16-833 Homework 1 Robot Localization using Particle Filters

Kangni Liu, Sreeram Thirupathi, Jinyun Xu

1. Particle Filter Code

1.1. Motion Model

The motion model implementation was done following the instructions for the Odometry Motion Model with sampling found in Table 5.6 in Chapter 5.4.1 of [1].

Algorithm sample_motion_model_odometry(u_t, x_{t-1})**:**

$\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$

$\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$

$\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot1}} + \alpha_2 \delta^2_{\text{trans}})$

$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta^2_{\text{trans}} + \alpha_4 \, \delta^2_{\text{rot1}} + \alpha_4 \, \delta^2_{\text{rot2}})$

$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot2}} + \alpha_2 \delta^2_{\text{trans}})$

$x' = x + \hat{\delta}_{\text{trans}} \cos(\theta + \hat{\delta}_{\text{rot1}})$

$y' = y + \hat{\delta}_{\text{trans}} \sin(\theta + \hat{\delta}_{\text{rot1}})$

$\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

return $x_t = (x', y', \theta')^T$

Table 5.6 Algorithm for sampling from $p(x_t \mid u_t, x_{t-1})$ based on odometry information. Here the pose at time t is represented by $x_{t-1} = (x \;\; y \;\; \theta)^T$. The control is a differentiable set of two pose estimates obtained by the robot's odometer, $u_t = (\bar{x}_{t-1} \;\; \bar{x}_t)^T$, with $\bar{x}_{t-1} = (\bar{x} \;\; \bar{y} \;\; \bar{\theta})$ and $\bar{x}_t = (\bar{x}' \;\; \bar{y}' \;\; \bar{\theta}')$.

The changes in rotation and translation between the two odometry states u_{t-1} and u_t are determined and then adjusted with random samples before describing the distribution of position and pose states at the later timestamp, and this distribution is then applied to the particles. The δ values are all scalars as they are derived from single state odometries at two different times. However, in our code we took multiple random samples with the proper variance values in the

algorithm. We took one sample for every particle in x_{t-1} at each point because we decided this would provide more variation in the particles than a single sample which would be applied to all particles at the same timestamp, leading to too much uniformity to properly account for randomness.

1.2. Sensor Model

The implementation of the correction or sensor model basically followed the instruction in chapter 6.3 in [1]. There are three main sections in the measurement update: odometry transformation, ray casting, and probability density mixture.

1: **Algorithm beam_range_finder_model**(z_t, x_t, m)**:**

2: $q = 1$

3: *for* $k = 1$ *to* K *do*

4: *compute* z_t^{k*} *for the measurement* z_t^k *using ray casting*

5: $p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$

6: $+z_{\text{max}} \cdot p_{\text{max}}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

7: $q = q \cdot p$

8: *return* q

The first step is the transformation of the sensor measurements and the robot odometry. There are 25 degree rotations and 10 cm unit grids that the measurements must fit into.

The next step, ray casting, is further divided into three steps. The 180 degree sensor measurements were divided into 18 beams that covered an arc of 10 degrees each (`self._subsampling = 10, self._K = 18`). The sensor model algorithm then keeps recursively expanding all beams and checks all collisions until they all stop. There are three conditions that can be defined as a type of collision: measurement out of boundary, reaching an occupied area in the map, and measurement out of range. After the ray casting algorithm, we can get *zt_star*, the "true" range of the object measured by *z_t*.

The last step is the probability density mixture. Following the algorithm above, the calculations were performed for hit, short, max, and random ("rand") densities. Using these parameters, the mixed density looks like Figure 1 below, which is not exactly the same as the ideal model, but works for this case. Short and max are the smaller weight distributions while hit and random densities are bigger.

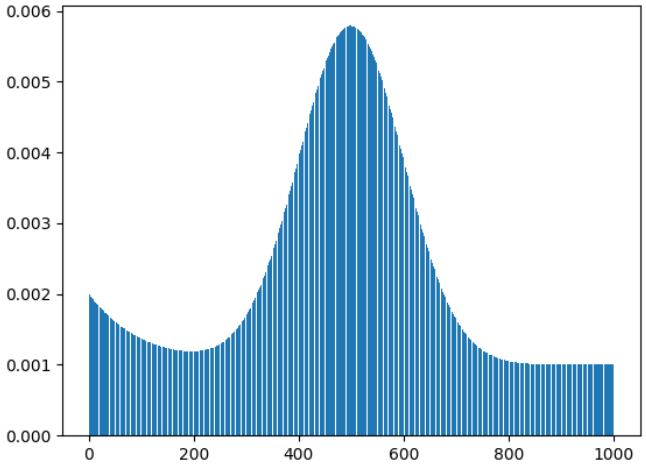

Figure 1: Mixed density

1.3. Resampling Process

The resampling process was performed with a low variance sampler algorithm, which was implemented following the instructions in chapter 4.2.4 in [1]. The algorithm can be divided into the following steps:

1. The weight calculated in the sensor model is accumulated. This creates a cumulative weight distribution, where the weight of each particle is the sum of its own weight and the weights of all the particles before it.
2. A random number r is generated between 1 and 1/M (We chose to use M = 5000).
3. Select the first particle in the sorted list whose cumulative weight is greater than or equal to r +(m-1)/M, where m is the index of the particle in the array of particles. That particle is added to the resampled set. This process is repeated for each particle until the array is filled with the same number of particles that it began with.

Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):
$\bar{\mathcal{X}}_t = \emptyset$
$r = \text{rand}(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
for $m = 1$ to M do
  $u = r + (m-1) \cdot M^{-1}$
  while $u > c$
    $i = i + 1$
    $c = c + w_t^{[i]}$
  endwhile
  add $x_t^{[i]}$ to $\bar{\mathcal{X}}_t$
endfor
return $\bar{\mathcal{X}}_t$

1.4. Parameter Tuning

1.4.1 Motion model parameters tuning

The results for different parameter configurations and other settings within the motion model are listed below.

Case number	Parameters	Running time	Motion performance description	Convergence frame
0	alpha1= alpha2=0.0001 alpha3=alpha4=0.01	225s	The robot moves out of the room in the center of map and sequentially move down, up, down along the hallways	29
1	alpha1= alpha2=0.00001 alpha3=alpha4=0.01	217s	The robot moves out of the room in the center of map and sequentially move down, up, down along the hallways	49
2	alpha1= alpha2=0.001 alpha3=alpha4=0.01	208s	The robot wanders in the room for a while and then move out of the room and sequentially move down, up along the hallways	33
3	alpha1= alpha2=0.01 alpha3=alpha4=0.01	233s	The robot converges at the edge of the room and keeps wandering, never enters the hallways	59
4	alpha1= alpha2=0.0001 alpha3=alpha4=0.001	222s	The robot moves out of the room in the center of the map and sequentially moves down, up, down along the hallways.	29
5	alpha1= alpha2=0.0001 alpha3=alpha4=0.1	227s	The robot moves out of the room in the center of the map and sequentially moves down, up, down along the hallways. But finally hits into the wall	29
6	alpha1=alpha2=0.0001; alpha3 = 0.013; alpha4 = 0.005	135s	Implemented Kidnapped Robot Problem solution (adding more particles) and adaptive particle resampling to speed up solution. The robot moves out of the room in the center of the map, then down and up the hallway before returning to the room. It does not hit the walls but the points scatter out when the robot is “kidnapped” and moves too much between odometry slices.	29

In summary, all 4 parameters have some influence on the running time of the algorithm, but not much. It was determined that increasing alpha1 or alpha2 negatively affects the robot motion performance, while decreasing them causes a longer convergence period, so they were kept at 10^{-4}. The parameter values in Case 0 (alpha1= alpha2=0.0001; alpha3=alpha4=0.01) were determined to be an acceptable parameter choice, considering the motion performance, run time, and convergence. It was also found later that the Case 6 parameter values (alpha1= alpha2=0.0001; alpha3=0.013; alpha4=0.005) gave a slightly better performance.

1.4.2 Sensor Model Parameters Tuning

We shall use alpha1= alpha2=0.0001,alpha3=0.013, alpha4=0.0005 as the motion model parameter and tune sensor model in this part. The default parameters are shown below. Since there are many parameters involved, we chose one parameter to optimize at one time and then choose another one while keeping the optimized value.

Case number	Parameters to be tuned	Running time	Motion performance description	Converge frame	Whether to update this parameters
0		1008s	The robot converge at the bottom of the map and then sequentially moves up, down, right, and down	116	
1	subsampling =5	407s	The robot move down and split at the crossroad, then move up and split at the cross road	295	No
2	subsampling=10	233s	Robot converges at the center hallways and then move up and then down	669	Yes
3	z_hit=0.9	234s	Robot converges into two parts at the center hallways and then move up and then down	808	No
4	z_hit=1.1	232s	Robot converges at the center hallways and then move up and then down	709	Yes
5	z_max=0.01	233s	Robot converges at the center hallways and then move up and then down	698	No
6	z_max=0.0003	231s	Robot converges at the center hallways and then move up and then down	600	Yes
7	z_short=1	232s	Robot converges at the center hallways and then move up and then down	690	No
8	z_short=0.01	227s	Robot wanders around crossroad and hit into wall	890	No
9	z_rand=10	191s	Robot hit the edge of map	618	No
10	z_rand=1	205s	Robot move sequentially right, left, up, down	84	Yes
11	sigma_hit=80	218s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down and hit the wall	66	No
12	sigma_hit=100	218s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down and hit the wall	29	Yes
13	lambda_short=0.1	221s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down and hit the wall	49	No
14	lambda_short=0.01	223s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down	29	Yes
15	max_range=100	85s	Robot not converges to a position	NaN	No
16	max_range=10000	405s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down	17	No
17	min_probability =0.1	218s	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down	71	No
18	min_probability =0.6	243	Robot converge in the room in the center of map, then move out of the room and move sequentially down, up, down	43	No

The final optimized sensor model parameters is shown below

Parameters	z_hit	z_short	z_max	z_rand
Default value	1.2	0.1	0.0005	1
sigma_hit	lambda_short	max_range	min_probability	subsampling
100	0.01	1000	0.35	10

2. Performance

2.1. Efficiency

After vectorization and free space initialization, the efficiency is significantly improved. With the video displayed and the pictures saved, the speed reached 22 FPS for log1 and log5. Without saving images, the speed will be even faster. (Also, this data was collected in Jinyun's personal laptop. Speed may vary with different computers.)

2.2. Converge Performance

Most of our tests took around 50 frames (out of 2000+) for particles to converge into one certain point. This efficiency is high enough, but the convergence stability needs to be improved. All tests have particles converging into one point, but there is a chance that they will converge to different locations with different random seeds.

2.3. Moving Direction

The robot's moving trajectory in robotdata1.log looks very similar to the movement in the given gif. After going outside the small room, the robot will turn left and move downward then turn 180 degrees. If the Robot Kidnapping Problem is not provided a solution in the program, the particle filter says the robot continues moving upward in the same hallway and then finally returns near the starting point. If the problem is solved as described in Section 3.1 on the next page, the particle filter instead says the robot enters the room it began in, goes to a corner, and then returns to the room's exit. If the first convergence point is assumed to be correct then the convergence is quick and the trajectory is well-followed, though the particle filter's behavior when the robot is "kidnapped" briefly indicates the second trajectory is more correct.

Additionally, with some less-effective parameter value sets, the robot observations will indicate the robot is going through a wall and the robot localization will be stuck in the wall for a while.

For other cases, although the robot may not converge to the correct starting point, the trajectory is similar to the desired path. For example, if the initial convergence is incorrect for robotdata1.log, the robot still moves in one direction, turns 180 degrees, then comes back

Video link to video of particle filter for robotdata1:
https://drive.google.com/file/d/1M4fZxn8Ynx4K6mnaXuy8hoRk2AA59fc4/view?usp=share_link
Video link to video of particle filter for robotdata5:
https://drive.google.com/file/d/1LvbeRs8dQwuzHeXrET54uonzf3eRK1mb/view?usp=share_link
Video link to video of particle filter for robotdata1 with the Kidnapped Robot Problem solution:
https://drive.google.com/file/d/1PCbC_wdKMek8QoBmeEFH95rarZfyi8Vb/view?usp=share_link
(This video was sped up to 200% speed to be within 2 minutes)
Video link to video of particle filter for robotdata1 with Adaptive Number of Particles:
https://drive.google.com/file/d/1w9EKx53d64FLElwVAdS936kbjnUYnwNo/view?usp=share_link

2.4. Repeatability

At the final settings listed in Section 1.4 Parameters Tuning, the particle filter (ran with both the Adaptive Number of Particles and the Kidnapped Robot Problem Solution) results were highly repeatable with robotdata1.log, as during 10 trials the simulation followed the robot from the center room, out into the hallway, down and up the hallway, and then back into the center room as the robot continued from the entrance to the edge and back to the entrance of the room. For robotdata5.log, the particle filter results were also highly repeatable as during 5 trials the simulation predicted the same behavior during every trial.

3. Extra Credit

3.1. Kidnapped robot problem

To solve the kidnapped robot problem, we decided the simplest solution would be to add more particles whenever a “kidnap” was detected. To do this, we added in a check right before the Motion model step of the particle finder algorithm. If the sum of square differences of the last odometry x- and y-positions and current odometry positions was over a threshold (we selected 70) then the robot is considered to be kidnapped, and a function similar to the particle initialization function gets called to initialize more particles and append them to the array of existing particles. The number of particles that are added depended on the current number of particles. If the adaptive particle resampling algorithm had reduced the number of particles significantly (by at least 20% of the original number of particles) then the difference in number of particles would be used as the number of particles to add back for the kidnapped robot problem. If the number of particles was still close to the originally initialized number of particles, then one-fifth of the original number of particles would be created for the kidnapped robot problem routine instead.

These new particles for the kidnapped robot problem condition were not drawn from the occupancy map like the initial array of particles, but instead were given random values within appropriate ranges for x, y, and theta. The new particles would have a random x-value drawn from a uniform distribution centered at the mean x-value of the existing particles, ranging from 200 below the center to 200 above the center. The new particles’ y-values were determined with a similar distribution centered around the mean of the y-values of the existing particles. The values of theta for the new particles, however, were randomly drawn from a distribution centered at the mean value of theta for the existing particles but ranging from $\frac{\pi}{4}$ below to $\frac{\pi}{4}$ above that means. The new particles all started with a balanced weight equal to one over the new total number of particles, previous and new combined. The called function would return only the new particles, and the main loop’s if condition would append the new particles to the array of all previous particles before continuing the loop with the motion model.

3.2. Adaptive number of particles

The number of particles was made adaptive in the resampling algorithm with a few changes. First, as usual, the low variance sampler would calculate the expected cumulative weight if everything was equally balanced, plus a random value between 0 and 1 over M, the number of particles. The weights would then be tracked with a cumulative sum that started with the first weight value in the list. If that cumulative weight sum was under the expected cumulative weight, the function would iterate across the list of weights until it came to a particle with enough weight that the cumulative sum of weights up to it met or exceeded the expected cumulative weight, and then remember that particle. Then, with the cumulative weight sum high enough, that current particle could be copied to the current row of the return array. This would continue for as many particles were provided to the function.

To change this, a maximum number of times a particle's data could be copied into the return array was initialized and set to 10. Another variable kept track of how many times a particle was added to the return array without changing the particle being copied. This way, as certain particles gained higher weight from accurately guessing the position of the robot over time, those particles would not be copied many times over and the array could be trimmed down to only a certain number of copies for every particle with the same data. The max value was chosen to be 10 so that it would still reduce the number of particles in the array over time, but even if there was only one estimate, there could still be enough particles for the motion model to apply a variety of random changes to accurately follow the robot.

3.3. Improving Efficiency

To increase the efficiency of computing and testing, the following suggested strategies were implemented: vectorized python and particle initialization in free space.

3.3.1. Vectorized Python

The code vectorization started at the very beginning of the particle filter algorithm. We changed the *main.py* file from using a for loop over every particle to instead using an array where every row was a particle. The variable *m* represents the number of particles. Every state and measurement variable for particles has m rows or is otherwise dependent on the value of m. This greatly improves the performance over a for loop. For both the motion model and the sensor model, the same shape structure was followed to process all calculations.

3.3.2. Particle Initialization

To further improve the efficiency, the function *init_particles_random* was replaced with *init_particles_freespace*, which chooses particles only from the free space from the data map. The basic idea remains the same, but now the initial set of particles only considers points where the occupancy boolean for that point is false.

4. Future Work

The algorithms converge properly as implemented for most cases, but the robustness can be further improved. Particles may converge to different locations with different random seeds, and the sensor parameters can be better tuned to ensure they converge to the same points during at least 95% of all trials. The time and memory efficiency of the code can be further improved with more data-efficient algorithms which use even fewer particles, or by using a better language for large data handling like C++.

Reference

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