Mobile Robotics and Autonomous Driving, Fall 2020

Homework 3 - Robot Localization using Particle Filters

Due: Tuesday September 15, 11:59pm, 2020

Your homework should be submitted as a **typeset PDF file** along with a folder including **code only (no data)**. The homework packet can be downloaded from **Moodle**, and your solution must be submitted there. If you have questions, please post them on **Telegram**. Please do not post solutions or codes on Telegram. Discussions are allowed, but each group must write and submit their own, original solution. Note that you should list the name and Innopolis IDs of each student you have discussed with on the first page of your PDF file. You have a total of 3 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 some random hall with nothing but odometry and a laser rangefinder. Fortunately, you have a map of random 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 $x_{t-1}^{[m]} \sim p(x_t|u_t, x_{t-1}^{[m]})$. The proposal distribution here is the motion model, $p(x_t|u_t, x_{t-1})$.

2. Correction Step: Computing an importance weight $x_{t-1}^{[m]}$ for each particle as the ratio of target and proposal distributions. This reduces to computing weights using the **sensor model**, that is $w_{t-1}^{[m]} = p(z_t|x_t^{[m]}, \boldsymbol{M})$.

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 $\boldsymbol{M}$ is the occupancy map. $x_t^{[m]}, w_t^{[m]}$ is the robot pose and importance weight of particle m at time t.

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)
  $w_t^{[m]} = p(z_t \mid x_t^{[m]})$ (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 $\mathcal{X}_t$

2. Monte Carlo Localization

Monte Carlo Localization (MCL), a popular localization algorithm, is essentially the application of particle filter 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.

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. 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 that 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).

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 real-time-ness requirement, although it is advisable to use something faster than MATLAB. Feel free to utilize the techniques that we have discussed in class as well as extensions discussed in [1] or elsewhere. You would 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:

- `instruct.txt` – Format description for the map and the data logs.
- `robotdataN.log.gz` – Five data logs (odometry and laser data).
- `wean.dat.gz` – Map of Wean Hall to use for localization.
- `wean.gif` – Image of map (just for your information).
- `bee-map.c` – Example map reader from BeeSoft that you may use if desired.
- `robotmovie1.gif` – Animation of data log 1 (just for your information).

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.
- `MapReader.py` – Reads in the Wean Hall map (`wean.dat`) and computes and displays corresponding occupancy grid map.
- `MotionModel.py, SensorModel.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.

Improving Efficiency

Although there is no real-time-ness requirement, the faster your code, the more particles you will be able to use feasibly and faster would be your parameter tuning iterations. 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're comfortable with C++, implementing your particle filter in that would be much faster!

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 would 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—you're 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, 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.

Score breakup

(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

(20 points) Write-up quality, video quality, readability, description of performance, and future work

(Optional Extra Credit: 10 + 10 points) Kidnapped robot problem and Adaptive number of particles

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?

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.

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++ vs. using the Python skeleton code.

References

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