16-833: Robot Localization and Mapping, Spring 2021

# Homework 1

# Robot Localization using Particle Filters

---

**Ashwin Nehete (anehete) , Harsh Dhruva(hdhruva)**

---

## Table of contents

# 1. Introduction

This homework assignment explores robot localization with particle filters, also known as *monte carlo localization*. Particle filter is a tool to track the state of a given dynamic system which considers an alternative non-parametric implementation of the Bayes filter. We implement a global localization filter for a lost indoor mobile robot operating in Wean Hall with nothing but odometry and a laser rangefinder and a map of Wean Hall. We observe a reasonable localization performance as demonstrated in the appropriate videos whose YouTube links have been attached in the results section for 3 different log files.

# 2. Implementation

## 2.1 Particle Initialization

We started out initializing particles in the entire map, however, you would require a lot more particles to have a fair amount of particles in the corridor areas of the map. Therefore, we implemented init_particles_freespace, assuming that the robot is located in the corridor areas of the hallway. The initialization of the particles is shown in Figure 1.

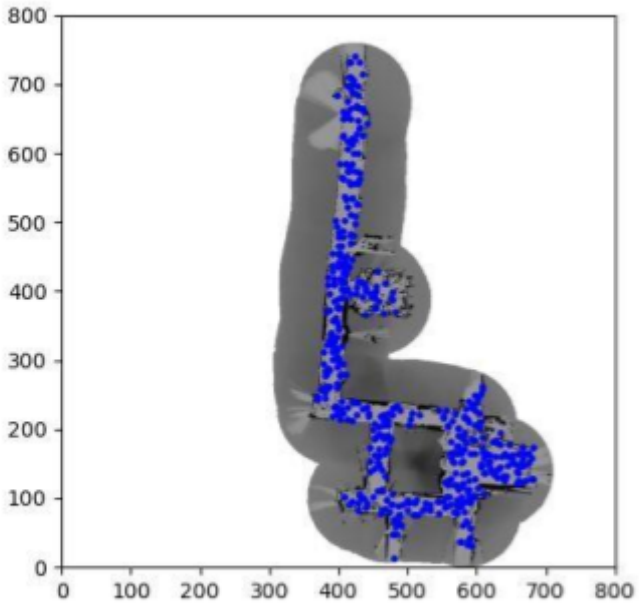


Figure 1: Particle initialization in the freespace

## 2.1 Motion Model

For our localization task, we use the Odometry motion model as described in Thrun et al. [1]. It uses relative motion information in time (t-1, t] to update the robot state from $\overline{X_1}$ = ($x_1$, $y_1$, $\theta_1$) to $\overline{X_2}$ = ($x_2$, $y_2$, $\theta_2$). Here, $\overline{X_1}$ *and* $\overline{X_2}$represent the odometry measurements embedded in the robot internal coordinate frame, and the relation to global coordinates is unknown. We extract relative odometry information by transforming $\overline{X_1}$ *and* $\overline{X_2}$into three steps, a rotation $\boldsymbol{\delta}_{rot1}$, a translation $\boldsymbol{\delta}_{trans}$ and another rotation $\boldsymbol{\delta}_{rot2}$. This is illustrated in Figure 2.

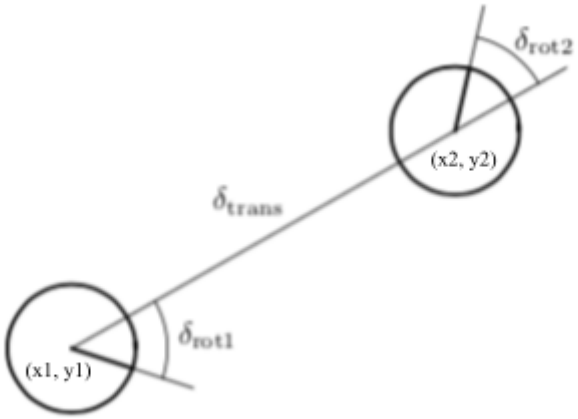


Figure 2: Odometry Model

The initial rotation can be calculated as,

$$\boldsymbol{\delta}_{rot1} = \text{atan2}(y_2 - y_1,\ x_2 - x_1) - \theta_1$$

The distance between the consecutive odometry reading is,

$$\boldsymbol{\delta}_{trans} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The final rotation is calculated as,

$$\boldsymbol{\delta}_{rot2} = \theta_2 - \theta_1 - \boldsymbol{\delta}_{rot1}$$

We incorporate robot specific gaussian noise parameters $\boldsymbol{\alpha}_1$, $\boldsymbol{\alpha}_2$, $\boldsymbol{\alpha}_3$, $\boldsymbol{\alpha}_4$ in the model, that specify noise in the motion. After playing with the parameter values, we set our rotation parameters $\boldsymbol{\alpha}_1$, $\boldsymbol{\alpha}_2$ to 0.0001 to allow for low noise in rotation and set $\boldsymbol{\alpha}_3$, $\boldsymbol{\alpha}_4$ to 0.01 to allow for relatively high noise in translation. Reducing rotational noise relative to translational noise, worked best for us,

as we observed keeping the values the same (0.01 for both sets) increased the motional error in the system, and too much noise in the system affected the convergence.

Table 1: Parameters for motion model

| No. | Symbol | Value |
|---|---|---|
| 1 | $\alpha_1$ | 0.0001 |
| 2 | $\alpha_2$ | 0.0001 |
| 3 | $\alpha_3$ | 0.01 |
| 4 | $\alpha_4$ | 0.01 |

### 2.1.1 Testing

We tested the motion model, after our sensor model was able to converge to a ballpark location. When starting with no noise in the parameters, the motion directly mimicked the odometry data, which caused drift in the motion. Once our sensor model enabled the particles to converge to a particular location, we were able to tweak the motion model parameters to get a better understanding of the robot trajectory in the map. When noise was too high for both sets of parameters, the location would converge, however the particles would not move around through the corridor. All our successful logs were generated with the parameters listed in the table above.

## 2.2 Sensor Model

The sensor model will model the noise in the sensor measurements, returning the product of individual measurement likelihoods, which are set as the weights of the particles. It models the generation of sensor measurements in the real world. We implement the Beam Model of Range Finder as described in [1]. This model incorporates four types of measurement errors, small measurement noises, errors due to unexpected objects, errors due to failures to detect objects, and unexplained noise. Our model is a mixture of the four densities shown in Figure 3, where $z_t^{k*}$ is the true sensor value obtained through ray casting, which is the mean of the Gaussian that models the sensor measurement noise.

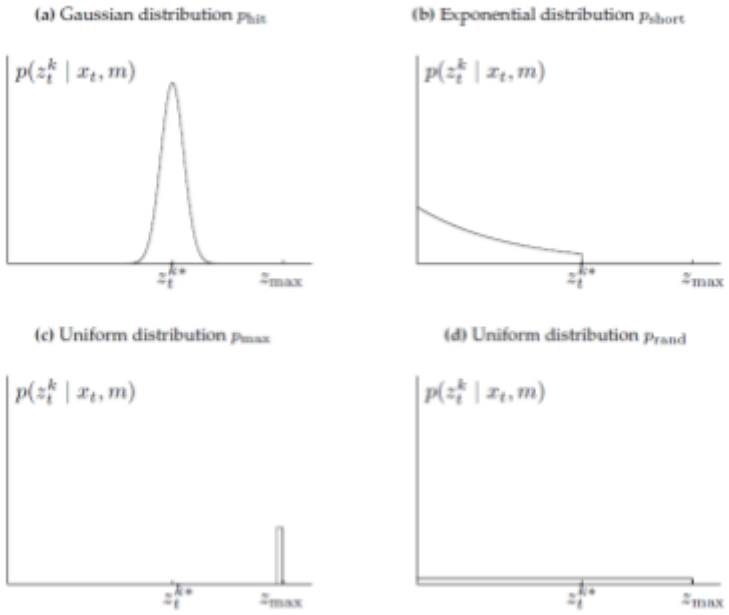


Figure 3: Components of beam range finder sensor model

Due to unexpected objects such as people in the way, the laser reading will be shorter than the true value $z_t^{k*}$, and the likelihood of sensing unexpected objects decreases with range. We also account for failures in the laser sensor when it does not detect a reading, and therefore assigns the maximum value measurement. Finally, random uncertainties need to be considered, which requires a uniform distribution over the entire measurement range, preventing a 0 probability assignment, which could eliminate the correct particles.

### 2.2.1 Ray Casting

The sensor model computes the weight for a particle by computing the individual likelihood of the odometry laser range measurement ($z_t^k$) based on the true laser range measurement ($z_t^{k*}$). Odometry laser range measurement is based on the readings captured by the robot in the real world while the true laser range measurement is calculated by considering the position of the particle in the given map.

There exist several line (ray) tracing algorithms such as DDA (Digital Differential Analyser) algorithm to trace out a pixelated line from a known start point to a given end point. We have successfully implemented Bresenham's line algorithm [2] in our implementation in order to obtain the true range reading in a particular direction. In order to obtain ($z_t^{k*}$), we shift the individual particle positions to the position where the laser is mounted on the robot. This is considered to be the starting point of the laser beam. We then convert the positions into pixels for efficient implementation of the ray tracing algorithm. The end points are then calculated for a given angle ($\theta$) of the robot orientation, ranging from ($\theta$ - $\pi/2$) to ($\theta$ + $\pi/2$) by considering the maximum range of the laser. Bresenham's line algorithm allows us to generate the intermediate pixels coordinates that lie on the line joining the start and the endpoint. These intermediate points enable us to check the probability of occupancy of the map before reaching the maximum laser range. Based on the map file format provided, we consider occupancy probability greater than or equal to 0.35 to be occupied, and less than 0.35 chosen according to it's probability.

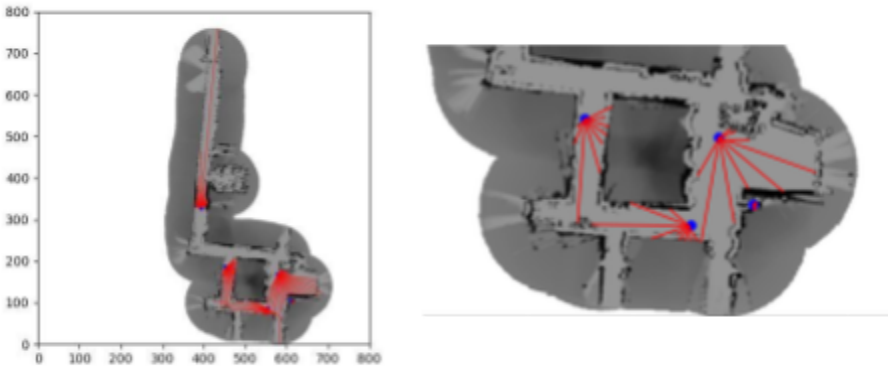


Figure 4: Visualization of ray-tracing

Considering that we process all the laser beams (i = 0 . . . 179), the ray casting algorithm implemented is shown in Figure 5.

***Set*** *laser_rayAngle* = $\theta$ - $\pi/2$ + i*$\pi/180$
***Set*** *start_point = laser position on the robot*
***Calculate*** *end_point = start_point + (max_laserRange * orientation)*
***Calculate*** *line tracing pixel using Bresenham's Line Algorithm*
***For*** *each pixel that lies on the ray:*
    ***If*** *occupancy probability > 0.35:*
        *Higher chance the pixel is occupied*
        ***Return*** *the distance upto the pixel*
    ***Else*** *:*
        *Higher chance the pixel is unoccupied*
        ***Return*** *the max_laserRange*
***Store*** *the true_laserRange* ($z_t^{k*}$)
***Visualise*** *laser_ray onto the map*

Figure 5: Pseudocode for ray_casting

The distance ($z_t^{k*}$) thus obtained is treated as the true range of the laser reading in that particular direction.

### 2.2.2 Parameter Tuning

The parameters for the sensor model were tuned over several iterations and observations. Over time, we understood the effects of different parameters to tune the system better. We visualized the net probability distribution,

$$p = z_{hit}\, p_{hit} + z_{short}\ p_{short} + z_{max}\, p_{max} + z_{rand}\, p_{rand}$$

by generating a range of fabricated sensor values, and assigned a value for the true range, $z_t^{k*}$. This gave us a good idea of the likelihoods of individual sensor beams. An example with $z_t^{k*}$ = 4000 is shown in Figure 6. For $p_{max}$, we assigned a high probability for any reading that would be below 80% of the max range which in our case is set to 8191 cm. This is because we found by testing our ray tracing, that the max true range we achieved was around 5000 cm, and therefore any laser reading considerably above the maximum true range obtained would be an error measurement.

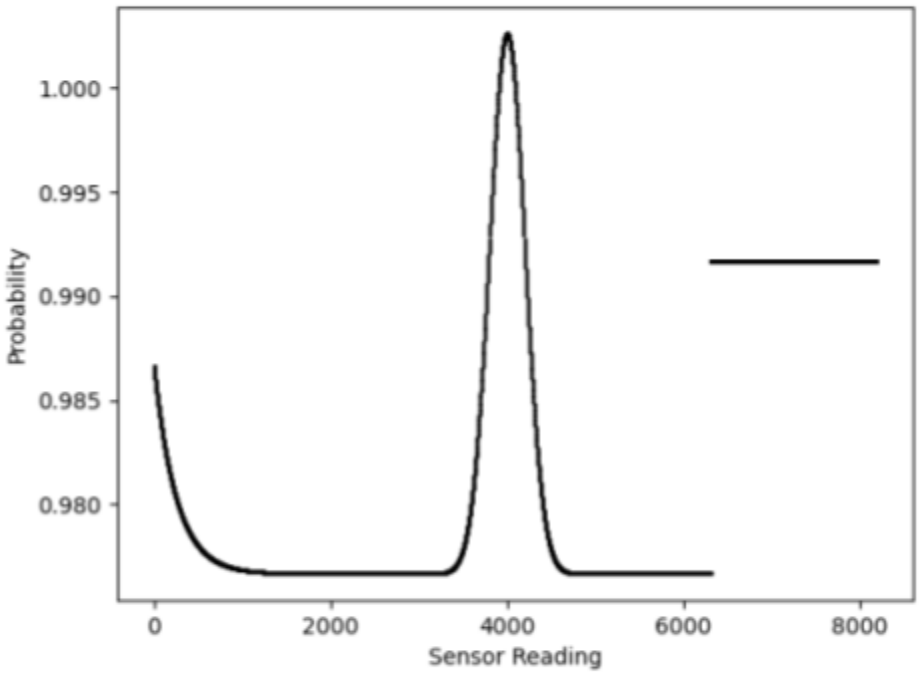


Figure 6: Density plot of our mixture distribution $p(z_t^k \mid x_t, m)$

Sensor model tunable parameters are elaborated below:

- $z_{hit}$ : Increasing the value would expect the sensor to be more accurate, and assign high likelihood values to measurements that agree with the true range. However, increasing this value too much would result in very fast convergence of the particles, and usually to the wrong location.
- $z_{short}$ : Increasing this parameter would assign higher likelihoods to values that are short of the true range. This parameter was tricky to play with since in the given .gif, there seems to be a person operating the robot who is in front of the robot, and therefore we expect several short readings. However increasing the value too much would result in wrong convergence, as particles that are facing towards obstacles, but in the wrong location would survive.
- $z_{max}$ : This parameter was set somewhat in between the uniform random probability distribution and the peak of the Gaussian distribution.
- $z_{rand}$ : This was adjusted to bring the likelihood values closer to 1, to avoid numerical issues. Setting the value such that the uniform distribution was too low of a likelihood compared to the gaussian, would cause the particles to converge very fast, as several particles would die out fast, due to the low weights assigned.
- $\sigma_{hit}$: Increasing this would give some slack to the sensor model, spreading the width of the Gaussian distribution, and therefore encapsulating several measurements around the true range value, therefore slowing down the convergence, but increasing the accuracy or robustness of the filter.
- $\lambda_{short}$: This scales the short distribution and is adjusted according to the $z_{short}$ to account for the unexpected object.
- *max_range*: We set this value to 8191 cm, which was the maximum range value observed in all the log files. This value is used in ray casting, and affects, $p_{hit}$ and $p_{rand}$. However, as explained above, any value over 6307 cm which is 77% of the max_range is considered in the likelihood distribution.
- *min_probability*: This is the minimum probability value for a pixel in the occupancy map to be considered occupied.
- *subsampling*: This value affects the number of rays to be sampled out of the 180 deg measurements. We set this value as 5 for most of our tests, as reducing this value increases the computation time.

Table 2: Parameters for sensor model

| No. | Symbol | Value |
|---|---|---|
| 1. | $z_{hit}$ | 12 |
| 2. | $z_{short}$ | 1 |
| 3. | $z_{max}$ | 0.01 |
| 4. | $z_{rand}$ | 8000 |
| 5. | $\sigma_{hit}$ | 200 |
| 6. | $\lambda_{short}$ | 0.002 |
| 7. | *max_range* | 8191 cm |
| 8. | *min_probability* | 0.35 |
| 9. | *subsampling* | 5 |
| 10. | *No. of particles* | 500 |

## 2.3 Resampling

We implemented both; multinomial sampling of the particles as well as low variance sampling. Multinomial sampling is a simple method used to sample particles from a multinomial distribution obtained from the importance weights of all particles.

For our implementation, we have used *low variance resampling* of the particles in order to update the particle set based on the weights calculated from the sensor model. We replace the assigned weights of the particles by the sensor model to the frequencies of the particles. This is essentially a survival of the fittest regime.

In *low variance resampling ,* we draw one random number and then draw the others equally spaced from the one sample, so instead of choosing $M$ random numbers and selecting those particles, low variance sampling generates a single random number $r$ and then selecting particles by repeatedly adding $M^{-1}$ to the random number. This basically enables the sampling of particles in a more systematic fashion. Moreover, we only sample the particles if the robot is considered to be moving in the world. This is a more efficient method than multinomial sampling since if all the particles have the same weights, this would replicate the result, while multinomial sampling could fail.

# 3. Performance

## 3.1 Lookup table

In order to enhance the efficiency of our code, we have a pre-computed lookup table for faster implementation of ray-casting. Basically, we pre-compute all the intermediate line tracing pixels for start point (0, 0) and for every angle ($\theta$) and then, we just shift those pre-computed points to the required laser location of a particle. This allows us to bypass one of the most computationally expensive operations i.e. ray-casting as we do not compute the ray casting intermediate points for every particle position and orientation.

## 3.2 Multiprocessing

To speed up the code, we implemented multiprocessing in python. Here we create a new function called monte_carlo, which is run with multiprocessing in the main code. As an experiment, we initialized 10 particles and ran it with and without multiprocessing. Without multiprocessing the code took 301 seconds to run and it took 208 seconds with. This increase in performance would be more significant for a higher number of particles. Implementation of multiprocessing enabled us to feasible test our code on a higher number of particles which eventually helped in faster parameter tuning iterations.

# 4. Results

The following are the YouTube video links for log1, log4 and log2 files.
Log1: https://youtu.be/kSVxaVqeip0
Log4: https://youtu.be/jUQijpoG68s
Log2: https://youtu.be/u97DY_wzUR8
We successfully generated these videos with parameters as listed in Table 2. However, we used a random seed for the random generation of particles, and running the code several times gave similar performances for that particular seed. The performances for log 4 and log 2 were more repeatable than those of log 1. There were several minor changes in parameters that affected the results, as the effects of the parameters are mentioned in Section 2.2.2. As the random seed was changed, the performance for log 1 required additional tuning, whereas the robustness for log 4 and log 2 were higher.

# 5. Future work and Improvements

i) Increase the speed of the code, by vectorized implementation in python.
ii) Increase the speed by utilizing the GPU, using libraries such as cuda, numba.
iii) Implement the algorithm that is able to automatically tune the sensor model parameters by studying the log file.
iv) Overall, the robustness of the system can definitely be increased by more appropriate tuning of parameters. We also ran into several issues with the motion of the particles, and therefore there is potential to further play with the motion model parameters to sync with a better performance.

# 6. Additional Tasks (Extra Credit)

### 6.1 Kidnapped Robot Problem

In order to solve the kidnapped robot problem, we first created a new log file with missing entries from its original version. We observed that around the kidnapped location, the localized particles tend to spread in all directions. This increases the chance of the robot to find the nearest landmark. This can be achieved by comparing the consecutive sensor readings that enables the robot to update the weights accordingly and localise to the new landmark.

https://youtu.be/U037GU0ST7c - Kidnapped Robot

### 6.2 Adaptive number of particles

The performance of particle filter algorithm is significantly affected by the number of particles. Adaptive number of particles over timesteps surely affects the computational time in a positive sense. But it gets a bit tricky while implementing this method. There are several points to be taken into consideration which include what criterion should be used to reduce the number of particles,

One thought is that during the resampling process, we keep the number of particles the same, although the frequency of certain particles will increase. Here based on the occurrence of each particle, we could proportionally reduce the number of particles by selecting from the resampled set, when a certain threshold number of particles of particles are the same. This would reduce the total number of particles, but retain information about the localized position.

# 7. References


1. Thrun, Sebastian 1967- author. *Probabilistic Robotics*. The MIT Press, 2005.
2. *The Bresenham Line-Drawing Algorithm*,

   www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html.