A Simulation System of Monte Carlo Localization

Zhengtao Gao
zhengtao.gao@usi.ch

Xingqiao Hu
xingqiao.hu@usi.ch

I. Introduction

Monte Carlo Localization (MCL) is used to estimate the position and orientation of the robot in a known environment. This project implements a complete simulation system with the following features:

- A point robot model equipped with a distance sensor.
- Distance measurement simulation with easily adjustable motion and sensor noise.
- Simple world map and obstacles creation.
- An MCL algorithm using a Gaussian likelihood model, with a tunable combination of Low-Variance Resampling (LVR) and random resampling strategies.
- Extensive configuration options for all simulation aspects.

This project is designed to help users develop a basic understanding of the MCL algorithm and see how the MCL algorithm works in an intuitive way. In the remainder of this report, we present the architecture of the system in Section II. Then, in Section III, we discuss the key components of some modules in detail. Finally, in Section IV, we illustrate some of the examples we produced.

II. System Architecture

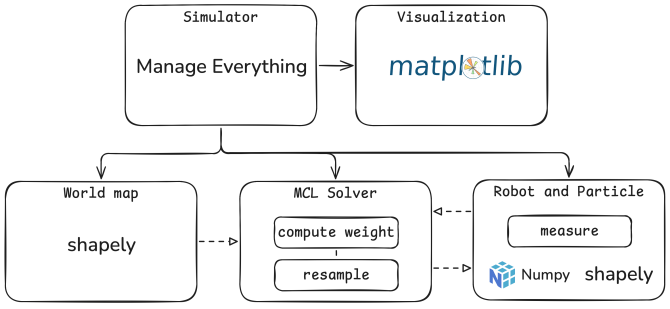

Fig. 1. The architecture of the MCL simulation system.

As shown in Figure 1, the MCL simulator is built with a modular design, dividing the system into four main components – **Simulator**, **World Map**, **MCL Solver**, and **Robot (Particles)**.

The Simulator class serves as the main controller, managing interactions between subsystems to provide a complete environment for robot localization simulation and visualization.

The World Map component defines the simulation environment using the Shapely[1] library. It manages the polygonal boundaries of the world map and obstacles. It also provides the service of point sampling within valid regions for MCL Solver.

Algorithm 1: Monte Carlo Localization (MCL)

```
foreach timestep do
    Move robot and particles with noise;
    Take distance measurements;
    Update particle weights based on measurements;
    if N_eff < threshold then
        Resample particles;
    end
    Update visualization;
end
```

The MCL Solver implements the core Monte Carlo Localization algorithm with two primary functions:

- Weight Computation: Updates particle weights using a Gaussian likelihood model based on sensor measurement error with respect to real robot.
- Resampling: Applies Low Variance Resampling (LVR) combined with random sampling to maintain particle diversity and prevent filter degeneracy.

The Robot (Particle) component simulates the action and observation model for both the real robot and the sampled particles. We use NumPy[2] to efficiently vectorize the computations associated with the particle set.

A visualization system based on Matplotlib[3] is integrated into the Simulator. It renders the world environment, robot position, and particle distribution. The result can be exported as MP4 videos for analysis and presentation purposes.

The system follows a cyclic data flow pattern:

1) **Initialization**: Simulator initializes all components, establishes the simulation environment, and sets up visualization.
2) **Action Phase**: The Robot and Particle execute movement commands with noise modeling.
3) **Sensing Phase**: Distance measurements are performed by the Robot and Particles against the boundaries of World Map.
4) **Update Phase**: MCL Solver update Particle weights based on the measurements and perform resampling if needed.
5) **Rendering Phase**: Simulator updates the visualization with the current states.

[1]https://github.com/shapely/shapely

[2]https://numpy.org/

[3]https://matplotlib.org/

6) The cycle then repeats from the Action Phase until the total number of steps is reached.

The overall procedure is summarized in Algorithm 1.

III. Key Components And Implementation Details

This section provides a detailed analysis of the core components within the MCL simulation system. Specifically, we focus on two main modules: the MCL solver (`mcl.py`) and the robot and particle modeling module (`robot.py`).

A. *MCL Solver*

The MCL module implements the core Monte Carlo Localization algorithm with two fundamental operations: weight update and particle resampling.

Weight Update: The weight of a particle evaluates how well each particle's sensor measurement matches the real robot's measurement. We apply a Gaussian likelihood function with Exponential Moving Average (EMA) to compute the weight:

$$w_i = (1-\alpha) \cdot \exp\left(-\frac{(d_i - d_{\text{real}})^2}{2\sigma^2}\right) + \alpha \cdot w_i^{\text{prev}}$$

where:

- w_i is the updated weight of the i-th particle.
- w_i^{prev} is the previous weight of the i-th particle.
- d_i is the distance measured by the i-th particle.
- d_{real} is the real (ground truth) distance measured by the robot.
- σ is the parameter which controls how tolerant the system is to measurement differences.
- $\alpha \in [0, 1]$ is the smoothing factor for the EMA.

The EMA was introduced because we observed a critical problem during early testing: when the robot made a turn near an obstacle, even a small orientation difference between a particle and the real robot can lead to one sensor ray hitting an obstacle while the other doesn't. This led to a large difference in the measurement, even if the particle was very close to the robot. As a result, such particles were given low weights and removed during resampling. The introduction of EMA helps preserve good particles that might otherwise be lost due to the measurement differences during turns.

Resampling Strategy: Resampling addresses the particle degeneration problem, i.e. most particles may have low weights and thus contribute little to the pose estimation. By generating a new set of particles, resampling helps maintain an accurate and robust estimate of the robot's pose over time. We determine whether to perform resampling using the effective sample size (N_{eff}), defined as:

$$N_{\text{eff}} = \frac{1}{\sum_i w_i^2},$$

where w_i is the normalized weights of the i-th particle. Resampling is triggered if N_{eff} falls below some threshold, usually a proportion of the number of particles $\beta \cdot N$. To balance estimation accuracy and particle diversity, we adopt a hybrid resampling strategy that combines two methods, controlled by a tunable parameter p:

Algorithm 2: Hybrid Resampling Strategy

Input:
- Particle pose set $\{x_1, \ldots, x_N\}$
- Particle weights $\{w_1, \ldots, w_N\}$
- Random sampling probability p

Output: Resampled particle set X'

```
Normalize the weights;
Compute CDF of normalized weights;
Draw s ~ U(0, 1/N);
for i ← 0 to N − 1 do
    Draw r ~ U(0, 1);
    if r < p then
        Sample particle pose x uniformly from valid
         configuration space;
        Add x to resampled set X';
    else
        u ← s + i/N;
        Find smallest j such that u ≤ CDF_j;
        Add particle x_j to resampled set X';
    end
end
return X'
```

- Low Variance Resampling (LVR): Applied with probability $(1-p)$. The algorithm generates N equi-distant sample points $\frac{i}{N} + s$ where $s \sim U(0, \frac{1}{N})$, $i = 0, 1, \ldots, N-1$, ensuring better coverage of the weight distribution compared to naive random sampling.
- Random Resampling: Applied with probability p. It uniformly samples new particles from the valid configuration space, helping the system recover from localization failures and maintaining particle diversity.

The full procedure is detailed in Algorithm 2.

B. *Robot and Particle*

The Robot class contains two major functions: moving the robot given the linear velocity v and angular velocity ω commands, and simulating the distance measurement.

Action Model: The Robot class implements an action model using differential drive kinematics with noise modeling.

- Linear Model ($\omega \approx 0$): When the angular velocity is close to 0, the robot follows a linear trajectory:

$$x' = x + v \cdot \cos\theta \cdot \Delta t$$

$$y' = y + v \cdot \sin\theta \cdot \Delta t$$

$$\theta' = \theta$$

- Curved Model ($|\omega| \geq 10^{-5}$): For non-zero angular velocities, the robot follows a circular arc trajectory:

$$x' = x + \frac{v}{\omega}\left(\sin(\theta + \omega \cdot \Delta t) - \sin(\theta)\right)$$

$$y' = y - \frac{v}{\omega}\left(\cos(\theta + \omega \cdot \Delta t) - \cos(\theta)\right)$$

$$\theta' = (\theta + \omega \cdot \Delta t) \bmod 2\pi$$

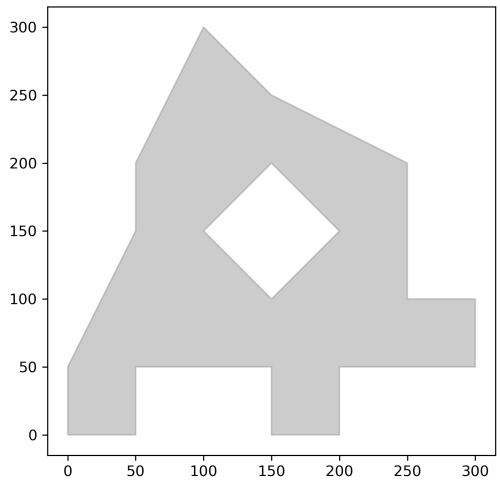

Fig. 2. Map 1: An irregular room with an obstacle at the center.

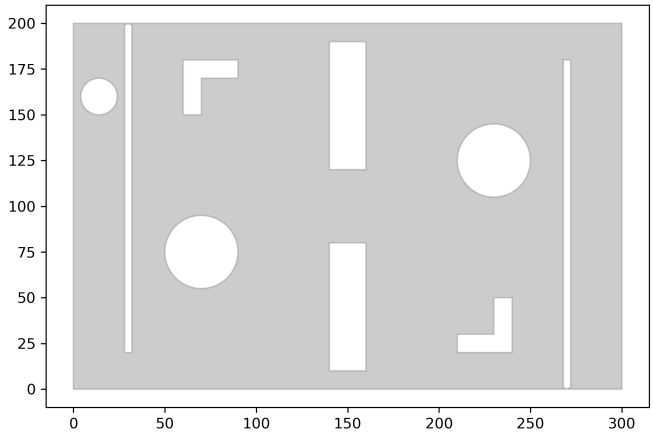

Fig. 3. Map 2: A central symmetric room except 2 different corridors.

- Motion Noise Modeling: The action model incorporates two sources of uncertainty to simulate real-world actuator imperfections:
  1) Linear velocity noise is applied when $|v| > 10^{-5}$:
  $$\tilde{v} = v + \mathcal{N}(0, \sigma_v^2)$$
  2) Angular velocity noise is applied when $|w| > 10^{-5}$:
  $$\tilde{w} = w + \mathcal{N}(0, \sigma_w^2)$$
  The noise is only added when the corresponding velocity component is significant, preventing unnecessary noise during stationary or pure translational states.

Distance Measurement: The distance sensor is simulated using a ray-casting approach based on Shapely's geometric operations. Specifically, the sensor ray is intersected with the world boundaries to compute the true distance measurement d. To account for sensor uncertainty, Gaussian noise is then added to the result: $\tilde{d} \sim \mathcal{N}(d, \sigma_m^2)$.

Efficient Particle Group Operations: Logically, the individual Particles are analogous to separate Robot instances. However, it is highly inefficient to maintain them as a list of Robot objects. This is primary because Shapely is designed for geometry manipulation but not for large-scale computation. To overcome this performance bottleneck, we optimized the computational logic within the ParticleGroup class by using Numpy for vectorized operations.

The formula in action model are naturally compatible with NumPy arrays. However, simulating distance measurements poses a particular challenge. The vectorized solution to this problem involves solving a parameterized equation, with calculations accelerated by computing cross products in a vectorized manner.

To calculate the intersection of two lines, then the two lines can be expressed in parameterized form:

$$\mathbf{p}(t) = \mathbf{p}_0 + t \cdot \mathbf{r}, \quad \text{where } \mathbf{r} = \mathbf{p}_1 - \mathbf{p}_0,$$

$$\mathbf{q}(u) = \mathbf{q}_0 + u \cdot \mathbf{s}, \quad \text{where } \mathbf{s} = \mathbf{q}_1 - \mathbf{q}_0,$$

and the intersection is given by (if $\mathbf{r} \times \mathbf{s} \neq 0$):

$$t = \frac{(\mathbf{q}_0 - \mathbf{p}_0) \times \mathbf{s}}{\mathbf{r} \times \mathbf{s}},$$

$$u = \frac{(\mathbf{q}_0 - \mathbf{p}_0) \times \mathbf{r}}{\mathbf{r} \times \mathbf{s}},$$

where "$\times$" denotes the cross product in 2D. The intersection is valid if $t \in [0, 1]$ and $u \in [0, 1]$. The distance measurements for all particles is calculated simultaneously by utilizing the broadcasting mechanism of Numpy.

With this optimization, we halved the rendering time compared to using Shapely functions with a for loop.

IV. Produced Example

We produced two examples demonstrating how MCL works on two different maps shown in Figure 2 and Figure 3.

A. Map 1: Impact of Measurement Noise

Experimental Configuration: We tested the relationship between sensor uncertainty and localization convergence by conducting MCL simulations on Map 1 with varying measurement noise levels. The experimental parameters were configured as follows:

- Number of particles: 5000
- Likelihood σ: 3
- Noise of measurement σ_m: $\{0, 5, 10, 30\}$
- EMA factor α: 0.66
- Linear velocity noise σ_v: 0.1
- Angular velocity noise σ_ω: 0.05
- Resample threshold factor β: 0.5
- Random resample probability p: 0.2
- Random seed: 0
- Simulation steps: 1600

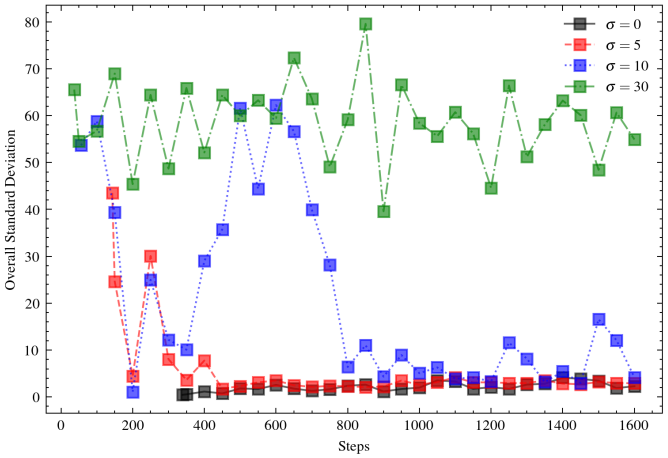

Fig. 4. Standard deviation throughout the simulation.

Convergence Evaluation Method: During the simulation, we tracked the survival times of particles through resampling cycles. Particles that survived 10 or more resampling events were labeled as *high-probability particles*, indicating their consistent alignment with sensor observations. To evaluate the convergence quality, we calculated the standard deviation of high-probability particle positions (x, y) using:

$$\text{std} = \sqrt{\frac{\text{var}(x) + \text{var}(y)}{2}}.$$

Results and Analysis: The trend of standard deviation throughout the simulation is shown in Figure 4. For clarity, we only plot the values at every 50 steps to improve visual readability. The first dot for each noise level indicates when high-probability particles first appeared in the system. An interesting observation is that higher measurement noise levels led to earlier appearance of high-probability particles. This is reasonable, as a larger noise level increases the likelihood that particles further from the true position can still produce sensor readings similar to the actual measurement, thus receiving relatively high weights.

The results demonstrate that MCL exhibits robust convergence behavior under mild measurement noise conditions ($\sigma_m = 0, 5, 10$), as evidenced by the decreasing standard deviation trends in Figure 4. However, excessive noise ($\sigma_m = 30$) disrupts the convergence process, leading to sustained high variance in particle positions.

Figure 5 shows the particle distributions at simulation step 1570, clearly illustrating the convergence outcomes. The scenarios with $\sigma_m = 0, 5$ and 10 achieved successful localization with well-concentrated particle clusters around the true robot position. In contrast, the high-noise case ($\sigma_m = 30$) failed to converge, maintaining a diffuse particle distribution that provides little localization confidence.

B. Map 2: The Ambiguity in Symmetric Environment

We designed a centrally symmetric map, as shown in Figure 3. The environment features a symmetric arrangement of obstacles, except for two corridors on the left and right:

Fig. 5. The particle distribution at simulation step 1570.

the left corridor contains a circular obstacle, whereas the right one does not.

Experimental Configuration: The experimental parameters were configured as follows:

- Number of particles: 5000
- Likelihood σ: 3
- Noise of measurement σ_m: 0
- EMA factor α: 0.66
- Linear velocity noise σ_v: 0.1
- Angular velocity noise σ_ω: 0.05
- Resample threshold factor β: 0.6
- Random resample probability p: 0.2
- Random seed: 123
- Simulation steps: 1000

Result and Analysis: During navigation within the symmetric region, the particle filter could identify two likely positions for the robot: the true location and its symmetric counterpart, due to the identical sensor readings received at symmetric locations (ignoring measurement noise). Notably, this behavior is not guaranteed. Sometimes, particles converge only to one position before entering one of the asymmetric corridor.

Once the robot enters one of the asymmetric corridor sections, incorrect particles begin to diverge from the true robot state, as their sensor readings no longer match the actual observations. After a few resampling steps, these inconsistent particles are eliminated, and the filter converges to the correct position.

Figures 6 and 7 show the particle distributions at step 500 and step 1000, respectively, illustrating the situation before and after the robot entered the corridor.

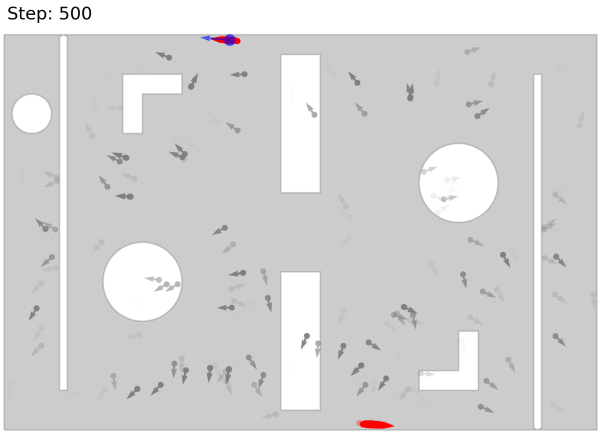

Fig. 6. The particle identify two possible positions for the robot when exploring the symmetric region at simulation step 500.

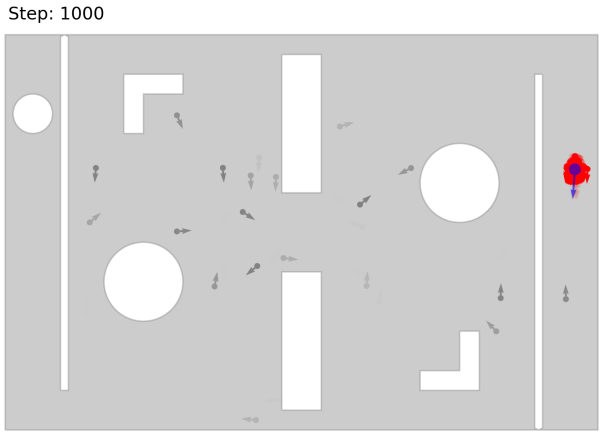

Fig. 7. The particle distribution after entering one of the asymmetric corridor at simulation step 1000.

V. Summary

In this project, we built a complete Monte Carlo Localization (MCL) simulation system, including map and obstacle creation, action and observation models for both the robot and particles, the MCL algorithm with weight updating and resampling, and a visualization pipeline that clearly presents the wonderful process of localization.

During the developing, we found that the core MCL is actually simple and can be implemented in around 100 lines of Python code (including comments). However, building a full simulation system posed more significant challenges. We carefully designed the system architecture to be modular and logically organized, allowing us to focus on individual components without being overwhelmed by complex data flow or dependencies.

The system is also highly efficient. We optimized particle operations using NumPy to vectorize large-scale computations like updating weights, computing new positions, and measuring the distances. For example, the noise-level experiment took about 20 minutes to complete, and the symmetric map experiment took about 12 minutes. Simulations with fewer particles would run even faster.

Overall, this project gave us an intuitive and deep understanding of how MCL works in practice. We believe that the system could also serve as a valuable educational tool for others learning about this topic.

The full source code and instructions are available on GitHub. We also provide complete videos of the two experiments described in Section IV, available here.