# Lab 2: Monte Carlo Localization

CSCI 545

Team Roomba
Brandon Rathburn, Devayani S Krishnan, Lakshya Ahuja, Bilin Pattasseril, Rut Patel

Spring 2024

## 1 Introduction

In this lab, we will implement a localization system for a mobile robot with a planar LIDAR range sensor. Localization systems are important elements of all robots that need to act. Before you can choose actions, you need to know where you are relative to your environment.

There are many types of localizers. Some use RGB camera sensors and attempt to recognize the viewpoint from which they are seeing a scene, which remains a difficult problem in the general case. Others look at pre-crafted features in the environment, like ArUco tags, which are easy to detect and placed in known locations. These systems are robust and reliable for controlled environments like factories, but infeasible for uncontrolled environments (like most of the world).

We will implement a common type of localizer, called a Monte Carlo Localizer, which is based on a probabilistic filter called a particle filter. Our sensory input will be a dense planar LIDAR scan of our surroundings, and we will try to localize against a map generated with the same kind of sensor. Using this sensory information, and integrating it over time, we can gain an accurate probabilistic understanding of where our robot is relative to an existing map.

## 2 Theory

### 2.1 Localization

A localizer is a system that takes a series of actions $\mathbf{u}_t$, observations $\mathbf{z}_t$ and a map $m$ and tries to estimate the global pose $\mathbf{x}_t$ of the robot relative to the map.

### 2.2 Particle Filters

A particle filter is a nonparametric Bayesian filter, which approximates the posterior distribution $bel(\mathbf{x}_t)$ using a finite number of parameters, called particles. We call our collection of particles $X_t = \{\mathbf{x}_t^{(1)}, \ldots, \mathbf{x}_t^{(n)}\}$, where each particle $\mathbf{x}_t^{(i)}$ represents a sample drawn from the posterior. More simply, each particle is a concrete hypothesis of the global pose of the robot.

Let's examine the algorithm for a particle filter line by line, specifically focused on the case of localization.

**Algorithm 1:** Particle Filter

**input :** $X_{t-1}$, $\mathbf{u}_t$, $\mathbf{z}_t$
**output:** $X_t$

1 $\overline{X}_t = X_t = \emptyset$;
2 **for** $i = 1 \to n$ **do**
3 | Sample $\mathbf{x}_t^{(i)} \sim p(\mathbf{x}_t | \mathbf{u}_t, \mathbf{x}_{t-1}^{(i)})$;
4 | $w_t^{(i)} = p(\mathbf{z}_t | \mathbf{x}_t^{(i)})$;
5 | Add $\{(\mathbf{x}_t^{(i)}, w_t^{(i)})\}$ to $\overline{X}_t$;
6 **end**
7 **for** $i = 1 \to n$ **do**
8 | Draw $i$ with probability $\propto w_t^{(i)}$;
9 | Add $\mathbf{x}_t^{(i)}$ to $X_t$
10 **end**

From the input and output, we can immediately see that the particle filter is a recursive Bayesian filter, that is, it takes a current estimate of the state, along with some new information, and produces a new state estimate. Recursive filters are often simpler to implement and more memory efficient than filters that require an entire history of sensory inputs.

Line 3 applies the *motion model* to each pose hypothesis $\mathbf{x}_t^{(i)}$. Given some motion information $\mathbf{u}_t$ from the robot, we change each pose by moving it in accordance with our understanding of how the robot moves. More information on motion models is presented in Section 2.3 below.

Line 4 calculates an importance weighting for each particle based on the *sensor model*. A sensor model tells us how likely the given measurement $\mathbf{z}_t$ is from each hypothesis $\mathbf{x}_t^{(i)}$, given our known map $m$. More information on sensor models is presented in Section 2.4 below.

Lines 8 and 9 take the updated hypotheses from the sensor and motion models and probabilistically resample the particles to match the importance weight of each sample.

After running this algorithm once, we see that our particles $X_t$ approximate the new posterior $p(\mathbf{x}_t | \mathbf{z}_t, \mathbf{u}_t, \mathbf{x}_{t-1})$.

For a more complete presentation of the general particle filter, including a mathematical derivation of its properties as an estimator and discussion on sampling bias and particle deprivation problems, please refer to Thrun's *Probabilistic Robotics* Section 4.3.

## 2.3 Motion Models

In this lab, we will consider a *velocity* motion model. (An alternative is an *odometry* motion model.)

A motion model specifies the probability distribution $p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{u}_t)$. That is, given our current pose and some control input, we can predict the likelihood of ending up in various next states. For the particle filter algorithm, however, we don't need to compute the exact probabilities over the state space, we just need to be able to draw samples from this posterior.

Odometry is a measure of how far a robot has driven. Oftentimes, a rotary encoder is attached to the wheel axle of a robot, and the ticks are integrated per wheel to get estimate the position of the robot. Understanding the relationship between the rotary encoder and the pose of the robot requires knowledge of the robot's *kinematics*, i.e. how large each wheel is, and where the wheels are relative to each other.

Note that odometry is a *sensor* measurement, while a motion model integrates *control* input. We work around this by treating a pair of odometry-estimated poses as a control input $\mathbf{u}_t = (\mathbf{x}_t, \mathbf{x}_{t-1})$. Practically, this is really the difference between these two poses $\Delta\mathbf{x} = (\Delta x, \Delta y, \Delta\theta)$.

Odometric sensors using rotary encoders cannot observe many common types of error, such as wheel slippage. Imagine a robot with one wheel resting on ice. The wheel may turn, and we will integrate information as robot motion, but in reality the robot has not moved because of the slippery surface.

To model uncertainty caused by sensor noise and possible slip, we perturb each dimension of the change in pose by a zero-mean Gaussian $\epsilon \sim \mathcal{N}(0, \sigma^2)$ with variance $\sigma^2$ a tunable parameter. By sampling the perturbation in each dimension, we can sample a new pose from the motion-model posterior

$$\mathbf{x}_t \sim p(\mathbf{x}_t | \mathbf{x}_{t-1}, \mathbf{u}_t)$$

or, more concretely,

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \Delta\mathbf{x} + \mathbf{w}_\mathbf{x}$$

where the noise $\mathbf{w}_\mathbf{x}$ is described as follows

$$\begin{aligned}
\mathbf{w}_\mathbf{x} &= (\mathbf{w}_x, \mathbf{w}_y, \mathbf{w}_\theta) \\
\mathbf{w}_x &\sim \mathcal{N}(0, \sigma_x^2) \\
\mathbf{w}_y &\sim \mathcal{N}(0, \sigma_y^2) \\
\mathbf{w}_\theta &\sim \mathcal{N}(0, \sigma_\theta^2)
\end{aligned}$$

## 2.4 Sensor Models

In our context, each observation $\mathbf{z}_t = \{\mathbf{z}_t^{(1)}, \mathbf{z}_t^{(2)}, \ldots, \mathbf{z}_t^{(n)}\}$ is a planar LIDAR scan of $n$ rays, where each ray $\mathbf{z}_t^{(i)} = (r_t^{(i)}, \theta_t^{(i)})$ consists of a measured distance $r_t^{(i)}$ and an angle from the sensor frame $\theta_t^{(i)}$.

Given the map $m$, these observations become conditionally independent, so our posterior can be broken apart as follows

$$p(\mathbf{z}_t | \mathbf{x}_t, m) = \prod_{i=1}^{n} p(\mathbf{z}_t^{(i)} | \mathbf{x}_t, m)$$

For a particular ray $\mathbf{z}_t^{(i)}$, we can simulate a corresponding ray $\hat{z}_t^{(i)} = (\hat{r}_t^{(i)}, \theta_t^{(i)})$ given a pose (or pose hypothesis) $\mathbf{x}_t$, and a map $m$ by raycasting in the map until we hit an obstacle. We can get this distance by using Bresenham's algorithm, or a similar line-tracing algorithm, along the discretised occupancy grid which represents our map.

We will use a simplified sensor model which models the probability of a laser hit using a single dimensional Gaussian centered on the simulated laser return. Given a particle hypothesis $\mathbf{x}_t$, we know what each ray of a simulated scan would look like. We can thus compute, per ray, the probability of our actual measurement $\mathbf{z}_t^{(i)}$ as

$$p(\mathbf{z}_t^{(i)} | \mathbf{x}_t, m) = \mathcal{N}(r_t^{(i)}; \hat{r}_t^{(i)}, \sigma_{hit}^2) = \frac{1}{\sqrt{2\pi\sigma_{hit}^2}} e^{-\frac{\left(r_t^{(i)} - \hat{r}_t^{(i)}\right)^2}{2\sigma_{hit}^2}}$$

The variance of this distribution, $\sigma_{hit}^2$, is an intrinsic tunable noise parameter which encompasses sensor error and map error.

In practice, beam sensors like LIDAR have more complex models. See *Probabilistic Robotics* Section 6.3 for a detailed discussion of other sources of discrepancy from map simulated returns, including unexpected objects, failures, and special cases at the maximum range of the sensor.

### 2.5 Monte Carlo Localization

After integrating information through the particle filter, we now have *almost* enough information to localize a robot. Note that $X_t$ now gives a collection of samples approximating the posterior $bel(\mathbf{x}_t)$. However, to plan and act using a robot, we need a concrete guess as to the robot's state.

We can concretize the distribution in a simple way by assuming it is unimodal, and taking the average pose of all particles in the filter

$$\hat{\mathbf{x}}_t = \left( \sum_{i=1}^{n} \frac{x_t^{(i)}}{n}, \sum_{i=1}^{n} \frac{y_t^{(i)}}{n}, \sum_{i=1}^{n} \frac{\theta_t^{(i)}}{n} \right)$$

# 3 Implementation

a. First, login to your GitHub account and accept this assignment (click your group name on the list if exist, otherwise type in your group name manually).
`https://classroom.github.com/a/MtFiFZNN`

b. You will write a single ROS node in Python, called `mcl545_node`. The skeleton code will open a pre-recorded bag file for you and subscribes to odometry and scan information for you.

c. Search the code in `catkin_ws/src/usc545mcl/bin/usc545mcl.py` for 3 comments titled `YOUR CODE HERE` for implementation instructions. This is all of the code you will need to write.

d. Use the same **Docker image and Docker Container** from **Lab 1**.

e. In your terminal 1, enable access to the X11 display server on Linux systems

```
xhost +
```

f. In your terminal 2, run the container, and then run

```
cd ~/catkin_ws
```

```
sudo ./setup.sh
```

```
catkin_make
```

g. To execute your lab code, still in terminal 2, go to the `catkin_ws` directory, run

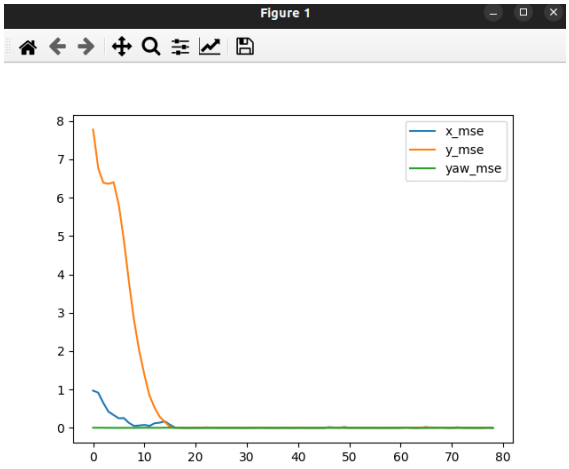
```
source devel/setup.bash
```

```
roslaunch usc545mcl usc545mcl.launch
```

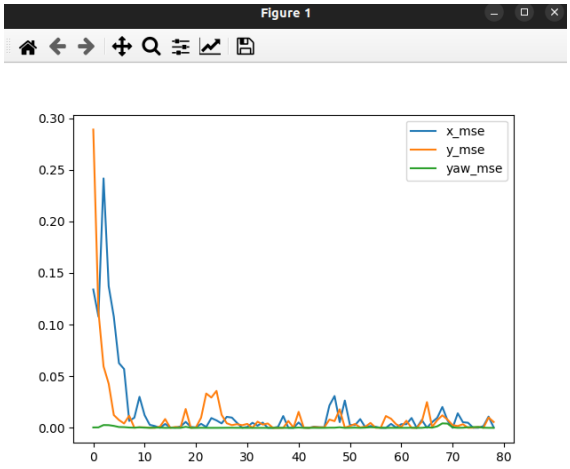
# 4 Output

Answer the following questions in 3-5 sentences each in a lab document.

1. **How does each of the model variance parameters affect the performance of the localizer?** Increasing the LINEAR_MODEL_VAR_X and LINEAR_MODEL_VAR_Y parameters makes the localizer becomes less confident in its motion model and more reliant on sensor input, which can lead to faster convergence because the filter quickly adjusts to new measurements. However, with low variance, the localizer trusts its internal model more, which can cause fluctuations if the model doesn't perfectly represent the real motion or if there is noise. Raising ANGULAR_MODEL_VAR reduces confidence in yaw predictions, while lowering it increases certainty in heading. Increasing SENSOR_MODEL_VAR causes the localizer to rely less on LIDAR and more on internal predictions, whereas lowering it makes the filter highly responsive to LIDAR, improving correction speed but risking sensitivity to sensor noise



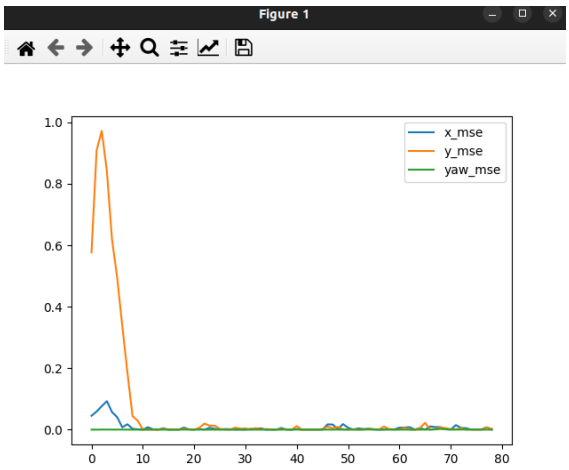

**(a) High Variance**



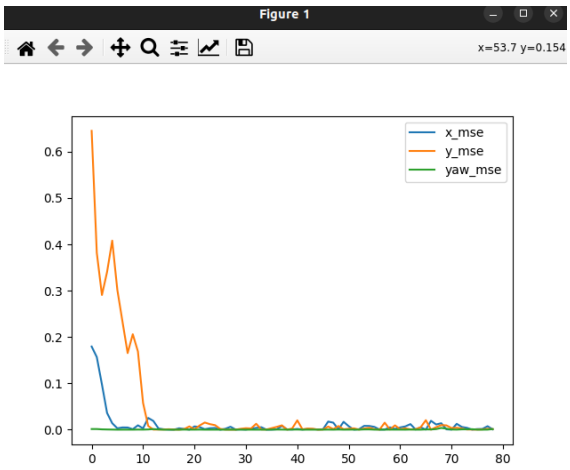

**(b) Low Variance**

*Comparison of high vs. low variance.*

2. **How does the number of particles affect the behavior of the localizer?** The number of particles plays a crucial role in how accurately the filter approximates the robot's true pose. Increasing the number of particles improves accuracy and robustness by representing a wider range of possible locations, but it also raises computational cost and slows down updates. In our experiments, using 10,000 particles took roughly five times longer to compute compared to baseline 2,000 particles. However, the results were significantly smoother and more stable. With only 100 particles, the filter showed noticeable fluctuations and less consistent localization, whereas the 10,000 particle case produced a much steadier line/convergence.




**(a) 10,000 Particles**

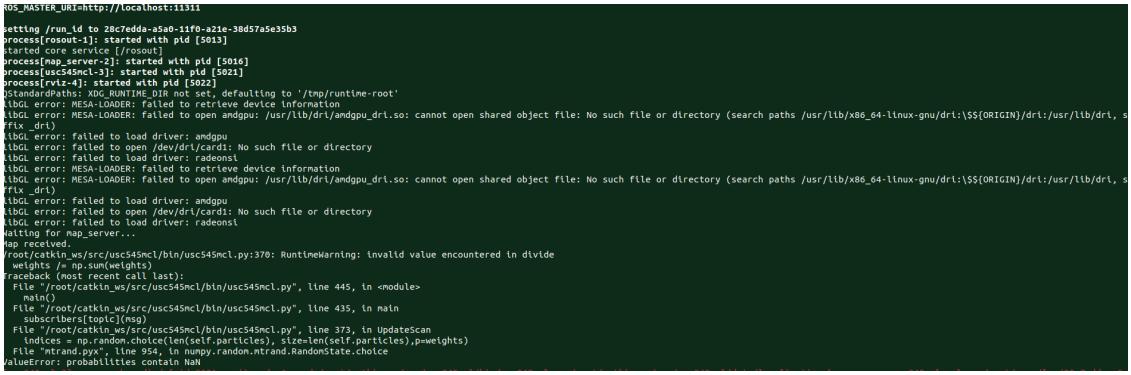



**(b) 100 Particles**

*Comparison of 10k particles vs. 100 particles.*

3. **Can a particle filter with a single particle perform well? Why or why not? What if it starts in the correct position?** A particle filter with only one particle generally performs poorly because it cannot capture uncertainty or recover from localization errors. If that single particle drifts away from the true pose due to noise, the filter has no mechanism to correct itself. When we tested this

setup, we encountered an error resulting in NaN probabilities, highlighting this limitation. However, if the particle starts precisely at the correct position and no noise is present, it can track motion accurately.




**Error with NaN probabilities**

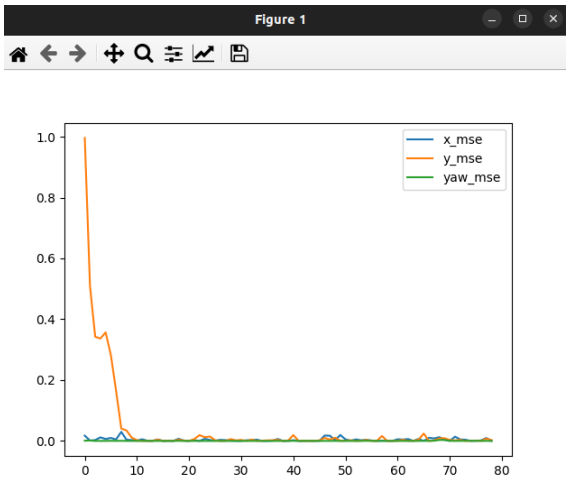
Include the error graph generated by the localizer in your writeup document.

# 5 Extra Credit

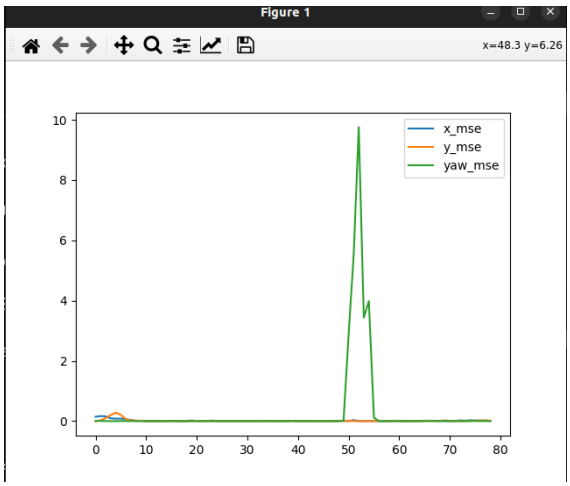
- **The noise model for the motion model presented in Section 2.3 of this document is overly simplistic. You can find a better noise model in Section 5.4 of *Probabilistic Robotics*. Explain why the noise model presented in this lab has poor properties. Implement the better noise model.**

  The noise model described in Section 2.3 is overly simplistic because it adds a fixed Gaussian variance that does not adapt based on the robot's motion. This approach ignores the fact that translational and rotational noise are often correlated with the actual movement of the robot. As a result, the model produces less realistic and less robust predictions.

  In contrast, the improved noise model from Section 5.4 of *Probabilistic Robotics* more accurately reflects real-world motion uncertainty by accounting for movement-dependent errors. As shown in the figures below, the YAW_MSE increases around the 50–60 time mark, indicating that the improved model successfully explores a new region that the simple model missed. This exploration helps the filter recover and converge faster. The MSE plot also shows that the improved model converges to zero more quicker than the simple one. Another thing we can see the inital Y_MSE is less and start from 0.




**(a) Section 2.3: Simple Noise Model**




**(b) Section 5.4: Improved Noise Model**

*Comparison of simplistic vs. improved noise models for robot motion.*

- **There are more components involved in an accurate beam sensor model. They can be found in Section 6.3 of *Probabilistic Robotics*. Implement the complete beam sensor model from this section.**
- **The particle filter is powerful because it can model complex, multimodal posterior distributions. Averaging all particles assumes a unimodal distribution and may yield unrealistic estimates. Use a clustering algorithm to estimate the particles in the largest mode and compute their average as a more realistic estimate to concretize the posterior.**

# 6 Additional Questions

As a very rough guideline, we anticipate that for most teams, the answers to each question below will be approximately one paragraph long (or about 4-5 bullet points). However, your answers may be shorter or longer if you believe it is necessary.

## 6.1 Resources Consulted

**Question:** Please describe which resources you used while working on the assignment. You do not need to cite anything directly part of the class (e.g., a lecture, the CSCI 545 course staff, or the readings from a particular lecture). Some examples of things that could be applicable to cite here are: (1) did you get help from a classmate *not* part of your lab team; (2) did you use resources like Wikipedia, StackExchange, or Google Bard in any capacity; (3) did you use someone's code (again, for someone *not* part of your lab team)? When you write your answers, explain not only the resources you used but HOW you used them. If you believe your team did not use anything worth citing, *you must still state that in your answer* to get full credit.

**Answer:** We used ChatGPT to better understand the context of the problem. For instance, during the importance sampling section, we struggled to fully grasp the underlying concept, so using ChatGPT to clarify the steps and criteria was extremely helpful. We also used it to decide on an appropriate clustering algorithm for the Extra Credit 3 problem. Another part we used it for was to better understand how to choose the alpha value to optimize performance, especially for extra credit 1.

## 6.2 Team Contributions

**Question:** Please describe below the contributions for each team member to the overall lab. *Furthermore, state a (rough) percentage contribution for each member.* For example, in a team of 4, did each team member contribute roughly 25% to the overall effort for the project?

**Answer:** There were three main sections of code to be implemented in Python (in the usc545mcl.py file). Each group member focused on a specific part: Brandon worked on the motion model, Devayani implemented the sensor model, and Bilin handled the importance sampling section. Rut contributed to Extra Credit 1 (implement better noise model) and Extra Credit 3 (implement a better particle filter). Finally, Lakshya worked on Extra Credit 2 (implementing a more accurate beam sensor model). Brandon Rathburn 20Devayani S Krishnan 20Lakshya Ahuja 20Bilin Pattasseril 20Rut Patel 20