UNIVERSITY OF VIRGINIA
LEARNING IN ROBOTICS
CS 6501
SPRING 2025

HOMEWORK 3

LIYUAN HOU [ZBZ4GU@VIRGINIA.EDU]

Solution 1 (Time spent: 14h 30min)**. (a)**

Environment Plot. We consider a 10×10 grid world, where each cell represents a unique state. The robot starts at state $(3, 6)$ (marked in blue) and aims to reach the goal at state $(8, 1)$ (marked in green). Several cells are designated as obstacles, including $(3, 2)$, $(4, 4)$, $(5, 7)$, and $(7, 5)$, and are shaded in light gray. These obstacle cells are treated as absorbing states that trap the robot and impose a heavy penalty.

At each valid state, the robot can take one of four stochastic actions: north, south, east, or west. The action succeeds in the intended direction with probability 0.7, and with probability 0.3 the robot transitions to a neighboring cell or remains in place. Any attempted transition into an obstacle or beyond the grid boundary results in the robot staying in its current state and incurring a large penalty.

The grid world environment was implemented using Python and the `matplotlib` library. Figure 1 provides a visualization of the coded environment, confirming its consistency with the problem description.

Cost Function Design. The objective of this problem is to design a cost structure that encourages the robot to reach the goal efficiently while avoiding obstacle states and invalid transitions. Since the problem is formulated as an infinite-horizon discounted Markov Decision Process, we define the core components of the environment, namely the transition probabilities and cost functions.

Let $\mathcal{S}$ denote the set of all valid states in the 10×10 grid, excluding obstacle states, and let $\mathcal{A} = \{\text{North}, \text{South}, \text{East}, \text{West}\}$ be the set of available actions. Each action $u \in \mathcal{A}$ taken at a state $x \in \mathcal{S}$ leads to a next state $x' \in \mathcal{S}$ with probability $T(x, u, x') = P(x' \mid x, u)$.

The transition model is represented by a tensor $T \in \mathbb{R}^{100\times100\times4}$, where the entry $T[x, x', u]$ denotes the probability of transitioning from state x to state x' under action u. That is, each action u corresponds to a transition matrix $T^{(u)} \in \mathbb{R}^{100\times100}$. The dynamics are stochastic: the intended direction occurs with probability 0.7, while the remaining probability 0.3 is distributed evenly among unintended neighboring directions or staying in place. Specifically, when the action is North, the robot moves north with probability 0.7, west with 0.1, east with 0.1, and remains in place with 0.1. When the action is South, the probabilities are 0.7 for south, 0.1 for west, 0.1 for east, and 0.1 for no movement. For the East action, the robot moves east with probability 0.7, north with 0.1, south

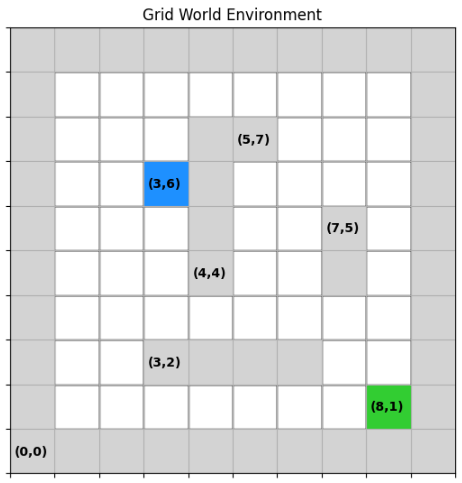


FIGURE 1. Grid World Environment

with 0.1, and remains in place with 0.1. Finally, for the West action, the robot moves west with probability 0.7, north and south each with 0.1, and remains in place with 0.1. If a transition would result in moving outside the grid or into an obstacle cell, the robot remains in its current state, and such transitions are treated as invalid and penalized. Obstacle cells are modeled as absorbing states with no outgoing transitions.

We define the immediate reward function $r(x, u, x')$ as follows. If the robot transitions into the goal state x_{goal} from a non-goal state, it receives a one-time reward of $+10$. If the resulting state x' or the current state x is an obstacle, or if the action attempts to move outside the grid, the robot incurs a penalty of -100. In all other cases, taking an action incurs a cost of -1, reflecting energy or time usage. The goal state is modeled as an absorbing state, where the robot remains indefinitely with zero additional reward.

The expected immediate cost function $q(x, u)$ is defined as the negative expected reward for taking action u in state x, computed as:

$$q(x, u) = \mathbb{E}_{x'}[-r(x, u, x')] = \sum_{x' \in \mathcal{S}} T(x, u, x') \cdot [-r(x, u, x')]$$

This formulation integrates the stochastic nature of actions and their associated outcomes. It penalizes transitions into obstacles and encourages the robot to reach the goal in as few steps as possible.

The overall optimization objective is to find a stationary policy π that minimizes the expected cumulative discounted cost:

$$J^\pi(x_0) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t q(x_t, u_t)\right]$$

where x_0 is the initial state, $u_t = \pi(x_t)$ is the action selected under policy π, and $\gamma = 0.9$ is the discount factor. This encourages short-term efficient behavior while accounting for long-term consequences.

(b)

Figure 2 presents the value function $J^{\pi^{(0)}}(x)$ under the initial policy $\pi^{(0)}$, in which the robot always attempts to move east. The heatmap highlights several notable characteristics about how the environment's structure and the stochastic nature of movement influence the expected cost-to-go from each cell.

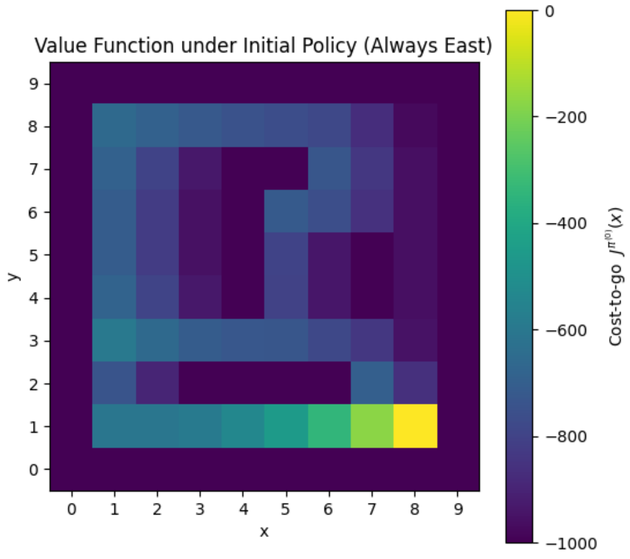


FIGURE 2. Value Function $J^{\pi^{(0)}}(x)$ under Initial Policy (Always East). Lower values indicate higher expected cost-to-go.

First, cells that are immediately to the left of obstacles or walls tend to exhibit a significantly lower value (i.e., higher cost-to-go). This outcome is due to the probabilistic transition model, which assigns a 0.1 chance of remaining in place, as well as a 0.1 chance each of veering north or south, even when the robot attempts to move east. As a result, any failed movement that leads into an obstacle or wall incurs the heavy penalty of -100, which dominates the expected return and leads to low value estimates for these cells.

Second, the value function exhibits a favorable gradient in the vicinity of the goal state at $(8, 1)$. States that are close to the goal, particularly those directly to its west or north, tend to have higher values (i.e., lower cost-to-go), as they are more likely to reach the terminal reward of $+10$ in fewer steps and thus accumulate less action penalty. Since the robot is allowed to stay in the goal state without further cost, the expected value stabilizes at a high level there.

Third, the value distribution also reflects the bias of the fixed policy. Because the robot always attempts to move east, it struggles to navigate around vertical obstacle walls or reach the goal from diagonally misaligned paths. This leads to poor value estimates in certain regions, particularly the top-left corner and those cells where eastward progress is blocked by obstacles or grid boundaries.

Lastly, obstacle cells themselves are clearly identifiable by their extreme negative values in the heatmap. Once a robot enters one of these states, it becomes trapped and continues to incur the severe penalty at every timestep, resulting in a cost-to-go near -1000 under the discount factor $\gamma = 0.9$.

(c)

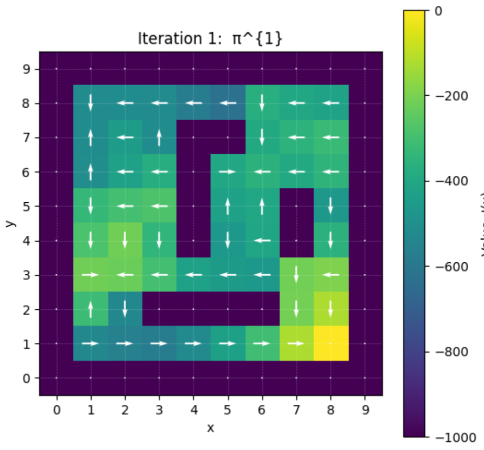
To solve for the optimal policy π^*, we implemented the policy iteration algorithm, which consists of alternating between policy evaluation and policy improvement until convergence. The algorithm begins with an initial policy where every valid state is assigned the action 'East'. The robot's environment includes both internal and boundary obstacles, and the stochastic transitions impose penalties for unintended outcomes or invalid moves.

In the policy evaluation step, we compute the value function $J^\pi(x)$ for the current policy by solving the Bellman expectation equation iteratively until the values converge within a small threshold. For each state, the expected return is computed by summing over all possible outcomes of the current action, accounting for the transition probabilities, the immediate reward, and the discounted value of the next state.

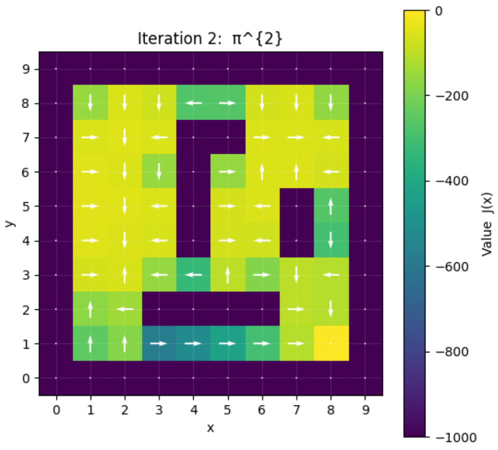
In the policy improvement step, we examine each valid state and compute the expected return for all four actions. The policy is updated to select the action with the highest expected return. If the policy remains unchanged after this step, the algorithm is considered to have converged.

Figure 3 presents the feedback control $u^{(k)}(x)$ and value function $J^{\pi^{(k)}}(x)$ over the first four iterations. Each subplot includes arrows to indicate the selected action and a heatmap to show the corresponding value. Initially, the policy lacks coherence, and the value function remains low across the grid. As iterations proceed, the policy begins to favor directions that reduce expected cost and move closer to the goal state at $(8, 1)$. The value function improves accordingly, with increasingly higher estimates in regions near the goal.

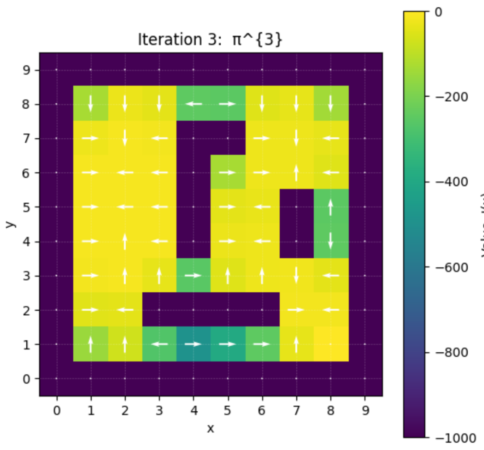
Based on our code outputs, the algorithm converged after 5 policy improvement steps. The final result is shown in Figure 4. However, the resulting policy does not visibly produce a clear trajectory toward the goal state, despite extensive debugging and verification of the implementation. I consider this behavior is likely due to a combination of strong penalties for invalid transitions and the stochastic nature of the movement model, which together discourage the agent from navigating near risky regions.



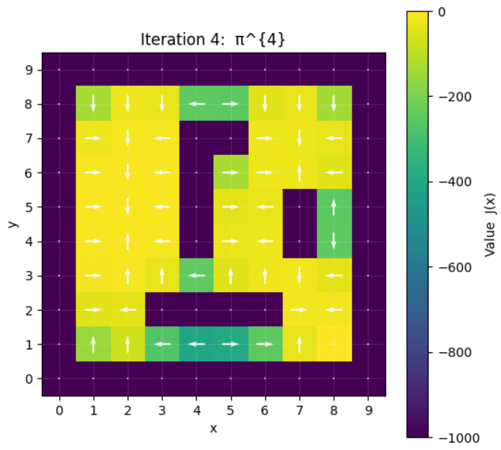
(A) Iteration 1: $\pi^{(1)}$



(B) Iteration 2: $\pi^{(2)}$



(C) Iteration 3: $\pi^{(3)}$



(D) Iteration 4: $\pi^{(4)}$

FIGURE 3. Policy and value function $J^{\pi^{(k)}}(x)$ over the first four iterations of policy iteration.

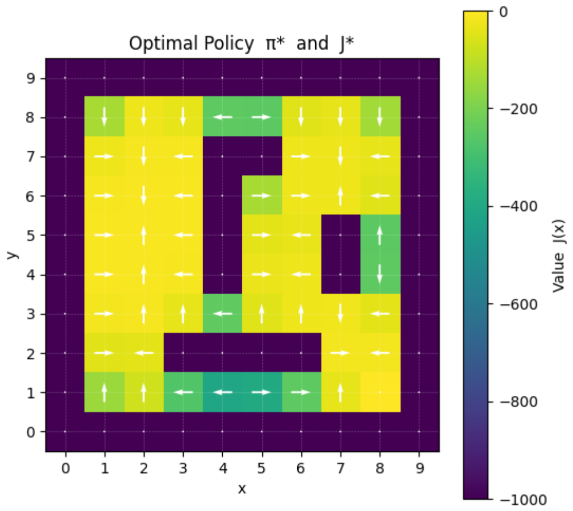


FIGURE 4. Final converged optimal policy π^* and value function J^*.

Solution 2 (Time spent: 10h)**. (c)**

In this section, we implemented the dynamics update step of a particle filter for simultaneous localization and mapping (SLAM). The purpose of this step is to propagate each particle forward in time based on the robot's estimated motion, while incorporating uncertainty through noise.

To achieve this, we first compute a control input that represents the robot's relative motion between two consecutive time steps. This control input includes a change in position and orientation $(\Delta x, \Delta y, \Delta \theta)$, and is computed using the robot's onboard odometry data. The relative transformation is calculated in the local frame of the robot at time $t-1$, so that the effect of heading is properly considered when moving forward.

Each particle's state is then updated by applying this control input to its current pose. This is done using a motion composition function that models rigid body transformations on SE(2), ensuring that both translation and rotation are handled correctly. To reflect real-world motion uncertainty, we perturb the control input with small Gaussian noise drawn from a zero-mean multivariate distribution with covariance matrix Q.

After applying the motion and noise, each particle represents a plausible pose of the robot at the new time step. This process allows the particle filter to maintain a distribution over potential robot locations over time.

To validate the implementation, we ran a test that visualizes both the odometry trajectory and the propagated particle trajectories. As shown in Figures 5 to 8, the propagated particle trajectories remain closely aligned with the odometry path, demonstrating the correctness of our dynamics step and confirming that the motion model accurately reflects the system's real movement.

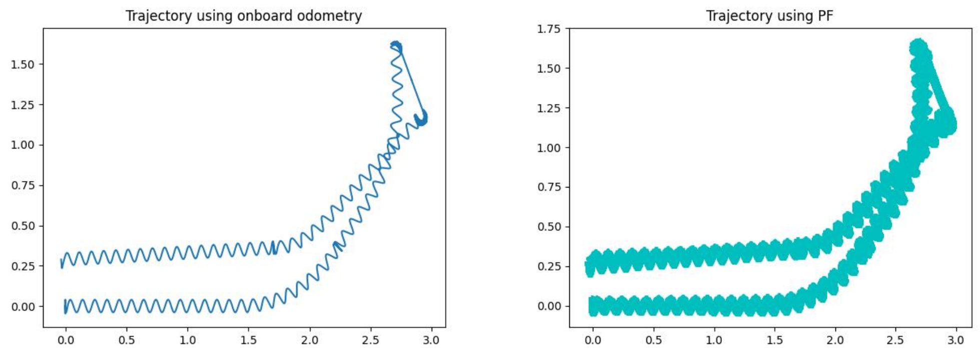


FIGURE 5. Dataset 0: Left: Robot trajectory using odometry. Right: Particle trajectories using the dynamics model. The similarity between the two indicates that the dynamics step is working as expected.

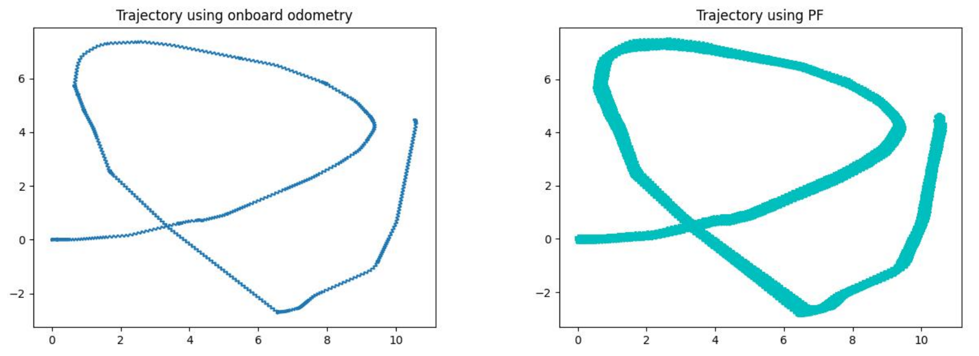


FIGURE 6. Dataset 1: Left: Robot trajectory using odometry. Right: Particle trajectories using the dynamics model. The similarity between the two indicates that the dynamics step is working as expected.

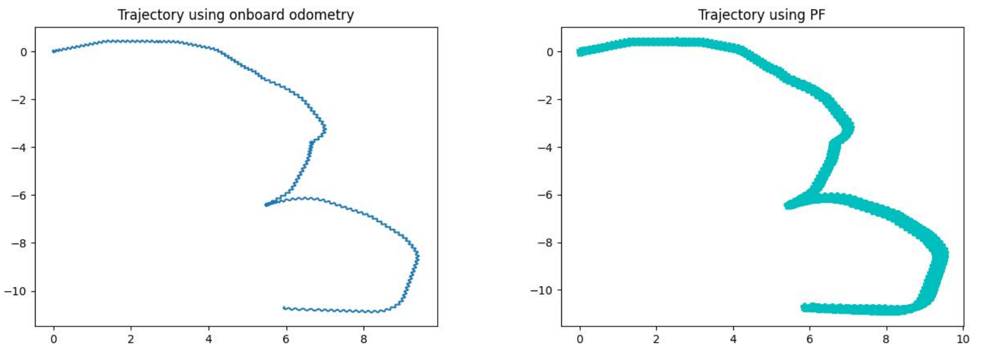


FIGURE 7. Dataset 2: Left: Robot trajectory using odometry. Right: Particle trajectories using the dynamics model. The similarity between the two indicates that the dynamics step is working as expected.

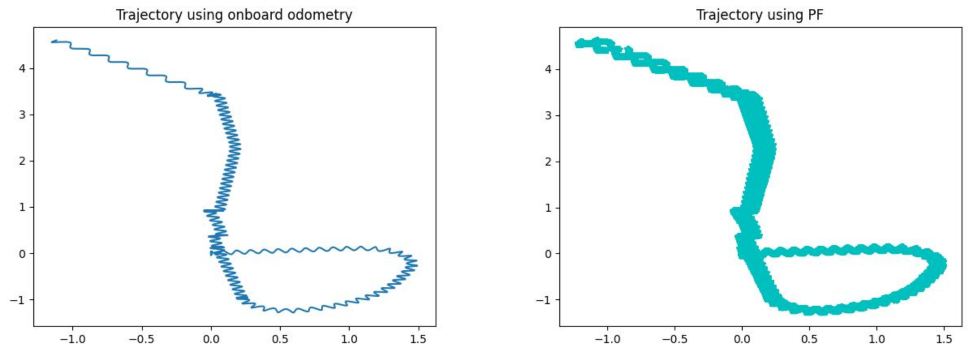


FIGURE 8. Dataset 3: Left: Robot trajectory using odometry. Right: Particle trajectories using the dynamics model. The similarity between the two indicates that the dynamics step is working as expected.

(d)

In the observation step, we integrate LiDAR sensor data with the current occupancy grid map to perform both localization and mapping. Localization is achieved by computing the likelihood of each particle based on how well its predicted sensor observations align with the map, while mapping involves updating the log-odds grid using the most probable particle.

We begin by extracting the head and neck joint angles for the current time step. These are used to determine the relative pose of the LiDAR in the robot's body frame. For each particle, the scan distances are projected from the LiDAR frame into world coordinates by applying two transformation matrices: one from the LiDAR to the body frame using the neck angle and LiDAR height, and another from the body frame to the world frame using the particle's pose and head height.

The resulting (x, y) positions represent the endpoints of LiDAR beams and are mapped to grid cells using the `grid_cell_from_xy` function. For each particle, we accumulate a log-likelihood score by summing the log-odds values of the corresponding grid cells. This score indicates how consistent the predicted hits are with the current map. Particle weights are then updated using a numerically stable log-softmax function.

The particle with the highest weight is selected for map updates. The LiDAR scan is re-projected into world coordinates using this particle's pose. The corresponding grid cells are marked as occupied. To update free space, we rasterize the cells between the particle's position and each LiDAR hit using skimage.draw.line. This method ensures that intermediate cells along the laser rays are marked as free space. A binary mask is used to collect all such traversed cells.

We then update the log-odds values: occupied cells are incremented by a constant (representing strong evidence of obstacles), and free cells are incremented by a smaller value (representing weaker evidence of free space). The log-odds values are clipped to remain within defined bounds, and the binary occupancy grid is updated by thresholding the log-odds values.

Finally, we perform stratified resampling to maintain a diverse and representative set of particles. This concludes the observation step, enabling robust SLAM through joint inference of the robot's pose and environment.

To verify our implementation, we ran the `run_observation_step()` function provided in `main.py`. The updated weights correctly reflected the particle closest to the origin receiving the highest likelihood, demonstrating that both the likelihood estimation and map update logic function as intended.

```
INFO:root:> Particles
: [[2.  0.2 3. ]
   [2.  0.4 5. ]
   [2.7 0.1 4. ]]
INFO:root:> Weights: [1.53e-150 1.18e-231 1.00000000e+000]

INFO:root:> Particles
: [[2.  0.2 3. ]
   [2.  0.4 5. ]
   [2.7 0.1 4. ]]
INFO:root:> Weights: [6.98e-81 0.00e+00 1.00000000e+00]
INFO:root:> Particles
: [[2.  0.2 3. ]
   [2.  0.4 5. ]
   [2.7 0.1 4. ]]
INFO:root:> Weights: [0. 0. 1.]
INFO:root:> Particles
: [[2.  0.2 3. ]
   [2.  0.4 5. ]
   [2.7 0.1 4. ]]
INFO:root:> Weights: [5.48e-037. 2.96e-323 1.00e+000]
```

(f)

To perform full SLAM, we implemented the `run_slam()` function in `main.py`, which iteratively performs a dynamics step followed by an observation step for every time step after t_0. We initialize the SLAM system using the earliest time step where both LiDAR and joint data are available. At this point, the map is constructed using a single high-confidence particle, and then re-initialized with $n = 100$ particles for full particle filtering. We assign uniform initial weights to the particles and propagate them through time using motion and observation updates.

In the dynamics step, each particle is updated based on the control input computed using smart minus between LiDAR poses at t and $t - 1$, and we apply Gaussian noise with covariance $Q = \text{diag}(2 \times 10^{-4}, 2 \times 10^{-4}, 1 \times 10^{-4})$ to ensure realistic motion uncertainty. In the observation step, we evaluate each particle using LiDAR likelihood computed from the current log-odds occupancy grid and update the weights using a softmax function in log-space.

We record two trajectories: the odometry trajectory obtained from LiDAR ground truth, and the estimated trajectory from the highest-weighted particle at each step. These trajectories, along with the final occupancy grid maps, are visualized in Figures 9 to 12. Each figure shows both the binarized occupancy map and a trajectory comparison, with the estimated particle filter path overlaid against the LiDAR odometry.

To ensure computational efficiency, the current implementation evaluates all time steps in the dataset. If runtime performance is a concern, one may consider processing every 5–10 frames, which would still maintain map fidelity but improve speed significantly. In our experiments, we observed that the SLAM trajectory qualitatively aligns well with odometry, and slight deviations are within expected margins due to particle diversity and observation noise.

Finally, we verified the correctness of the implementation by inspecting the final map and comparing the estimated and odometry trajectories. For all four datasets, our system successfully constructed accurate maps and produced reasonable trajectories. Minor tuning was applied to the process noise covariance Q to strike a balance between exploration and localization accuracy.

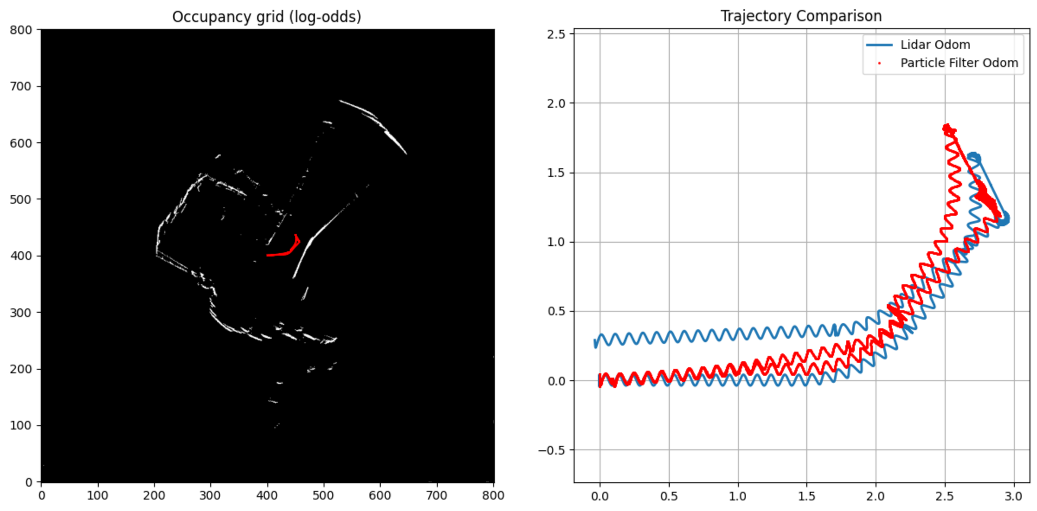


FIGURE 9. Dataset 0: (Left) Final Occupancy Grid. (Right) Trajectory Comparison.

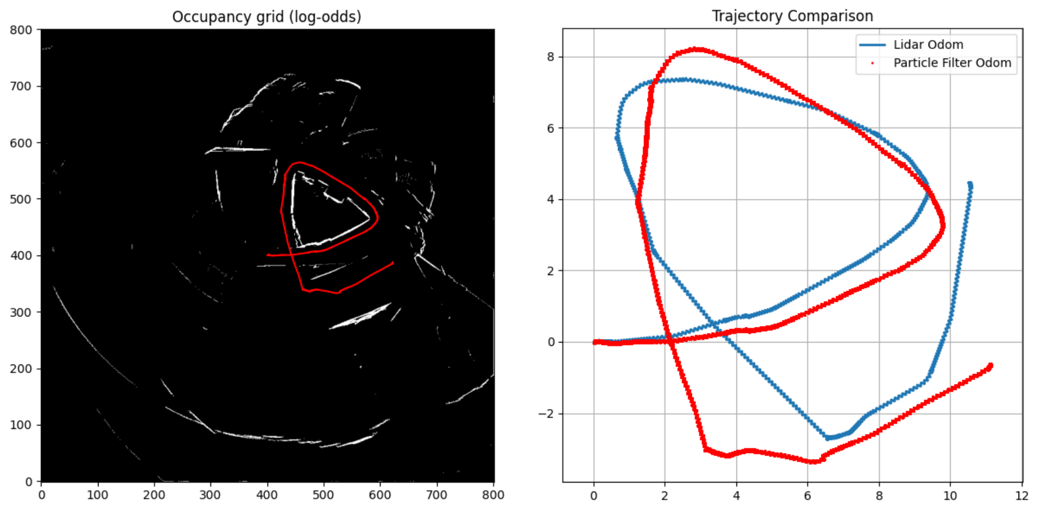


FIGURE 10. Dataset 1: (Left) Final Occupancy Grid. (Right) Trajectory Comparison.

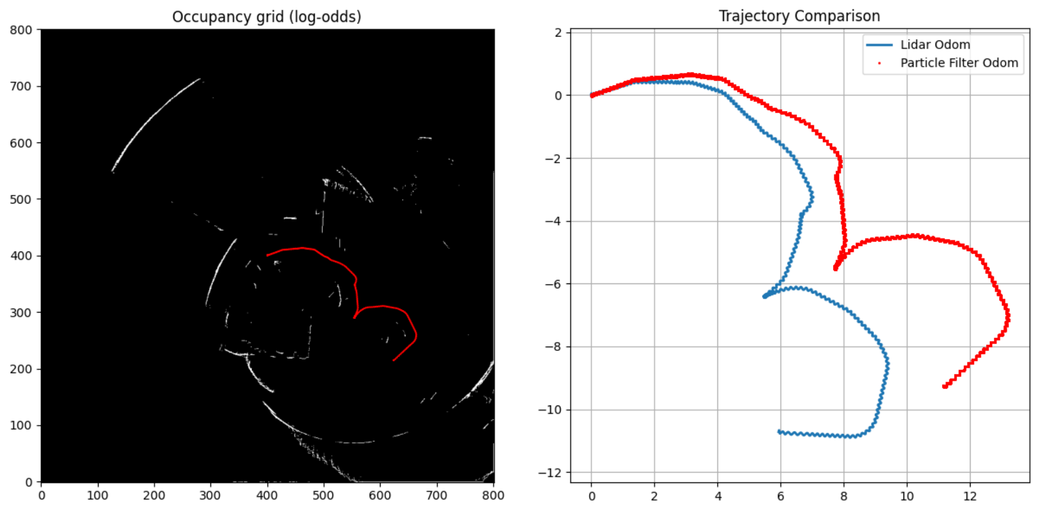


FIGURE 11. Dataset 2: (Left) Final Occupancy Grid. (Right) Trajectory Comparison.

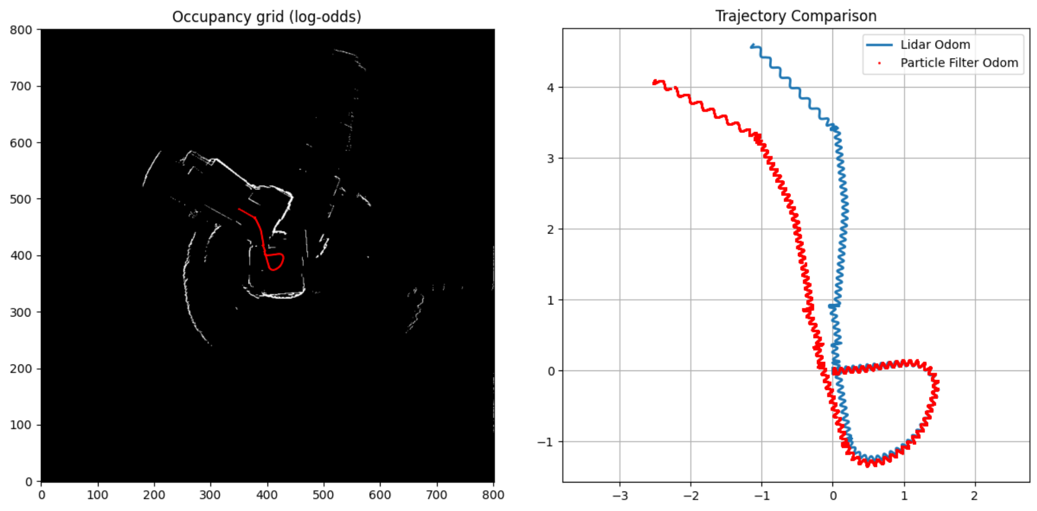


FIGURE 12. Dataset 3: (Left) Final Occupancy Grid. (Right) Trajectory Comparison.