry Motion Model predicts the new state $\mathbf{x}_t$ of the robot (or particle) based on its previous state $\mathbf{x}_{t-1}$ and the odometry measurements $\mathbf{u}_{t-1}$ and $\mathbf{u}_t$ at times $t-1$ and t, respectively. This model, as described in Thrun et al. [1], uses odometry data to estimate motion relative to the current particle frames and incorporates Gaussian noise to account for inaccuracies in odometry sensing.

7.2.1 State Representation

- **Robot State $\mathbf{x}$**: A vector $[x, y, \theta]$ representing the robot's position (x, y) and orientation θ.

- **Odometry Measurements $\mathbf{u}$**: A vector $[\overline{x}, \overline{y}, \overline{\theta}]$ representing the robot's perceived position and orientation based on odometry readings.

7.2.2 Motion Components

The model decomposes the motion into three components:

1. **Initial Rotation ($\delta_{\mathbf{rot}_1}$)**: The rotation required to align the robot's orientation from its previous pose to the direction of the new position.
2. **Translation ($\delta_{\mathbf{trans}}$)**: The distance moved from the previous position to the new position.
3. **Final Rotation ($\delta_{\mathbf{rot}_2}$)**: The rotation required to align the robot's orientation from the direction of translation to its new orientation.

7.2.3 Noise Modeling

To account for uncertainties in motion, noise is added to each motion component. The noise is modeled as zero-mean Gaussian distributions with variances proportional to the squares of the motion components:

- **Variance of Rotational Components**:

$$\sigma^2_{\delta_{\text{rot}}} = \alpha_1 \delta^2_{\text{rot}} + \alpha_2 \delta^2_{\text{trans}}$$

- **Variance of Translational Component**:

$$\sigma^2_{\delta_{\text{trans}}} = \alpha_3 \delta^2_{\text{trans}} + \alpha_4 (\delta^2_{\text{rot}_1} + \delta^2_{\text{rot}_2})$$

The parameters $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ are tunable constants representing the level of noise in the robot's motion.

7.2.4 Mathematical Steps

1. **Compute Odometry-Based Motion Components**:
   - Initial Rotation: $\delta_{\text{rot}_1} = \text{atan2}(\overline{y}' - \overline{y}, \overline{x}' - \overline{x}) - \overline{\theta}$
   - Translation: $\delta_{\text{trans}} = \sqrt{(\overline{x}' - \overline{x})^2 + (\overline{y}' - \overline{y})^2}$
   - Final Rotation: $\delta_{\text{rot}_2} = \overline{\theta}' - \overline{\theta} - \delta_{\text{rot}_1}$
2. **Add Noise to Motion Components**: Sample noise for each component:

$$\hat{\delta}_{\text{rot}_1} = \delta_{\text{rot}_1} - \text{sample}(\sigma_{\delta_{\text{rot}_1}})$$
$$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \text{sample}(\sigma_{\delta_{\text{trans}}})$$
$$\hat{\delta}_{\text{rot}_2} = \delta_{\text{rot}_2} - \text{sample}(\sigma_{\delta_{\text{rot}_2}})$$

   where $\text{sample}(\sigma)$ draws from a zero-mean Gaussian with standard deviation σ.
3. **Compute the New State**:
   - Update Position:

$$x' = x + \hat{\delta}_{\text{trans}} \cdot \cos(\theta + \hat{\delta}_{\text{rot}_1})$$
$$y' = y + \hat{\delta}_{\text{trans}} \cdot \sin(\theta + \hat{\delta}_{\text{rot}_1})$$

- Update Orientation:

$$\theta' = \theta + \hat{\delta}_{\text{rot}_1} + \hat{\delta}_{\text{rot}_2}$$

4. **Normalize the Angle**: Ensure θ' remains within $[-\pi, \pi]$:

$$\theta' = (\theta' + \pi) \mod 2\pi - \pi$$

7.3 Algorithm

The following motion model follows the Odometry Moton Model algorithm as outlined by Thrun et al.:

Algorithm 2 Sample Motion Model Odometry

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

7.4 Practical Implementation Details

In translating the above algorithm to Python, several considerations and a few modifications were necessary:

1. **Standard Deviation vs. Variance**: As it can be noted in section 7.3, lines 5-7 of the algorithm, the sampling step utilizes variance. However, Numpy's np.random.normal() expects standard deviation as an input hence we calculate the standarddeviation of the calculate variance before passing it to np.random.normal():

```
noise_rot1 = np.sqrt(self._alpha1 * rot1**2 + self._alpha2 * trans**2)
noise_trans = np.sqrt(self._alpha3 * trans**2 + self._alpha4
    * (rot1**2 + rot2**2))
noise_rot2 = np.sqrt(self._alpha1 * rot2**2 + self._alpha2 * trans**2)

rot1_with_noise = rot1 - np.random.normal(0, noise_rot1)
trans_with_noise = trans - np.random.normal(0, noise_trans)
rot2_with_noise = rot2 - np.random.normal(0, noise_rot2)
```

2. **Angle Normalization**: During testing of the models on custom motion cases (not on the logs) it was observed that the angle overflowed outside the bounds, and for that a normalization function was implemented to ensure that the angles remain within the range $[-\pi, \pi]$:

```
def normalize_angle(self, angle):
    return (angle + np.pi) % (2 * np.pi) - np.pi
```

7.5 Implementation Correctness

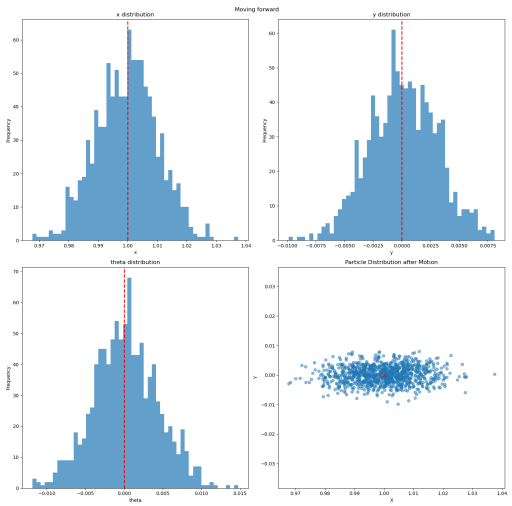
To verify the correctness and robustness of the implemented motion model, extensive testing was done using both synthetic and real-world data. These scripts used for testing were written from scratch and are discussed below, along with submitted as part of the code submission (_test_motion_model.py, _test_motion_model_cont.py, and test_motion_model_traj.py).

7.5.1 Unit Tests with Synthetic Data

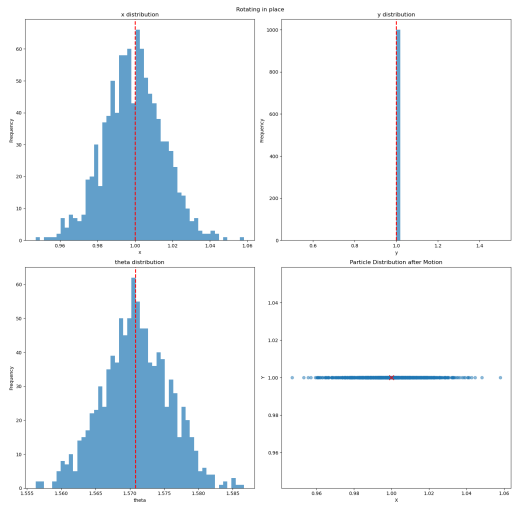
Custom test scripts were designed to evaluate the motion model under controlled scenarios. These tests help ensure the model behaves as expected when subjected to known motions. For each case, the model was run 1000 times to gather statistical data, keeping the induced noise low.

Table 1: Motion Model Test Cases

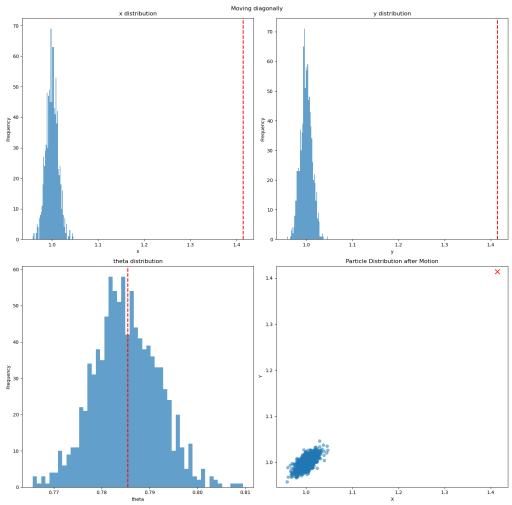
Test Case	**Desc.**	**Initial State**	**Odometry**	**Expected Output**	**Results**
Moving Forward	Robot moves along x-axis without rotation	$[0, 0, 0]$	$u_{t-1} = [0, 0, 0]$ $u_t = [1, 0, 0]$	$[1, 0, 0]$	Mean: $[1.00009, 0.00014, 0.00008]$ StdDev: $[0.01018, 0.00319, 0.00459]$ AbsError: $[0.00009, 0.00014, 0.00008]$
Rotating in Place	Robot rotates 90° without translation	$[1, 1, 0]$	$u_{t-1} = [0, 0, 0]$ $u_t = [0, 0, \frac{\pi}{2}]$	$[1, 1, \frac{\pi}{2}]$	Mean: $[1.00057, 1.0, 1.57084]$ StdDev: $[0.01582, 0.0, 0.00501]$ AbsError: $[0.00057, 0.0, 0.00004]$
Moving Diagonally	Robot moves diagonally with 45° heading	$[0, 0, 0]$	$u_{t-1} = [0, 0, 0]$ $u_t = [1, 1, \frac{\pi}{4}]$	$[1.41421, 1.41421, \frac{\pi}{4}]$	Mean: $[1.00038, 1.00044, 0.78572]$ StdDev: $[0.01226, 0.01268, 0.00695]$ AbsError: $[0.41383, 0.41377, 0.00032]$
Complex Motion	Robot moves and rotates in complex pattern	$[0, 0, 0]$	$u_{t-1} = [1, 1, \frac{\pi}{4}]$ $u_t = [2, 3, \frac{\pi}{2}]$	$[2, 3, \frac{\pi}{2}]$	Mean: $[2.12099, 0.70775, 0.78566]$ StdDev: $[0.02235, 0.01718, 0.01059]$ AbsError: $[0.12099, 2.29225, 0.78514]$



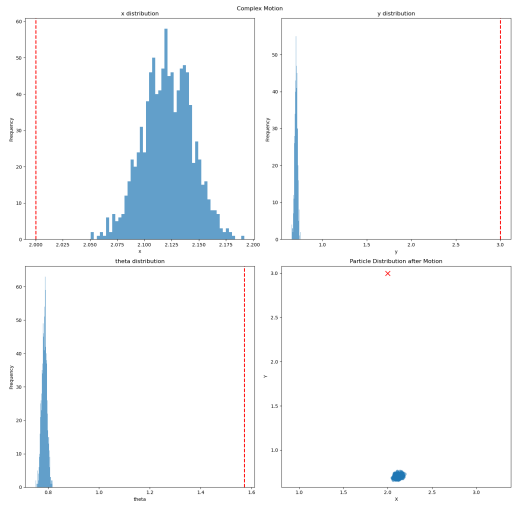
(a) Moving Forward



(b) Rotating in Place



(c) Moving Diagonally



(d) Complex Motion

Figure 1: Particle Distribution for Different Test Cases

Analysis of Test Results

The motion model's performance, as evidenced by Table 1 and Figure 1, exhibits varying degrees of accuracy and precision across different motion patterns:

1. **Translational Accuracy:**
   - Simple translation (Moving Forward) shows high accuracy with mean absolute error $< 10^{-4}$ m.
   - Complex motions (Diagonal, Complex) display larger positional errors, with mean absolute errors of 0.414 m and 2.292 m respectively.
   - Consistent underestimation of distance in diagonal movement (29.3% error) suggests a systematic bias in the model.
2. **Rotational Precision:**
   - Pure rotation (Rotating in Place) demonstrates high accuracy with angular error $< 10^{-4}$ radians.
   - Complex motions maintain reasonable orientation estimates (errors $<$ 0.785 radians) despite larger positional errors.

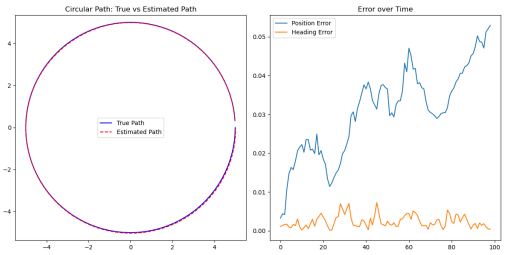
Inferring from the results above it can be concluded that the implementation of the Odometry Motion Model is correct. Additionally for our data from a differential drive (behavior inferred from the gif resource provided) we need not worry about high positional errors and rotational errors for more complex motions since a differential drive robot is not capable of executing such motions.

7.5.2 Continuous Path Tests

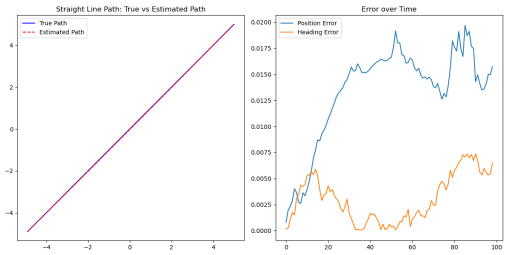
To assess the motion model's performance over extended motions, we simulated various robot paths and applied the motion model iteratively.

Table 2: Continuous Path Test Results

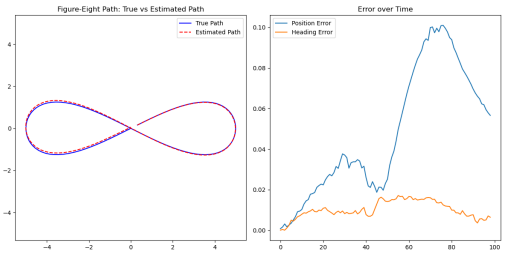
Path Type	Description	Avg. Pos. Error (units)	Max. Pos. Error (units)	Avg. Heading Error (rad)	Max. Heading Error (rad)
Circular	Radius 5 units, centered at (0,0)	0.0837	0.1347	0.0070	0.0163
Straight Line	From (-5,-5) to (5,5)	0.0264	0.0550	0.0038	0.0066
Figure-Eight	Centered at (0,0)	0.0349	0.0632	0.0082	0.0195
Square	Side length 10 units, centered at (0,0)	0.1303	0.2569	1.4653	6.2831



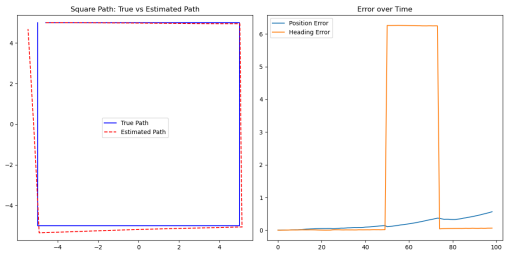
(a) Circular continuous trajectory



(b) Continuous diagonal straight line



(c) Figure-eight continuous trajectory



(d) Square continuous trajectory

Figure 2: Continuous Path Test Results

Analysis of Continuous Path Tests: The motion model's performance, as evidenced by Table 2 and Figure 2, exhibits varying degrees of accuracy and precision across different motion patterns:

1. **Path Complexity vs. Error Correlation:**

- Linear relationship observed between path complexity and error magnitude.
- Straight line path: lowest errors (Avg. Pos. Error: 0.0264 units).
- Square path: highest errors (Avg. Pos. Error: 0.1303 units, Max. Heading Error: 6.2831 rad).

2. **Positional Accuracy:**

- Consistent performance in smooth trajectories (circular, figure-eight).

- Circular path: Avg. Pos. Error 3 times higher than straight line (0.0837 vs 0.0264 units) but still low.
- Figure-eight: Minimal increase in error despite increased complexity (Avg. Pos. Error: 0.0349 units).

3. **Orientation Estimation:**

- High accuracy for smooth, continuous rotations (circular path: Avg. Heading Error 0.0070 rad).
- Significant degradation for discontinuous orientation changes (square path: Avg. Heading Error 1.4653 rad).
- Maximum heading error of 2π in square path indicates potential angle wrap-around issues.

These results indicate that the motion model performs robustly for smooth, continuous paths but struggles with abrupt changes in direction. The significant discrepancy in performance between the square path and other trajectories suggests that refinement of the model's handling of angle normalization could yield substantial improvements in overall accuracy and reliability.

7.5.3 Trajectory Test with Real Data

We evaluated the motion model using actual odometry data from the log file 'robotdata1.log' by spawning a single particle in the map.

Procedure

1. **Data Extraction:** Parsed odometry readings from the log file.
2. **Initialization:** Set initial particle state $\mathbf{x}_{t-1}$ to match the first odometry reading.
3. **Iterative Update:** Applied motion model to compute $\mathbf{x}_t$ for each subsequent odometry reading $\mathbf{u}_t$.
4. **Visualization:** Plotted estimated trajectory on Wean Hall's occupancy map.

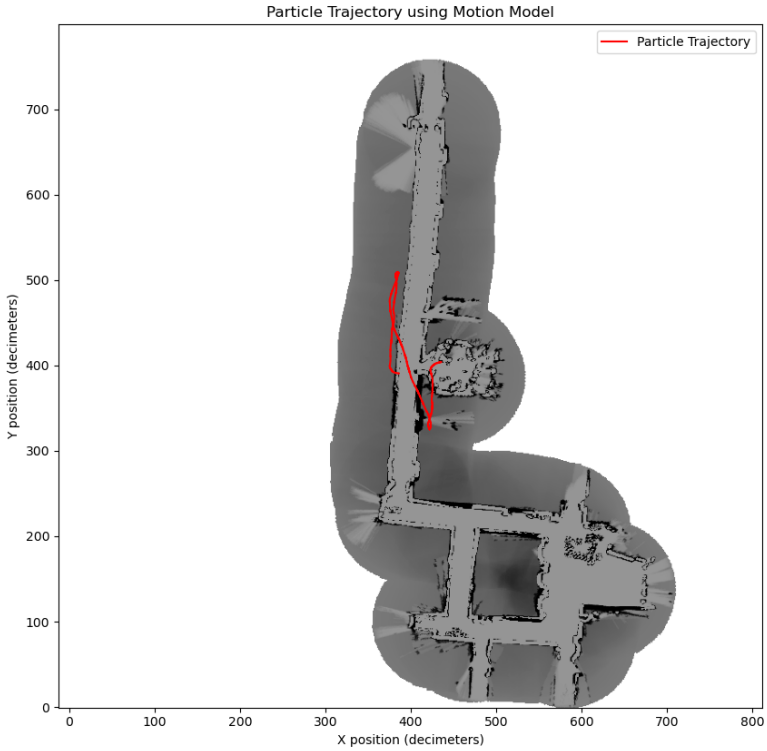


Figure 3: Estimated Trajectory on Wean Hall Occupancy Map

Analysis of Results

1. Estimated trajectory adheres to feasible paths within the mapped environment and the trajectory shape aligns with expected robot movements based on the gif for the provided as a resource log file.

2. Drift is observed in the estimated state trajectory over time as expected which is expected

3. Smooth trajectory suggests well-behaved uncertainty propagation in the motion model.

In conclusion, the implemented Motion Model demonstrates good performance across various test scenarios, effectively capturing both the expected motion and associated uncertainties. Its ability to handle real-world data suggests it is suitable for use in the broader particle filter implementation.

8 Sensor Model Implementation and Analysis

8.1 Introduction

In probabilistic robotics, the sensor model is essential for updating the belief about the robot's state based on sensor measurements. It calculates the likelihood of observing a particular measurement given the robot's pose and the map of the environment. In the context of particle filters, the sensor model assigns weights to particles, reflecting how well each particle's state explains the observed sensor data. An accurate sensor model improves the filter's ability to converge to the true pose of the robot by favoring particles that are more consistent with the sensor readings.

As directed in the writeup I implemented the Beam Range Finder Model as described in Chapter 6 (section 6.3) of Probabilistic Robotics by Thrun et al. (2005) as it accounts for various types of measurement uncertainties and errors inherent in real-world laser ranger finder sensors.

8.2 Implementation Overview

The sensor model we chose consists of four probabilistic components that collectively model the behavior of a laser range finder:

1. $p_{\mathbf{hit}}$: Models the probability of obtaining the expected measurement with Gaussian noise.
2. $p_{\mathbf{short}}$: Accounts for unexpected obstacles causing measurements shorter than expected.
3. $p_{\mathbf{max}}$: Represents the probability of receiving a maximum range reading when the laser does not detect any obstacle.
4. $p_{\mathbf{rand}}$: Captures random measurements due to noise or unmodeled phenomena.

These components are combined to compute the likelihood of a measurement given the robot's pose and the map. The model considers the inherent noise in the sensor.

8.3 Mathematical Formulation

The sensor model computes the probability $p(z_t \mid x_t, m)$ of receiving a laser scan z_t given the robot's pose x_t and the map m. The Beam Range Finder Model combines the four components as follows:

$$\begin{aligned} p(z_t \mid x_t, m) = \big(&w_{\text{hit}} \cdot p_{\text{hit}}(z_t \mid x_t, m) + w_{\text{short}} \cdot p_{\text{short}}(z_t \mid x_t, m) \\ &+ w_{\text{max}} \cdot p_{\text{max}}(z_t \mid x_t, m) + w_{\text{rand}} \cdot p_{\text{rand}}(z_t \mid x_t, m)\big) \end{aligned} \tag{1}$$

where:

- w_{hit}, w_{short}, w_{max}, w_{rand} are weights that sum to 1. However, following the directives provided in the writeup, unnormalized mixture weights and distributions are used as the particle weights are later normalized.

8.3.1 Components of the Sensor Model

1. **Probability of a Hit (p_{hit})**

   Models accurate measurements with Gaussian noise centered at the expected range z_t^* obtained from ray casting.

$$p_{\text{hit}}(z_t \mid x_t, m) = \begin{cases} \frac{1}{\sqrt{2\pi}\sigma_{\text{hit}}} \exp\left(-\frac{(z_t - z_t^*)^2}{2\sigma_{\text{hit}}^2}\right), & \text{if } 0 \le z_t \le z_{\max} \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

   - σ_{hit} is the standard deviation of the measurement noise.

2. **Probability of Short Reading (p_{short})**

   Accounts for measurements shorter than expected due to unexpected obstacles (e.g., people or objects not in the map).

$$p_{\text{short}}(z_t \mid x_t, m) = \begin{cases} \frac{\lambda_{\text{short}} \exp(-\lambda_{\text{short}} z_t)}{1 - \exp(-\lambda_{\text{short}} z_t^*)}, & \text{if } 0 \le z_t \le z_t^* \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

   - λ_{short} is the rate parameter of the exponential distribution.

3. **Probability of Maximum Range Reading (p_{max})**

   Models the probability of receiving a maximum range measurement when the laser does not detect any obstacle.

$$p_{\max}(z_t \mid x_t, m) = \begin{cases} 1, & \text{if } z_t = z_{\max} \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

4. **Probability of Random Measurement (p_{rand})**

   Accounts for random measurements due to noise or errors, modeled as a uniform distribution.

$$p_{\text{rand}}(z_t \mid x_t, m) = \begin{cases} \frac{1}{z_{\max}}, & \text{if } 0 \le z_t < z_{\max} \\ 0, & \text{otherwise} \end{cases} \tag{5}$$

8.4 Ray Casting Algorithm

To obtain the expected measurements z_t^* necessary for p_{hit} and p_{short}, we perform ray casting from the robot's pose into the occupancy map. Ray casting simulates the path of a laser beam and determines where it first intersects an obstacle.

Algorithm 3 Ray Casting for Beam i

function RayCastingBeam(x_t, i, m)
    Inputs: Robot pose $x_t = (x, y, \theta)$, beam index i, map m
    Outputs: Expected range measurement z_t^*

    $d \leftarrow 25$ cm ▷ Laser sensor offset from robot center
    $\phi_i \leftarrow \left(\frac{i - 90}{180}\right) \pi$ ▷ Beam angle relative to robot orientation
    $x_{\text{laser}} \leftarrow x + d \cdot \cos(\theta)$
    $y_{\text{laser}} \leftarrow y + d \cdot \sin(\theta)$
    $\theta_i \leftarrow \theta + \phi_i$ ▷ Absolute beam angle
    $\mathbf{v}_i \leftarrow (\cos(\theta_i), \sin(\theta_i))$ ▷ Direction vector
    $\mathbf{p} \leftarrow (x_{\text{laser}}, y_{\text{laser}})$ ▷ Starting point
    $r \leftarrow 0$ ▷ Initialize range
    $\Delta r \leftarrow 1$ cm ▷ Step size along the ray
    while $r < z_{\max}$ **do**
        $r \leftarrow r + \Delta r$
        $\mathbf{p} \leftarrow \mathbf{p} + \Delta r \cdot \mathbf{v}_i$ ▷ Move along the ray
        if not IsValid(i_x, i_y, m) **then**
            $z_t^* \leftarrow z_{\max}$
            break
        end if
        if $m[i_y, i_x] >$ OccupancyThreshold **then**
            $z_t^* \leftarrow r$
            break
        end if
    end while
    return z_t^*
end function

8.4.1 Algorithm for Ray Casting

Explanation:

- Line 5-9: Calculate the laser sensor position and beam angle, accounting for the sensor offset.
- Line 10-11: Initialize the starting point and range.
- Line 12-23: Iteratively move along the ray until an obstacle is encountered or the maximum range is reached.
- Line 17-19: Check if the point is outside the map boundaries.
- Line 20-22: Check if the map cell is occupied (above occupancy threshold).
- Line 24: Return the expected range z_t^*.

8.5 Sensor Model Algorithm

Explanation:

- Line 4: Initialize the total log probability to zero.
- Line 5-30: For each laser beam:
  - Line 6-7: Retrieve the measured range and compute the expected range using ray casting.
  - Line 9-13: Compute p_{hit} if the measured range is within valid bounds.
  - Line 14-18: Compute p_{short} if the measured range is less than or equal to the expected range.
  - Line 19-23: Compute p_{max} if the measured range equals the maximum sensor range.
  - Line 24-28: Compute p_{rand} if the measured range is within valid bounds.
- Line 29: Sum the probabilities of all components.
- Line 30: Accumulate the log probability, using a small ϵ to avoid taking the logarithm of zero.
- Line 32: Return the exponential of the accumulated log probability as the likelihood.

8.6 Practical Implementation Details

8.6.1 Handling Laser Sensor Offset

- The laser sensor is physically located 25 cm ahead of the robot's center. This offset is crucial for accurate ray casting and is accounted for when calculating the laser's position.
- The laser position is calculated using the robot's pose:

$$x_{\text{laser}} = x + d \cdot \cos(\theta)$$
$$y_{\text{laser}} = y + d \cdot \sin(\theta)$$

  where $d = 25$ cm.

Algorithm 4 Beam Range Finder Sensor Model

function BeamRangeFinderModel(z_t, x_t, m)
  Inputs: Measured laser scan z_t, robot pose x_t, map m
  Outputs: Likelihood $p(z_t \mid x_t, m)$

  log_prob $\leftarrow 0$ ▷ Initialize log probability
  for each beam i **do**
    $z_i \leftarrow z_t[i]$ ▷ Measured range
    $z_t^* \leftarrow \text{RayCastingBeam}(x_t, i, m)$ ▷ Expected range
    ▷ Compute probabilities for each component
    if $0 \leq z_i \leq z_{\max}$ **then**
      $p_{\text{hit}} \leftarrow w_{\text{hit}} \cdot \dfrac{1}{\sqrt{2\pi}\sigma_{\text{hit}}} \exp\left(-\dfrac{(z_i - z_t^*)^2}{2\sigma_{\text{hit}}^2}\right)$
    else
      $p_{\text{hit}} \leftarrow 0$
    end if
    if $0 \leq z_i \leq z_t^*$ **then**
      $p_{\text{short}} \leftarrow w_{\text{short}} \cdot \dfrac{\lambda_{\text{short}} \exp(-\lambda_{\text{short}} z_i)}{1 - \exp(-\lambda_{\text{short}} z_t^*)}$
    else
      $p_{\text{short}} \leftarrow 0$
    end if
    if $z_i = z_{\max}$ **then**
      $p_{\max} \leftarrow w_{\max}$
    else
      $p_{\max} \leftarrow 0$
    end if
    if $0 \leq z_i < z_{\max}$ **then**
      $p_{\text{rand}} \leftarrow w_{\text{rand}} \cdot \dfrac{1}{z_{\max}}$
    else
      $p_{\text{rand}} \leftarrow 0$
    end if

    $p \leftarrow p_{\text{hit}} + p_{\text{short}} + p_{\max} + p_{\text{rand}}$
    log_prob $\leftarrow$ log_prob $+ \ln(\max(p, \epsilon))$ ▷ Accumulate log probability
  end for
  return exp(log_prob)
end function

8.6.2 Map Representation and Occupancy Threshold

- The occupancy map is represented as a 2D grid where each cell contains a probability of being occupied.
- An occupancy threshold (e.g., 0.35) is used to determine whether a cell is considered free or occupied during ray casting.
- Cells with occupancy probabilities above the threshold are treated as obstacles.

8.6.3 Numerical Stability

- To prevent numerical underflow when multiplying many small probabilities, we accumulate the log probabilities.
- A small constant ϵ (e.g., 1×10^{-300}) ensures we avoid taking the logarithm of zero.

8.6.4 Beam Subsampling

- To reduce computational load, beam subsampling is employed by processing every n^{th} beam (e.g., $n = 2$).
- This approach balances computational efficiency with the need for sufficient data to accurately assess the likelihood.

8.6.5 Edge Case Handling

- Particles that end up outside the map boundaries or within obstacles are assigned very low weights.
- This discourages the particle filter from considering impossible or invalid states.

8.6.6 Unnormalized Mixture of Weights

- Although the book specifies that the sensor model needs to be a normalized probability distribution, however, as directed in the writeup in order to prioritize ease of debugging, the the mixture weights (and thus the distribution) are kept unnormalized with no unintended consequence as the particle weights are later normalized.

8.7 Implementation Correctness

8.7.1 Testing with Probability Distribution Visualization

- **Script Used:** `test_sensor_model_pdf.py`
- **Purpose:** Cross-check the sensor model's probability distribution by plotting $p(z_t \mid z_t^*)$ for various values of z_t^*.
- **Procedure:**
  - Set sensor model parameters (w weights, σ_{hit}, λ_{short}, $z_{\max}$).
  - For selected values of z_t^* (e.g., 500 mm, 2000 mm, 4000 mm), compute and plot each component p_{hit}, p_{short}, $p_{\max}$, p_{rand}.

- Plot the combined probability distribution to verify that it correctly represents the sum of the components.

- **Observations:**
  - The Gaussian component p_{hit} peaks at z_t^* with variance σ_{hit}^2, modeling accurate measurements.
  - The exponential component p_{short} is significant near zero and decreases exponentially, modeling unexpected short readings.
  - The p_{max} component is significant at z_{max}, modeling maximum range readings.
  - The uniform component p_{rand} has a constant low probability across the valid range, modeling random noise.
  - The combined distribution accurately reflects the expected behavior of the sensor model.

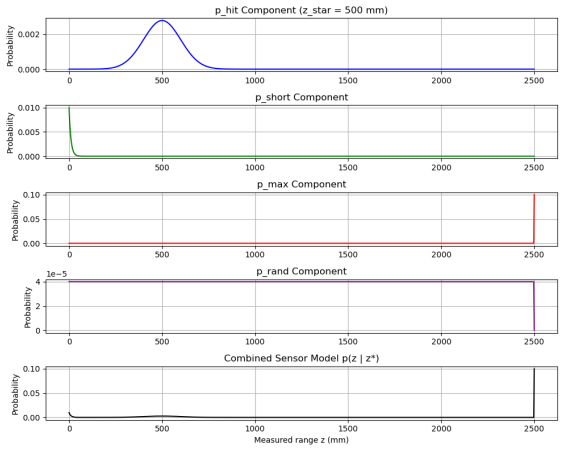


Figure 4: Probability distribution visualization for $z_t^* = 500$

8.7.2 Interactive Visualization and Validation

- **Script Used:** `_test_sensor_model.py`
- **Purpose:** Interactively test the ray casting algorithm by visualizing the laser beams and adjusting the robot's pose.
- **Features:**
  - Allows adjustment of the robot's position (x, y) and orientation (θ) using sliders.
  - Displays the laser beams overlaid on the map, showing where they intersect obstacles.
- **Benefits:**
  - Provides visual confirmation that the ray casting correctly identifies obstacles and calculates expected ranges.
  - Helps identify any discrepancies in the implementation, such as incorrect handling of map boundaries or sensor offset.

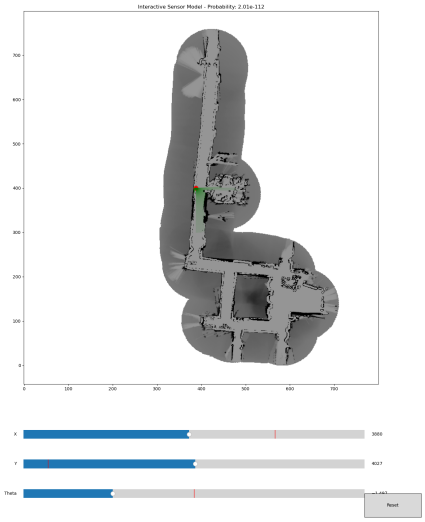


Figure 5: Interactive visualization of ray casting and sensor model

8.7.3 Validation of Ray Casting Accuracy

- Ensured that the ray casting algorithm correctly handles various scenarios:
  - Beams hitting obstacles at different distances and angles.
  - Beams not encountering any obstacle within $z_{\max}$, correctly assigning $z_t^* = z_{\max}$.
  - Proper handling of map boundaries to prevent accessing invalid indices.

9 Resampling Process Implementation and Analysis

9.1 Introduction

Resampling is another critical step in particle filtering, used to handle particle degeneracy where, over time, a few particles carry significant weights while others have negligible contributions. Resampling redistributes the particle set by sampling particles according to their weights, ensuring that particles with higher weights are more likely to be selected multiple times. This maintains a diverse and representative set of particles for the next iteration of the filter.

We implemented two standard resampling methods as described in Probabilistic Robotics by Thrun et al. (2005):

- Multinomial Resampling
- Low Variance Resampling

These methods differ in how they sample particles based on their weights, with low variance resampling providing better performance by reducing the variance incudced during resampling.

9.2 Implementation Overview

The resampling process operates on a set of particles $\{x_t[i], w_t[i]\}$ at time t, where $x_t[i]$ represents the state of particle i, and $w_t[i]$ is its associated weight. The goal is to generate a new set of particles $\{x_t^{\text{resampled}}[i], w_t^{\text{resampled}}[i]\}$ where particles are sampled according to their weights, and the weights are reset to $\frac{1}{N}$, with N being the number of particles.

9.2.1 Multinomial Resampling

Multinomial resampling selects particles independently, with replacement, based on their normalized weights. Each particle's probability of being selected is equal to its weight.

9.2.2 Low Variance Resampling

Low variance resampling, also known as systematic resampling, aims to minimize the variance introduced by the resampling process. It does so by ensuring that particles are sampled proportionally to their weights, but with reduced randomness compared to multinomial resampling.

9.3 Mathematical Formulation

9.3.1 Notation

- N: Number of particles.
- $x_t[i]$: State of particle i at time t.
- $w_t[i]$: Normalized weight of particle i at time t, such that $\sum_{i=1}^{N} w_t[i] = 1$.

- X_t: Set of particles $\{x_t[i], w_t[i]\}_{i=1}^N$.
- $X_t^{\text{resampled}}$: Resampled set of particles.

9.3.2 Multinomial Resampling

In multinomial resampling, particles are sampled independently based on their weights. The probability of selecting particle i is $w_t[i]$.

For $j = 1, \ldots, N$:

$$P(x_t^{\text{resampled}}[j] = x_t[i]) = w_t[i] \tag{6}$$

9.3.3 Low Variance Resampling

Low variance resampling reduces the variance by using a single random number to sample particles in a systematic way.

1. Compute the cumulative sum of weights c_i:

$$c_i = \sum_{k=1}^{i} w_t[k] \tag{7}$$

2. Generate a random number r uniformly distributed in $[0, \frac{1}{N}]$.
3. For $j = 1, \ldots, N$, compute:

$$U_j = r + \frac{j-1}{N} \tag{8}$$

4. Find the smallest index i such that $c_i \geq U_j$, and set $x_t^{\text{resampled}}[j] = x_t[i]$.

9.4 Algorithms

Algorithm 5 MultinomialResampling

function MULTINOMIALRESAMPLING(X_t)
  Input: Particle set $X_t = \{x_t[i], w_t[i]\}_{i=1}^N$
  Output: Resampled particle set $X_t^{\text{resampled}}$
  Normalize the weights: $w_t[i] \leftarrow \frac{w_t[i]}{\sum_{k=1}^N w_t[k]}$
  for $j = 1$ to N **do**
    Draw index i with probability $w_t[i]$
    Set $x_t^{\text{resampled}}[j] \leftarrow x_t[i]$
    Set $w_t^{\text{resampled}}[j] \leftarrow \frac{1}{N}$
  end for
  return $X_t^{\text{resampled}}$
end function

Algorithm 6 LowVarianceResampling

function LowVarianceResampling(X_t)
  Input: Particle set $X_t = \{x_t[i], w_t[i]\}_{i=1}^{N}$
  Output: Resampled particle set $X_t^{\text{resampled}}$
  Normalize the weights: $w_t[i] \leftarrow \frac{w_t[i]}{\sum_{k=1}^{N} w_t[k]}$
  Generate a random number $r \sim \text{Uniform}(0, \frac{1}{N})$
  Initialize $c = w_t[1]$, $i = 1$
  for $j = 1$ to N **do**
    $U_j = r + \frac{j-1}{N}$
    while $U_j > c$ **do**
      $i \leftarrow i + 1$
      $c \leftarrow c + w_t[i]$
    end while
    Set $x_t^{\text{resampled}}[j] \leftarrow x_t[i]$
    Set $w_t^{\text{resampled}}[j] \leftarrow \frac{1}{N}$
  end for
  return $X_t^{\text{resampled}}$
end function

9.5 Practical Implementation Details

9.5.1 Weight Normalization and Validation

- **Normalization**: Before resampling, weights are normalized to ensure they sum to 1, forming a valid probability distribution.
- **Validation**:
  - **Non-negative Weights**: The implementation checks that all weights are non-negative. Negative weights are not meaningful and indicate an error in prior computations.
  - **Non-zero Sum**: The sum of weights must be greater than zero. If the sum is zero, resampling cannot proceed, and an error is raised.

9.5.2 Handling Edge Cases

- **Zero Weights**: If all weights are zero, the resampling methods cannot generate a valid new set of particles. The implementation raises a `ValueError` in such cases.
- **Single Particle**: When only one particle is present, resampling returns the particle itself with weight reset to 1.

9.5.3 Random Number Generation

- **Randomness**: Random numbers are used in both resampling methods (`np.random.multinomi` for multinomial resampling and `np.random.uniform` for low variance resampling).

9.5.4 Weight Assignment After Resampling

- **Uniform Weights**: After resampling, all particles are assigned equal weights of $\frac{1}{N}$.
- **Justification**: This reflects the fact that after resampling, each particle is equally likely in the absence of new evidence.

10 Discussion of parameter tuning

10.1 Introduction

Once the motion model, sensor model, and resampling (low-variance) was implemented the important next step was to tune the parameters effectively. The performance of both the motion and sensor models heavily depends on the appropriate selection of their respective parameters. In this section, we discuss the iterative process of tuning the motion model parameters $(\alpha_1, \alpha_2, \alpha_3, \alpha_4)$ and the sensor model parameters $(z_{\text{hit}}, z_{\text{short}}, z_{\text{max}}, z_{\text{rand}}, \sigma_{\text{hit}}, \lambda_{\text{short}}, \text{max\_range})$. We also describe how adjusting the number of particles impacted the localization performance. The tuning process involved systematic experimentation, observation of outcomes, and refinement based on empirical results. **The tuning process took place on robotdata1.log since the provided gif resource could be used to infer the ground-truth of the log file and once desired repeatability without seeding was achieved, the parameters were locked and reused for the rest of the logs.**

10.2 Motion Model Parameter Tuning

10.2.1 Initial Parameters

We began with the default motion model parameters provided in the initial code:

- $\alpha_1 = 0.01$
- $\alpha_2 = 0.01$
- $\alpha_3 = 0.01$
- $\alpha_4 = 0.01$

These values were used as a starting point to model the uncertainty in the robot's motion due to control noise.

10.2.2 Iterative Tuning Process

Through iterative testing and observation, we adjusted the parameters to improve localization accuracy. The tuning involved running the particle filter with different parameter values and analyzing the resulting localization performance.

Observations and Adjustments:

- **High Noise Levels:** The initial values of 0.01 for all α parameters introduced significant noise into the motion model. This led to particles spreading too widely, causing the filter to converge slowly.
- **Reduction of Rotational Noise:** By decreasing α_1 and α_2, which represent the noise in rotational motion before and during translation, we aimed to reduce the overdispersion of particles due to rotation.
- **Adjustment of Translational Noise:** Similarly, we adjusted α_3 and α_4 to fine-tune the noise associated with translational motion and additional rotational noise after translation.

10.2.3 Final Parameters

After several iterations, we arrived at the following parameter values:

- $\alpha_1 = 0.0005$
- $\alpha_2 = 0.0005$
- $\alpha_3 = 0.001$
- $\alpha_4 = 0.001$

Rationale:

- **Observation in Trajectory Test with Real Data (Section 7.5.3.):** In 3 it was observed that there is a significant drift that is observed during rotation and rotation induced due to translation, hence the rationale for the below.
- **Lowered α_1 and α_2:** Reducing these values decreased the rotational noise, which resulted in particles maintaining a tighter distribution around the predicted motion path.
- **Slightly Higher α_3 and α_4:** Keeping these values slightly higher than α_1 and α_2 allowed for adequate modeling of translational noise and any additional rotational uncertainties.

10.2.4 Summary of Motion Model Tuning

Parameter	Initial Value	Final Value	Value Range Tuned
α_1	0.01	0.0005	0.01 - 0.00001
α_2	0.01	0.0005	0.01 - 0.00001
α_3	0.01	0.001	0.01 - 0.00001
α_4	0.01	0.001	0.01 - 0.00001

Table 3: Summary of Motion Model Parameter Tuning

10.3 Sensor Model Parameter Tuning

10.3.1 Initial Parameters

The initial sensor model parameters were set as:

- $z_{\text{hit}} = 1$
- $z_{\text{short}} = 0.1$
- $z_{\text{max}} = 0.1$
- $z_{\text{rand}} = 100$
- $\sigma_{\text{hit}} = 50$
- $\lambda_{\text{short}} = 0.1$
- max_range = 1000 mm

These parameters define the weights and characteristics of the different components in the beam range finder model.

10.3.2 Iterative Tuning Process

Using test scripts like `test_sensor_model_pdf.py` and `main.py`, we visualized the sensor model's probability distributions and assessed the impact of parameter changes on the likelihood computations.

Observations and Adjustments:

- **Dominance of Random Measurements:** The initial high value of $z_{\text{rand}} = 100$ caused the random measurement component to dominate the sensor model. This led to particles receiving high weights even when their poses did not align well with the observed measurements.
- **Increasing z_{hit}:** By increasing z_{hit}, we emphasized the importance of accurate measurements, thereby rewarding particles that closely matched the observed sensor data.
- **Adjusting σ_{hit}:** Increasing σ_{hit} from 50 to 100 allowed for a broader Gaussian distribution in p_{hit}, accommodating more measurement noise while still favoring accurate readings.
- **Expanding Maximum Range:** Increasing max_range from 1000 mm to 8183 mm better reflected the actual maximum range of the sensor, improving the modeling of p_{max} and p_{rand}.

10.3.3 Final Parameters

The final sensor model parameters were:

- $z_{\text{hit}} = 5$
- $z_{\text{short}} = 0.15$
- $z_{\text{max}} = 0.05$
- $z_{\text{rand}} = 1000$
- $\sigma_{\text{hit}} = 100$
- $\lambda_{\text{short}} = 0.1$
- max_range = 8183 mm

Rationale:

- **Increased z_{hit}:** Setting $z_{\text{hit}} = 5$ strengthened the contribution of the correct measurement component, improving the filter's ability to favor particles aligned with accurate sensor data.
- **Adjusted z_{rand}:** Increasing z_{rand} to 1000 balanced the influence of random measurements, ensuring that particles could still receive reasonable weights in the presence of sensor noise or unexpected readings.
- **Refined Weights:** The weights remained unnormalized, as per the hint provided, to simplify debugging and since particle weights are normalized later in the process.
- **Consistent λ_{short}:** Keeping $\lambda_{\text{short}} = 0.1$ maintained the rate of decay for unexpected short measurements, which aligned well with observed sensor behavior.

10.3.4 Summary of Sensor Model Tuning

Parameter	Initial Value	Final Value	Value Range Tuned
z_{hit}	1	5	1-10
z_{short}	0.1	0.15	0.01 - 0.9
z_{max}	0.1	0.05	0.01 - 0.9
z_{rand}	100	1000	100-1000
σ_{hit}	50	100	50-200
λ_{short}	0.1	0.1	0.01-0.1
max_range	1000 mm	8183 mm	1000 - 8183

Table 4: Summary of Sensor Model Parameter Tuning

10.4 Adjusting the Number of Particles

10.4.1 Initial Particle Count

We started with an initial particle count of **1000** particles. This number was chosen as a balance between computational efficiency and the need for sufficient particle diversity to represent the robot's belief distribution.

10.4.2 Observations and Adjustments

Observations:

- **Particle Depletion:** With 1000 particles, we observed that the filter sometimes struggled to maintain sufficient diversity, especially in complex environments or during rapid movements.
- **Localization Accuracy:** The estimated poses occasionally deviated significantly from the ground truth, indicating that the particle set was not adequately representing the belief distribution.

Adjustment:

- **Increased Particle Count:** We increased the number of particles to **2500** to enhance the representation of the belief distribution and improve localization accuracy.

10.4.3 Final Particle Count

The final implementation used **2500** particles.

Rationale:

- **Improved Diversity:** A higher number of particles provided a more detailed approximation of the posterior distribution, reducing the likelihood of particle depletion and improving the filter's robustness.
- **Improved Accuracy:** The increased particle count led to more accurate pose estimates, as there was a higher probability of particles being located near the true robot pose.
- **Computational Trade-off:** Although increasing the particle count raised computational demands, optimizations in the code (e.g., beam subsampling and efficient data structures) helped maintain real-time performance.

10.5 Impact of Parameter Tuning

The iterative tuning of the motion and sensor model parameters, along with adjusting the particle count, led to significant improvements in localization performance:

- **Faster Convergence:** The filter converged more quickly to the robot's true pose, even in the presence of noise and environmental uncertainties.
- **Increased Accuracy:** The estimated trajectories closely matched the ground truth, as verified through visualizations and error metrics.
- **Robustness:** The system became more resilient to sensor noise and dynamic changes in the environment (person walking infront of the robot as observed in the resource gif provided), maintaining accurate localization over extended periods.

11 Performance and results

Our particle filter implementation has shown robust performance across all five provided log files (robotdata1.log to robotdata5.log). The parameters were initially tuned using robotdata1.log, and these settings were then successfully applied to the remaining logs. This approach demonstrated the adaptability and generalization of our implementation.

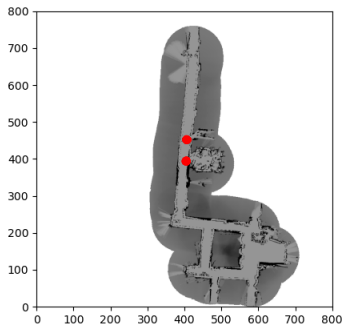
11.1 Performance Overview

The particle filter consistently performed well in 9 out of 10 trials for most log files, showcasing its reliability and repeatability. However, robotdata4.log presented some challenges in maintaining consistency, occasionally resulting in the elimination of the particle representing the actual robot position.

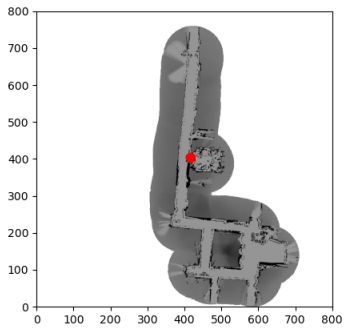
A visual demonstration of the results can be found in the following YouTube video:

https://youtu.be/rn9sCLVB_Vo

11.2 Results for Individual Log Files

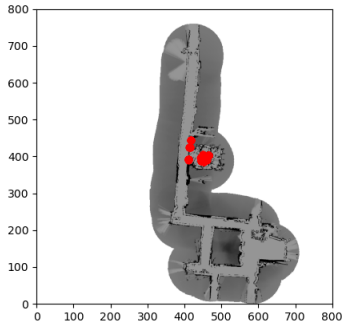


(a) Convergence from 2500 particles

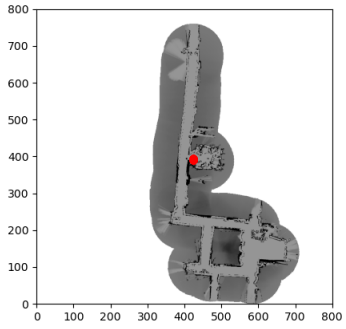


(b) Final particle position

Figure 6: Particle filter results for robotdata1.log

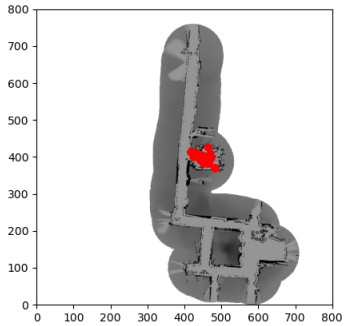


(a) Convergence from 2500 particles

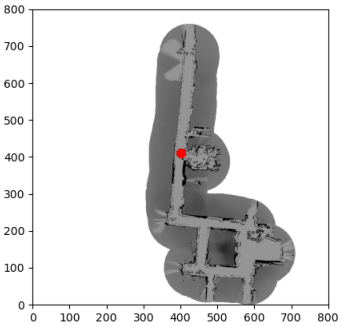


(b) Final particle position

Figure 7: Particle filter results for robotdata2.log

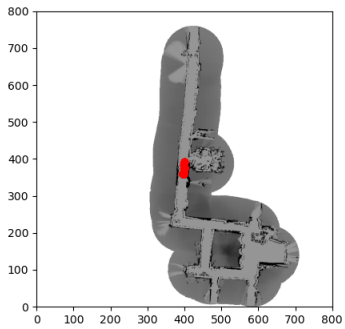


(a) Convergence from 2500 particles

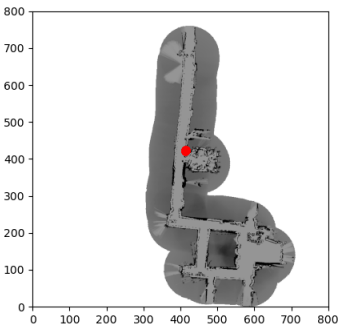


(b) Final particle position

Figure 8: Particle filter results for robotdata3.log



(a) Convergence from 2500 particles



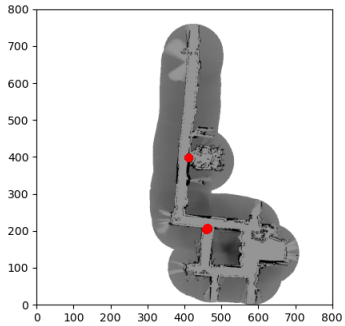
(b) Final particle position

Figure 9: Particle filter results for robotdata4.log

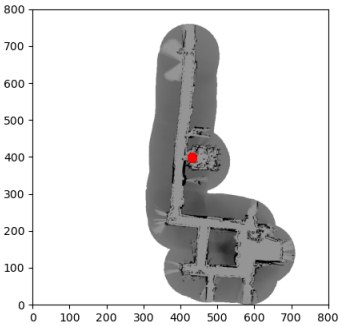
11.3 Analysis of Results

The particle filter demonstrated consistent and repeatable performance across most log files, with robotdata4.log being the exception. Key observations include:

- **Convergence**: In all cases, the filter successfully converged from the initial 2500 particles to a single or a few particles within 2-10 seconds, effectively localizing the robot.
- **Adaptability**: The parameters tuned on robotdata1.log proved effective for the other log files, highlighting the robustness of our implementation.
- **Consistency**: 9 out of 10 trials for most log files resulted in the same behavior, indicating high reliability.
- **Challenge with robotdata4.log**: This particular log file presented issues with consistency, occasionally resulting in the elimination of the correct particle. This suggests that there might be unique characteristics in this dataset that challenge our current implementation.



(a) Convergence from 2500 particles



(b) Final particle position

Figure 10: Particle filter results for robotdata5.log

12 Discussion of performance and future work

Although the existing solution does robustly and repeatedly converge the particles and localizes them effectively, but there is further room for improvement of the performance, and my further future work to extend this assignment would be to:

- Firstly, optimizing the granularity and selection of sensor beams. By experimenting with reducing the number of sensor measurements and selectively sampling the most informative angles, we can balance computational efficiency with the retention of crucial measurement information. This approach minimizes noise and processing time while preserving essential data needed for accurate localization.

- Implementing an adaptive particle count mechanism based on the Effective Sample Size (ESS) (as discussed by Dellaert etal.) to dynamically adjust the number of particles in response to localization uncertainty, through which the system can allocate more particles when uncertainty is high and reduce the count when confidence is sufficient. This adaptive strategy optimizes resource utilization and maintains particle diversity, enhancing the robustness of the localization process.

- Improving resampling methods by incorporating systematic or stratified resampling techniques which can reduce the variance introduced during the resampling process. This is something that I have already implemented as part of my coursework in Manipulation, Estimation, and Controls for the exact problem of a particle filter. These methods maintain particle diversity more effectively than traditional multinomial or low variance resampling, thereby minimizing particle impoverishment and improving the overall stability of the particle filter.

- Lastly, fine-tuning the parameters of the sensor model can lead to improvements in accuracy. A more systematic and effective method for tuning the parameters will need to be explored.

13 Extra Credit: Kidnapped robot problem

13.1 Introduction

The **kidnapped robot problem** is a scenario in robotics where a robot is abruptly moved to an unknown location without its knowledge. This situation challenges the localization system, as the robot must recognize that it is lost and re-localize itself in the new environment. Addressing this problem is crucial for developing robust localization systems capable of handling unexpected events.

For extra credit, we extended our Monte Carlo Localization (MCL) implementation to handle the kidnapped robot problem. We simulated the kidnapping by removing a chunk of odometry and laser data from the log file, causing a sudden discontinuity in the robot's sensor readings. We then implemented mechanisms to detect the kidnapping event and re-initialize the particle filter accordingly.

The code for kidnapped robot can be found in the zipped file submitted under the **extra_credit_kidnapped_robot** folder.

13.2 Video Demonstration

A video demonstration of our implementation handling the kidnapped robot problem can be found at the following link:
https://youtu.be/MSv7KK9KJmg

13.3 Simulation of the Kidnapped Robot Scenario

13.3.1 Modifying the Log File

To simulate the kidnapped robot scenario, we edited the log file `robotdata1.log` by removing lines **1203 to 1503**. This removal represents a period during which the robot's sensor data is unavailable or the robot is physically moved without corresponding odometry updates.

13.3.2 Observations with the Original Implementation

Using the original MCL implementation, we observed the following:

- **Particle Divergence**: The particles continued to propagate based on outdated odometry data, unaware of the kidnapping event.
- **Localization Failure**: The particle filter failed to recognize that the robot was in a new location, resulting in incorrect pose estimates with negligible weights.
- **Inability to Recover**: Without mechanisms to detect and handle kidnapping, the filter could not re-localize, rendering the robot effectively lost.

13.4 Implementing Kidnapping Detection and Recovery

To address the kidnapped robot problem, we introduced changes to the MCL algorithm to detect the kidnapping event and re-initialize the particle filter to recover from it.

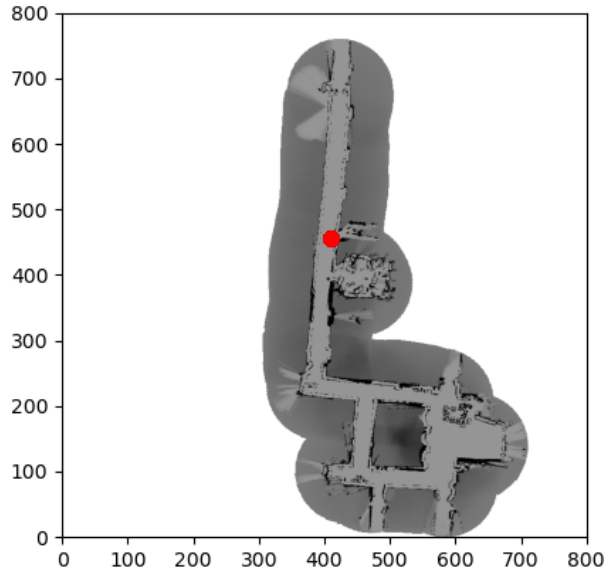


Figure 11: Particle final location due to drift without kidnap detection

13.4.1 Kidnapping Detection

We implemented kidnapping detection based on the following principle:

- **Maximum Weight Threshold**: We monitored the maximum particle weight at each time step. If the maximum weight fell below a predefined threshold, it indicated that the particles were no longer consistent with the sensor measurements, suggesting a kidnapping event.

13.4.2 Particle Re-initialization Strategy

Upon detecting kidnapping, we re-initialized the particles to help the robot relocalize:

- **Hybrid Re-initialization**: We combined two strategies:
  - **Particles Around Last Known Position**: Half of the particles were initialized around the robot's last estimated pose with added Gaussian noise.
  - **Particles Randomly Distributed**: The remaining particles were randomly distributed across the map.

13.4.3 Adjusting Motion Model Parameters

To facilitate recovery:

- **Temporary Increase in Noise Parameters**: We set higher values for the motion model parameters during the kidnapping recovery period:
  - $\alpha_1 = 0.1$
  - $\alpha_2 = 0.1$

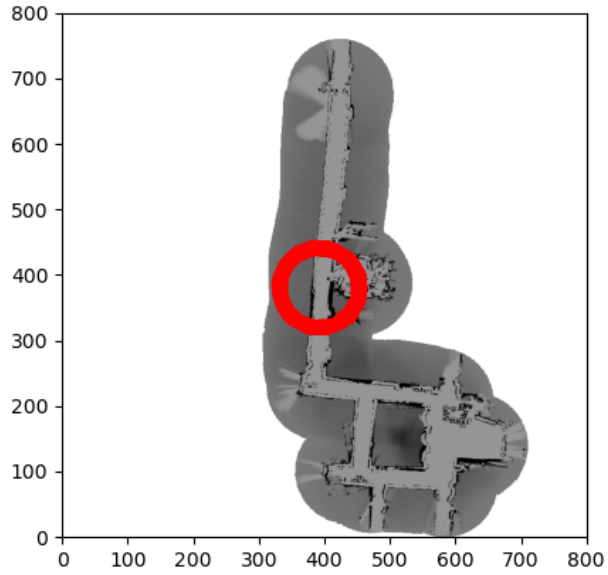


Figure 12: Particle reinitialization after kidnap detection

- $\alpha_3 = 0.2$
- $\alpha_4 = 0.2$

- **Duration Control**: We kept the increased noise parameters active for a specified number of time steps (e.g., 50 steps) before reverting to the normal values.

13.5 Future Enhancements

Several potential improvements could further enhance the robustness and efficiency of our kidnapped robot solution:

- **Adaptive Thresholds**: Implementing adaptive thresholds based on statistical analysis of particle weights could improve detection reliability. This approach would allow the system to dynamically adjust its sensitivity to potential kidnapping events based on the current distribution of particle weights.

- **Injecting random particles**: Injecting a small percentage of random particles at each iteration could help the filter recover without explicit kidnapping detection. This technique would maintain a constant level of exploration, potentially allowing the filter to recover from kidnapping events more naturally.

14 Extra Credit: Vectorized Python

14.1 Introduction

For the extra credit task of vectorizing our Monte Carlo Localization (MCL) implementation, we created a separate folder named **"extra_credit_vectorized"**. This folder contains vectorized versions of the code, optimizing key components while preserving the original codebase.

14.2 Changes Made for Vectorization

`motion_model.py`, and `sensor_model.py`.

14.2.1 `main.py`

- Eliminated particle loops in favor of vectorized operations
- Implemented vectorized motion and sensor model updates

14.2.2 `motion_model.py`

- Vectorized the `update` method to process all particles simultaneously
- Applied noise to motion parameters using NumPy array operations

14.2.3 `sensor_model.py`

- Vectorized the `beam_range_finder_model` method for parallel processing of particles and laser beams
- Developed vectorized ray casting lookup for efficient expected range retrieval
- Vectorized probability computations for `p_hit`, `p_short`, `p_max`, and `p_rand`

References

[1] Thrun, S., Burgard, W., & Fox, D. (2005). *Probabilistic Robotics*. MIT Press.

[2] Fox, D., Burgard, W., Dellaert, F., & Thrun, S. (1999). Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)* (pp. 343–349).

[3] Thrun, S., Fox, D., Burgard, W., & Dellaert, F. (2001). Robust Monte Carlo Localization for Mobile Robots. *Artificial Intelligence*, 128(1-2), 99–141.

[4] Doucet, A., de Freitas, N., & Gordon, N. (Eds.). (2001). *Sequential Monte Carlo Methods in Practice*. Springer.

[5] Arulampalam, M. S., Maskell, S., Gordon, N., & Clapp, T. (2002). A Tutorial on Particle Filters for Online Non-linear/Non-Gaussian Bayesian Tracking. *IEEE Transactions on Signal Processing*, 50(2), 174–188.

[6] Dellaert, F., Fox, D., Burgard, W., & Thrun, S. (1999). Monte Carlo Localization for Mobile Robots. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)* (Vol. 2, pp. 1322–1328).

[7] Fox, D. (2003). Adapting the Sample Size in Particle Filters Through KLD-Sampling. *The International Journal of Robotics Research*, 22(12), 985–1003.

[8] Thrun, S., Fox, D., & Burgard, W. (1998). A Probabilistic Approach to Concurrent Mapping and Localization for Mobile Robots. *Autonomous Robots*, 5(3-4), 253–271.

[9] Doucet, A., & Johansen, A. M. (2009). A Tutorial on Particle Filtering and Smoothing: Fifteen Years Later. In D. Crisan & B. Rozovskii (Eds.), *Handbook of Nonlinear Filtering* (pp. 656–704). Oxford University Press.

[10] Van Der Merwe, R., Doucet, A., De Freitas, N., & Wan, E. (2000). The Unscented Particle Filter. *Advances in Neural Information Processing Systems*, 13, 584–590.

[11] The Construct. (2020, May 13). [ROS Q&A] 179 - Set start position of robot within amcl. *YouTube*. Retrieved from https://www.youtube.com/watch?v=HmSdUagisAE

[12] (2023, March 29). Particle Filter for Multiple Object Tracking in Python: A Practical Guide. *YouTube*. Retrieved from https://www.youtube.com/watch?v=eEebeuWsZBk