CSCI545 Monte Carlo Localization Lab

Deadline: Oct. 13th 11:59pm PST

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.

Lets 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. The alternative, an *odometry* motion model, was presented in class. Please refer to your lecture notes for the difference between the two.

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 σ^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, σ_{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

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.

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.

(Mac M1/M2 users should follow a different set of procedures outlined in the subsection below) Before you can execute your code, `cd` to the directory into which you cloned your repo, and then (you only need to do this once)

```
sudo ./setup.sh
cd catkin_ws
catkin build
```

then, to execute your lab code, still in the `catkin_ws` directory, run

```
source devel/setup.bash
roslaunch usc545mcl usc545mcl.launch
```

3.1 For Mac M1/M2 users

Please copy the `/usc545mcl` folder directly into `/home/csci545/ros_ws/src`. Do not run `sudo ./setup.sh`. Instead run `cd /home/csci545/ros_ws`, and then `catkin build`. Procedures for executing the lab code remains the same. Remember to give `usc545mcl.py` execute permission by running `cd $(rospack find usc545mcl)/bin && chmod +x usc545mcl.py`.

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? Give possible reasons for the behavior you see. The parameters are defined at the top of the file.
2. How does the number of particles affect the behavior of the localizer? The number of particles can be modified at the top of the file.
3. Can a particle filter with a single particle perform well? Why or why not? What if it starts in the correct position?

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.
- 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 very 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.