16-833: Robot Localization and Mapping, Spring 2021

Homework 1 Robot Localization using Particle Filters

Tianxiang Lin (tianxian)
Kerou Zhang (kerouz)
Jiayin Xia (jiayinx)

Contents

1 Abstract

This assignment utilizes a particle filter to localize a lost robot given the map, its odometry and a laser rangefinder. The algorithm was completed by initializing particles, programming motion model, sensor model and resampling. Parameters were tuned to yield proper results. Vectorization and Parallelization were implemented to improve the performance of the algorithm. Robotdata1.log and robotdata2.log were used to test our algorithm and results are uploaded as videos. The implementation is successful in both data and can be repeated, proving the performance, repeatability and robustness of the algorithm.

2 Approach, Implementation and Parameter Tuning

Particle filter, also known as Monte-Carlo Localization, is implemented for this robot localization task. With the information of odometry, a laser rangefinder and a map of Wean Hall, this algorithm can be divided into four main parts: particle initialization, motion model, sensor model and resampling. Figure 1 demonstrates the Particle Filter main algorithm, which is implemented in main.py.

```
Algorithm 1 Particle Filter for Robot Localization
 1: $\bar{\mathcal{X}}_t = \mathcal{X}_t = \phi$
 2: for $m = 1$ to $M$ do
 3:     sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$          ▷ Motion model
 4:     $w_t^{[m]} = p(z_t \mid x_t^{[m]})$                              ▷ Sensor model
 5:     $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
 6: end for
 7: for $m = 1$ to $M$ do
 8:     draw $i$ with probability $\propto w_t^{[i]}$                   ▷ Resampling
 9:     add $x_t^{[i]}$ to $\mathcal{X}_t$
10: end for
11: return $\mathcal{X}_t$
```

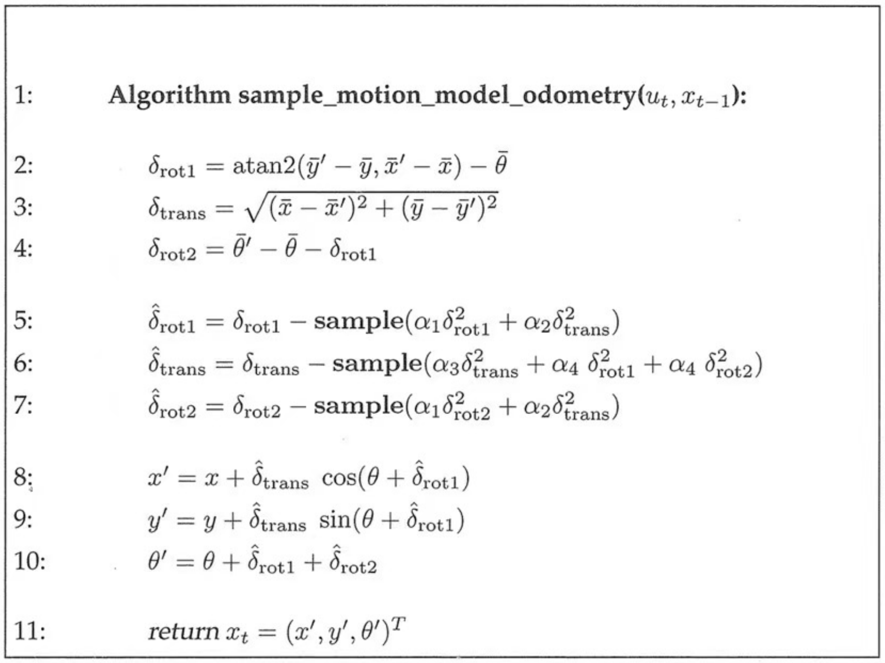
Figure 1: Particle Filter Algorithm [2]

2.1 Initialization of Particles

It can be assumed that the robot can only be inside the hallway or rooms. Hence, to increase computation efficiency and robustness, the method of init_particles_freespace was used to initialize particles uniformly inside the hallway and rooms instead of anywhere. It was completed by choosing coordinates that have an occupancy map value of 0. Weights were also uniformly distributed to every particle.

2.2 Motion Model

Motion model algorithm is adapted from table 5.6. The algorithm is based on odometry information. The pose at time t is represented by x, y coordinates, and theta of time t. The control is a differentiable set of two pose estimates obtained by the robot's odometer. Change in both initial and final rotation and change in translation were calculated in lines 2-4 in Figure 2. In line 5-7, the relative motion parameters for the given poses at time x_{t-1} and x_t were calculated. Drawn samples from the parameterized normal distribution were returned in "samples". Lines 8 to 10 computed the updates to the translation and rotation and returned as a 3 by 1 array.



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

2:        $\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
3:        $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
4:        $\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

5:        $\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot1}} + \alpha_2 \delta^2_{\text{trans}})$
6:        $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta^2_{\text{trans}} + \alpha_4 \ \delta^2_{\text{rot1}} + \alpha_4 \ \delta^2_{\text{rot2}})$
7:        $\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta^2_{\text{rot2}} + \alpha_2 \delta^2_{\text{trans}})$

8:        $x' = x + \hat{\delta}_{\text{trans}} \ \cos(\theta + \hat{\delta}_{\text{rot1}})$
9:        $y' = y + \hat{\delta}_{\text{trans}} \ \sin(\theta + \hat{\delta}_{\text{rot1}})$
10:       $\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

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

Figure 2: Motion Model Algorithm [2]

2.2.1 Motion Model Parameter Tuning

There were four parameters to be tuned in the calculation of drawn samples. These parameters determine the noise of the motion model. They were selected based on the effect they have on the convergence and particle movement compared to the true odometry movement after convergence. They were tuned after the sensor model yielded a proper result in probabilities.$\alpha 1$ and $\alpha 4$ affect translation motion, while $\alpha 2$ and $\alpha 3$ affect rotational motion. A relatively small need for a noisy motion model was determined. Giving too large values for $\alpha 1$ and $\alpha 4$ will cause particles to be along a line. Giving too large values for $\alpha 2$ and $\alpha 3$ will cause excessive rotational motion. Hence, after tuning around the initial values, all four parameters were kept at 0.001 as the best motion noise.

2.2.2 Vectorization of Motion Model

To improve the performance, the entire algorithm was vectorized to accept particle state odometry readings and beliefs at time t and t-1 with all particles. The inputs' shape was changed from 3 by 1 to three by the number of particles. So did the output motion updates. It improved the speed of the algorithm significantly as there is no need for thousands of iterations in a for loop in python.

2.2.3 Validity of Motion model Function

To test the functionality of the motion model, 5 random particles were generated, and their trajectories of movements were plotted using the odometry measurements, with 4 alpha weights all as 0.001. Figure 3 and 4 demonstrates that the trajectories of 5 particles (red) have similar shape to the robot trajectory (blue) obtained from odometry measurement log file.

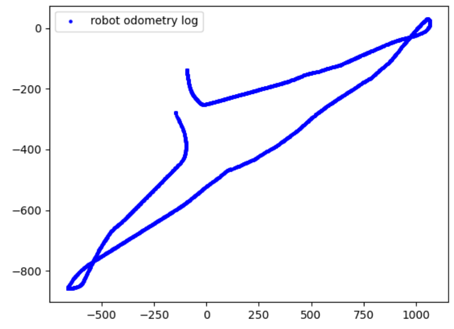


Figure 3: Robot trajectory obtained from odometry measurement log1 file

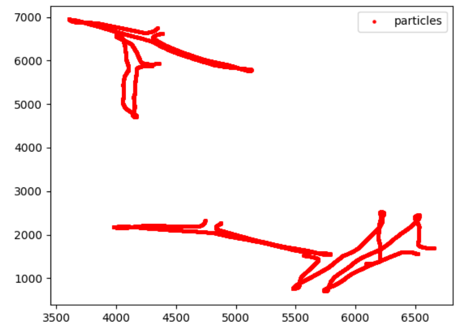


Figure 4: The trajectories of 5 particles

2.3 Sensor Model

Beam model of range finders in section 6.3 of Probabilistic Robotics is implemented as the sensor model. Figure 5 demonstrates the beam range finder algorithm. The input of the algorithm is a complete range scan, a robot pose of x, y coordinates and theta, and a map with x, y coordinates. The four probability distributions for hit, short, max and rand are shown in Figure 6. The first Gaussian distribution models correct range with local measurement noise. The exponential distribution accounts for unexpected objects near the sensor, such as human beings. The max value at the maximum range is for failures that obstacles are missed altogether, and the laser doesn't receive the signal back. Lastly, random distribution remedies entirely unexplained measurements and keep the probability from zero. The four probabilities are multiplied by their corresponding parameters and added together. In line 7, exponential of log sum is used instead of multiplication of previous probabilities.

```
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:               $+ z_{\max} \cdot p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$
7:           $q = q \cdot p$
8:       return $q$
```

Figure 5: Beam Range Finder Model [2]

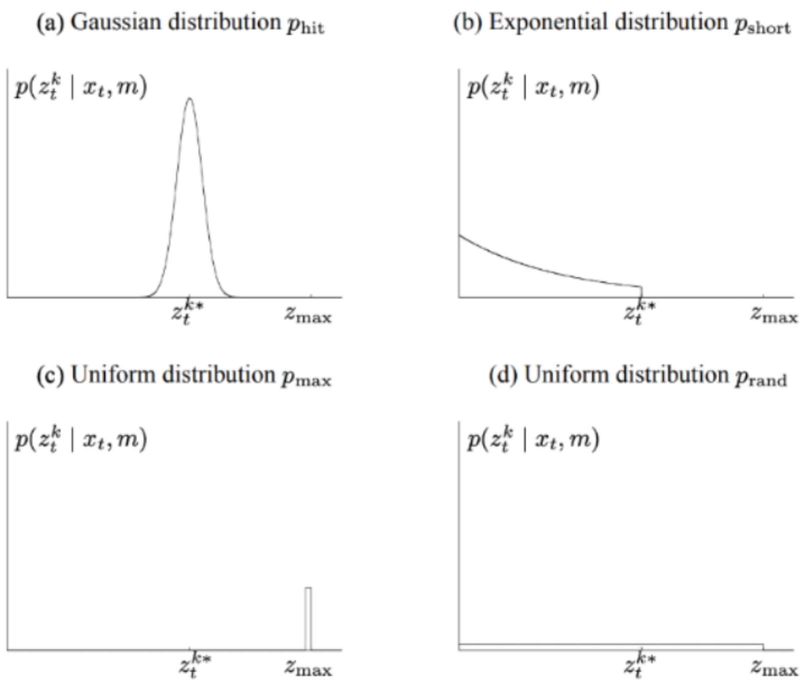


Figure 6: Components of the Range Finder Probabilities [2]

2.3.1 Ray Casting Algorithm

In line 4 of the beam range finder model, ray casting is computed to find the noise free range for a particular sensor measurement. It receives map coordinates and laser pose as input. There are mainly three steps in the algorithm : (1) Cast a ray from the laser position. (2) The coordinates of the ray reaching its first obstacle are recorded as the end point. (3) The distance between the laser position and the end point is calculated. The detailed algorithm is presented in Figure 7. To detect the first obstacle of ray, a step called stride was tuned as part of parameter tuning for the sensor model.

```
Algorithm 1: Ray Casting (map, Start point, theta)
  Result: Ray Casting Distance
  end point = start point;
  index = round(end point / map resolution);
  radians = (index * subsampling - 90) * $\pi/180$
  direction = theta + radians
  while obstacle not detected and map index not exceeded do
      end point += stride * direction;
      end point = round(end point / map resolution);
  end
  calculate distance between end point and start point;
  return distance;
```

Figure 7: Ray Casting Algorithm [2]

2.3.2 Validity of Ray casting Function

To test the validity of our ray casting function, the end points of casted rays for 10 particles were plotted after the first time of ray casting. Figure 8 demonstrates the correct functionality of the algorithm by showing that the rays all ended at positions on walls. We also initialized a particle state almost the same as the ground truth of the robot, Figure 9 demonstrates the 180 distances of ray casting of the particle compared to the odometry measurement data obtained from the log1 file after the first time of ray casting.

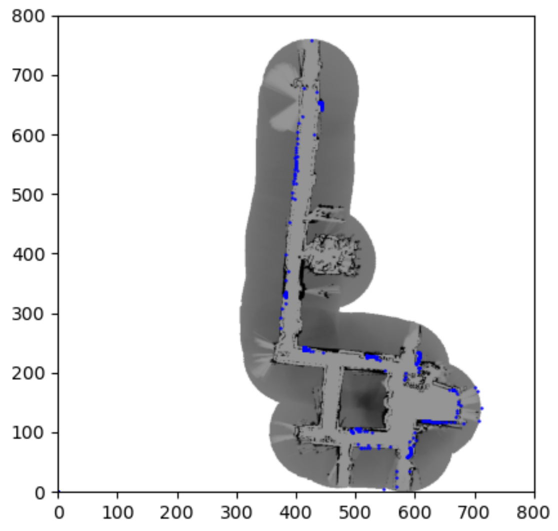


Figure 8: Ray Casting Ending Positions

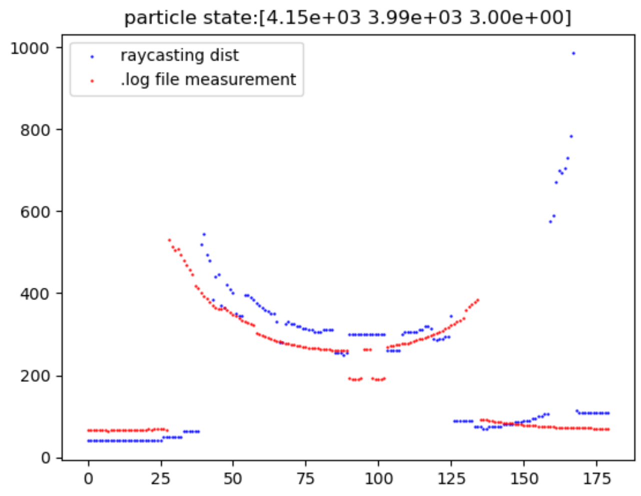


Figure 9: 180 distances of raycasting comparing to odometry measurement

2.3.3 Parameter Tuning

Parameter tuning for sensor models is separated into two areas: parameters for probability distributions and parameters for ray casting.

Parameters for probability distributions include four weights, standard deviation for Gaussian, and exponential factor for short distribution. The standard deviation and scale of Gaussian significantly account for the convergence rate and performance of the sensor model. Setting the standard deviation values too small (less than 50) will cause the convergence too fast and probably at a wrong location, while too large (more than 150) will let the probabilities be relatively minimal (if Gaussian scale unchanged) so that the particles won't converge. The scale of Gaussian, z_hit, influences the convergence rate as well for it is the vital element that mostly decides the value of overall probability of one particle. Too large the scale can cause the converging process too quick which may kill promising particles, while too low may prevent convergence and fail on localizing the robot. The exponential distribution describes situations where rays detect unknown objects, to adjust the distribution to a reasonable shape, we finalize the lamda_short as 0.1. The random measurements when rangefinders generate completely unexplained results, are described by a uniform distribution with weights z_max. This ensures the minimal probability value is about 1 which helps control numerical stability. Failures of measurement are described by an indicator function and weights of it z_max is set to a little bit larger than 1 but lower than the scale of the Gaussian.

Figure 10 demonstrates the sensor model mixture probability distribution $p(z|z*)$ using weights with $z*$ as 500 and max range with 1000.

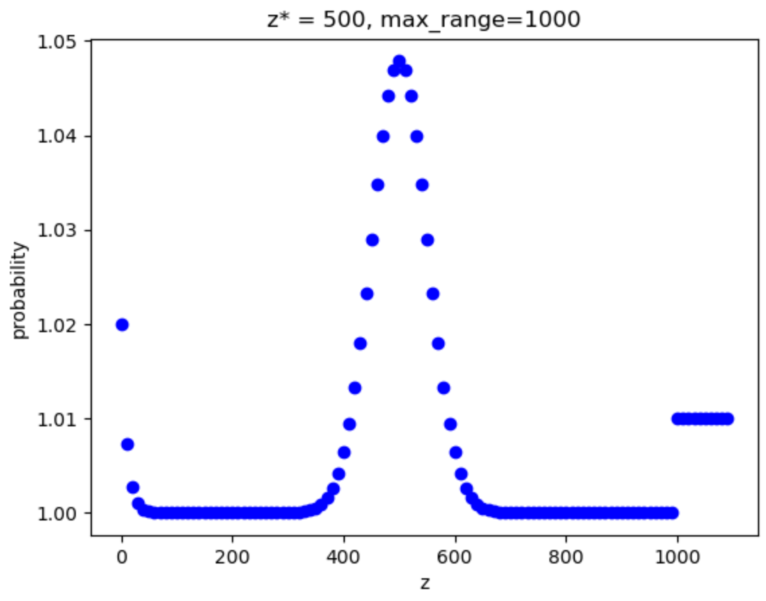


Figure 10: Sensor model mixture probability distribution

The other set of parameters include max range, minimum probability which accounts for ray casting's obstacle detecting, subsampling which defines the step of degrees from 0 to 180, and stride which defines the step of casting a ray.

Particularly, subsampling affects the value of probability, as well as adequation of the information. Too large may cause inadequate information obtained from raycasting and lead to not accurate raycasting results. But a too small subsampling like 1 may cause the probability gap between promising samples and bad samples too big, therefore lead to too speedy convergence and lose robustness.

A testing particle initialization was programmed for the tuning of parameters. From the GIF given, the true position of the robot is speculated and four particles accounting for true positions are initialized. On the other hand, four other particles are generated randomly in the hallway and rooms. Probabilities of the first four particles and second four are calculated correspondingly by the sensor model. Since the probability of the first four particles are expected to be large enough so that the algorithm converges to these positions, parameters are adjusted to make the difference of probabilities between first and second four particles as large as possible.

Meanwhile, subsampling and stride also determines the speed of the program. To make the sensor model accurate enough while efficient, the values need to be chosen wisely. The efficiency of the algorithm is largely improved with the help of vectorization and CUDA parallel programming, which will be elaborated later.

After tweaking parameters for the testing initialization algorithm, weights for hit, short, max, and rand are set to be 6, 0.2, 1.01, and 1000 correspondingly. Sigma_hit is 100 and lambda_short is 0.1. Max range, minimum probability, subsampling, and stride are set to be 1000, 0.35, 1 and 5.

2.3.4 Parallelization

To improve the efficiency of the system, a GPU-accelerated module was introduced by using numba, which contains a CUDA module. Sensor Model is parallelly implemented because the result of one point's weight will not influence others for sampled points. Hence, using CPU for sequential computation is a waste of computing power. Instead, parallel programming by CUDA is efficient in this scenario. From the aspect of GPU architecture, NVIDIA's GPUs consist of several streaming multiprocessors (SM), which can execute repeated instructions with different data with multiple threads in each SM. Different from CPU parallel programming, CUDA's thread hierarchy is organized with grids and blocks (Figure 11). Grids are composed of several blocks, while blocks consist of threads. An unique block index was used to denote a particular sampled point while thread indexes were used to denote the sampled angles in ray casting. In each thread, real distances by ray casting algorithm in Figure 6 were calculated and returned as a vector with size number of particles x (180 / subsampling).

2.3.5 Vectorization

For the same reason in the motion model, the sensor model algorithm is vectorized to accept input in two dimensional numpy arrays with all numbers of particles instead of a for loop. Particularly, the sensor model needs to be vectorized again since all probabilities are calculated from 0 to 180 degrees of the range sensor with a step of subsampling. Calculations of four probability distributions are all converted to 3 dimensional numpy arrays with size of 3 x number of particles x step angle degrees.

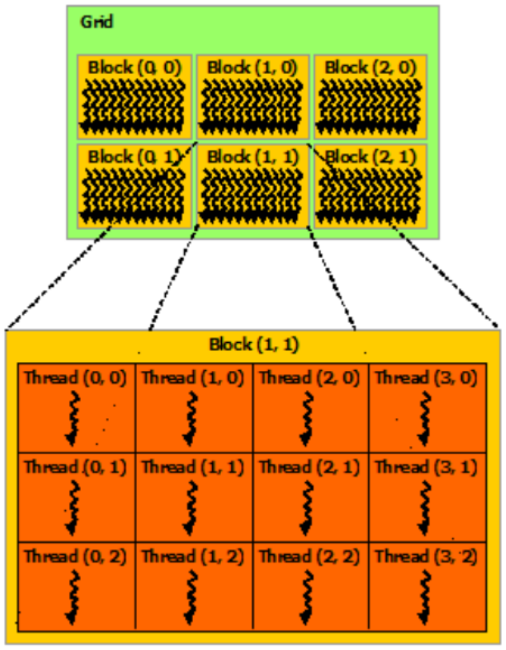


Figure 11: Thread Hierarchy of CUDA [1]

2.4 Resampling

Low variance sampler is implemented as the resampling method as shown in Figure 9. It uses a single random number to sample from the particle set with associated weights, yet the probability of a particle to be resampled is still proportional to its weight. It covers the space of samples in a more systematic fashion. It prevents particles with high weights to be discarded during the resampling while maintaining a low computation time.

2.4.1 Comparison of Low Variance model and Multinomial model

After normalization of all weights of M samples, a list of samples that can be considered samples arranged in a line and the length of each sample is proportional to its weights is created before resampling. The basic approach of resampling is randomly choosing M samples from the list. It is likely that the particles with higher weights are picked but it is not ensured every time. Therefore, the variance of such an algorithm is high and not ideal.

The low Variance model uses a single randomly generated number r, and picks the first sample from location r, and the rest samples from the location r+(m-1)/M. By moving along the list with the small increment of 1/M, it is ensured that samples with higher weights generate more samples.

3 Results, Performance, Robustness and Repeatability

The results of the particle filter are published in the Youtube links in the Appendix. 2000 particles are used for our implementation. For datalog1, our algorithm's particles converge to the positions close enough to the true robot position within several time indices. The convergence happens roughly around the coordinates of (400,400) and the particles move downward along the hallway. After the particles are about to reach the corner at roughly (400,300), they then rotate and move upward along the hallway to roughly (450, 650), rotate and move downward again. Finally, they ended just inside the room. The convergence rate is decent and motion traces are like the given GIF of robot motion. With the help of vectorization and CUDA, the speed of the algorithm improves dramatically. Within the limitation of GPU memory and number of streaming multiprocessors, the computation time for the algorithm is independent upon number of particles and subsampling rate. This means depending on the requirement of output accuracy, different numbers of particles and subsampling rate can be selected with no need to concern about computation time. Table 1 shows the implementation time with respect to number of particles and processing methods.

Table 1: Log 1 Data Implementation Time

Number of Particles	**Processing Method**	**Execution Time**
1000	Sequential	4 hours
2000	One-step Parallel	2 hours
1000	One-step Parallel	40 minutes
2000	Fully parallel	7 minutes
1000	Fully parallel	7 minutes

Repeatability is around 50% since almost half of the test rounds of fully parallel processing with 2000 particles ended up with similar convergence locations and similar particle motions. Convergence is not promised for every test round since initialization of particles is random every time. The test rounds which failed to converge are due to the incorrect position or orientation of initial particles. Robustness is proved since the same algorithm and parameters produce acceptable results when the data is switched to robotdata2.log. Using 2000 particles with fully parallel processing, the implementation time of robotdata2.log is also around 10 minutes.

4 Future Works and Improvements

The improvements can be classified to two main categories: parameters and implementation speed. The parameters tuned are within acceptable ranges and can yield decent results, but they are not promised to be the optimal solution. The Learn Intrinsic Parameters algorithm in table 6.2 of Probabilistic Robotics can be implemented for better parameter tuning for the sensor model. Moreover, with the help of CUDA and parallel processing, the implementation time improves significantly. However, python is relatively slow compared to other computing languages. Future works can be utilizing C++ for significant improvement in computation speed and efficiency.

5 Appendix

5.1 Video for datalog1-5

`https://youtu.be/rQKZF9E91iI`

References

[1] *CUDA C++ Programming Guide*. `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`.

[2] Sebastian Thrun. “Probabilistic robotics”. In: *Communications of the ACM* 45.3 (2002), pp. 52–57.