By: Huangji Wang(519030910100) and Zhiteng Li(519030910105)

HW3: Particle Filter

December 13, 2021

Contents

I. INTRODUCTION

A. Purpose

The goal of this assignment is to implement particle filters for state estimation. We will use an autonomous robot DR20 and try to estimates its location given inputs from sensors and a model of the robot motion. In general, a particle filter is a statistical model used to track the evolution of a variable with possibly non-gaussian distribution. The particle filter maintains many copies (particles) of the variable, where each particle has a weight indicating its quality. The particle filter is continuously iterated to improve the localization estimate and update localization after the robot moves. This happens in three steps: Prediction, Update, and Resample. Our final goal is to realize the **Local Position Tracking problem, Global Localization problem** and **Kidnapped Robot Problem**.

At last, we learned MCL and AMCL algorithms and try to realize them.

B. Equipment

There is a minimal amount of equipment to be used in this lab. A few requirements are listed below:

- Visual Studio Code: Version 1.60.
- Windows environment: Python 3.6, Numpy 1.19.5
- Required environment: Anaconda 3.6.
- Note: The scene model file is not supported on the macOS version of CoppeliaSim.

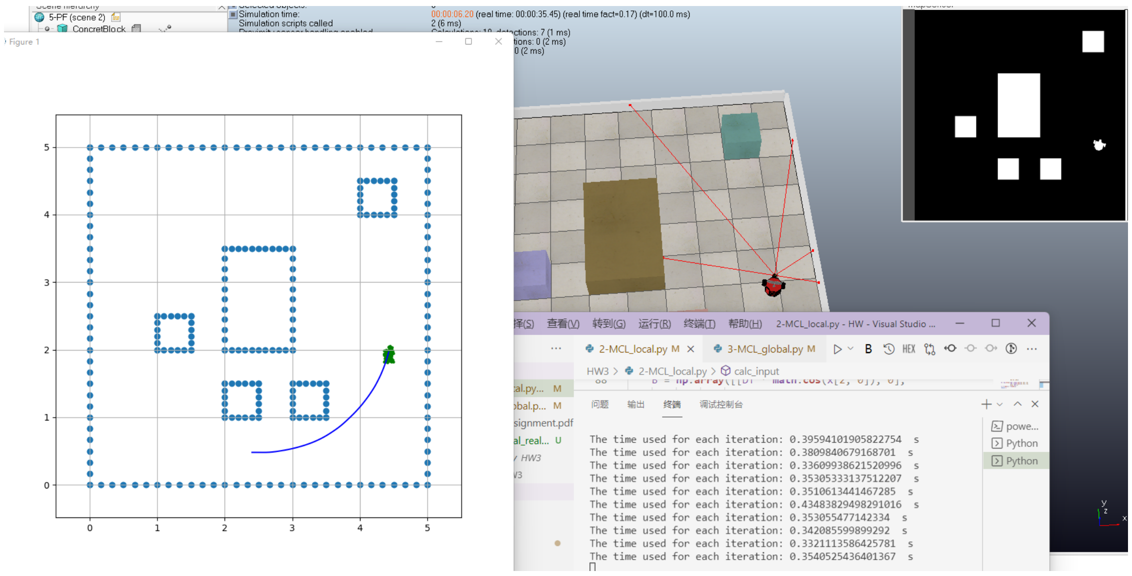
C. Procedure

1. Install the essential environment.
2. Start VS code. Open all the main codes.
3. Run the codes in the terminal. Observe the trajectory of the agent.
4. Plot the walk of the particles and analyze the situation.
5. Stop the simulation, examine the graphs, and make a summary.

II. PATH DIAGRAMS

This section consists of screenshots taken during the procedure in three tasks. All the videos are presented in the zip pack. The screenshots give you the time cost of each iteration, the state of the agent and the path and particles around the agent. You can just read these screenshots to gain information.

A. Local MCL Algorithm

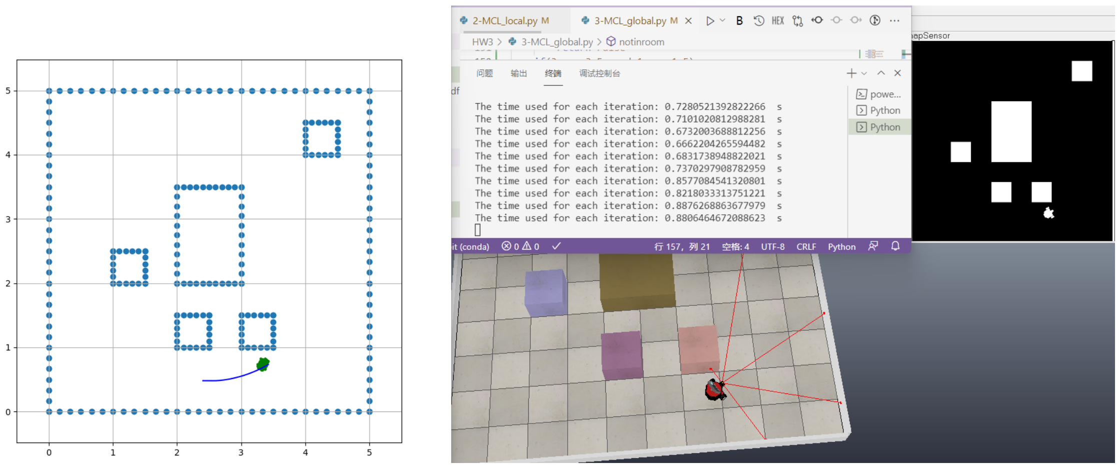


2-MCL_local.py - HW - Visual Studio ...
2-MCL_local.py M
3-MCL_global.py M
HW3 > 2-MCL_local.py > calc_input
问题
输出
终端
调试控制台

```
The time used for each iteration: 0.39594101905822754  s
The time used for each iteration: 0.3809840679168701  s
The time used for each iteration: 0.33609938621520996  s
The time used for each iteration: 0.35305333137512207  s
The time used for each iteration: 0.351061344146728 5  s
The time used for each iteration: 0.43483829498291016  s
The time used for each iteration: 0.353055477142334  s
The time used for each iteration: 0.342085599899292  s
The time used for each iteration: 0.3321113586425781  s
The time used for each iteration: 0.3540525436401367  s
```

FIG. 1: Local MCL Algorithm result.

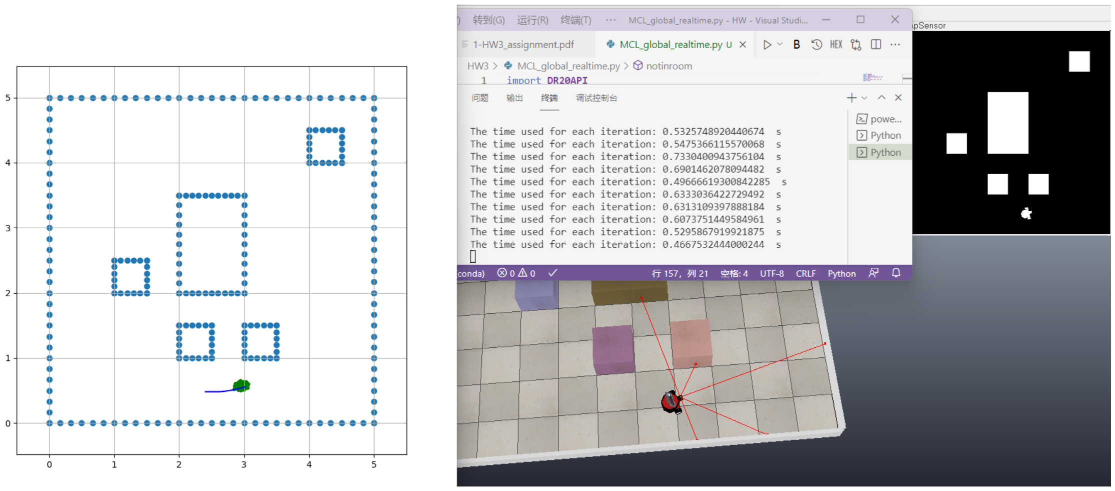
B. Global MCL Algorithm



```
The time used for each iteration: 0.7280521392822266  s
The time used for each iteration: 0.7101020812988281  s
The time used for each iteration: 0.6732003688812256  s
The time used for each iteration: 0.6662204265594482  s
The time used for each iteration: 0.6831738948822021  s
The time used for each iteration: 0.7370297908782959  s
The time used for each iteration: 0.8577084541320801  s
The time used for each iteration: 0.8218033313751221  s
The time used for each iteration: 0.8876268863677979  s
The time used for each iteration: 0.880646467208862 3  s
```

FIG. 2: Global MCL Algorithm result.

C. Global Realtime MCL Algorithm



```
import DR20API
```

```
The time used for each iteration: 0.5325748920440674  s
The time used for each iteration: 0.5475366115570068  s
The time used for each iteration: 0.7330400943756104  s
The time used for each iteration: 0.6901462078094482  s
The time used for each iteration: 0.49666619300842285  s
The time used for each iteration: 0.633303642272949 2  s
The time used for each iteration: 0.631310939788818 4  s
The time used for each iteration: 0.6073751449584961  s
The time used for each iteration: 0.5295867919921875  s
The time used for each iteration: 0.4667532444000244  s
```

FIG. 3: Global Realtime MCL Algorithm result.

III. MONTE CARLO LOCALIZATION(MCL) ALGORITHM

This section will consist of the important code and the main idea of all the functions which displays our comprehension of these algorithms to meet the requirements of the lab.

A. Structure of MCL Algorithm

Based on the idea of MCL algorithm, we can easily know that the structure of this algorithm goes like:

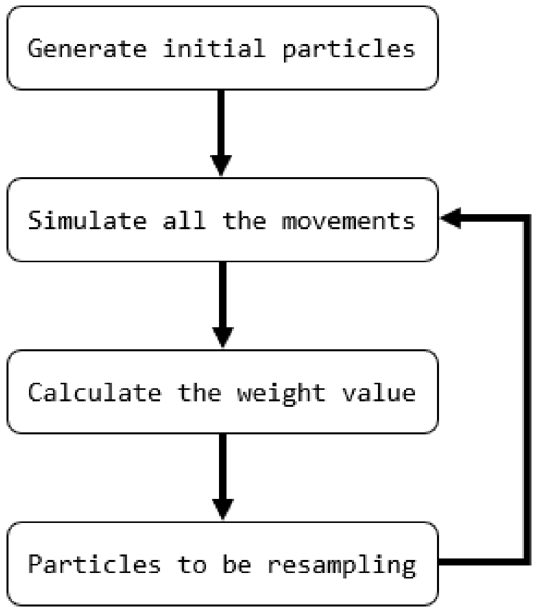

Generate initial particles
Simulate all the movements
Calculate the weight value
Particles to be resampling

FIG. 4: Structure of MCL Algorithm

Actually, the step of generating initial particles is finished at first. We should just finish the circulation then we can finish this lab.

B. Simulate All the Movements

This step aims at predicting the state of each particle with random input sampling. Therefore, we can just use the function **motion_model** which is used to return the state at next time when being given the current state and control input and then we can get the next state of each particle. In this step, we add the error of linear velocity v and angular velocity w, which we think can make the particle move in random ways.

```
tmp_px =np.array([[px[0][ip]],[px[1][ip]],[px[2][ip]]])
tmp_px =motion_model(tmp_px,u +np.array([u_error[ip]]).T)
px[0][ip], px[1][ip], px[2][ip] =tmp_px[0], tmp_px[1], tmp_px[2]
```

Based on the method, we can have the **prediction** of each particle in the next step.

C. Calculate the Weight Value

We want to assign each particle with a new weight, which based on the similarity between the particle and the real position of the robot. Since we have obtained the lidar reading with form

$$data = [0.4813, 0.6849, 2.6704, 0.9539, 0.5187]$$

where each lidar reading is represented by the distance value from the nearest obstacle in counterclockwise order. Based on this conception, we want to calculate the distance value from the nearest obstacle for each particle in a similar way.

1. Calculate Distance Between Particle and Obstacles

The way we measure the distance to the nearest obstacle is just try to move the particle in a small step, which is 0.1 for each step in the corresponding angle, and judge weather it meets an obstacle. By this way, we can get a corresponding $data'$ for each particle. However, it doesn't comes from the LIDAR, but comes from our simulation. The procedure is as below.

- Calculate the compound angle by $\theta \leftarrow \theta + \alpha$ for $\alpha \in (-\frac{\pi}{2}, -\frac{\pi}{4}, 0, \frac{\pi}{4}, \frac{\pi}{2})$.
- Move one step with step size 0.1 in the direction of θ.
- Judge weather we torch an obstacle.
- Stop when we torch an obstacle and return the distance we move.

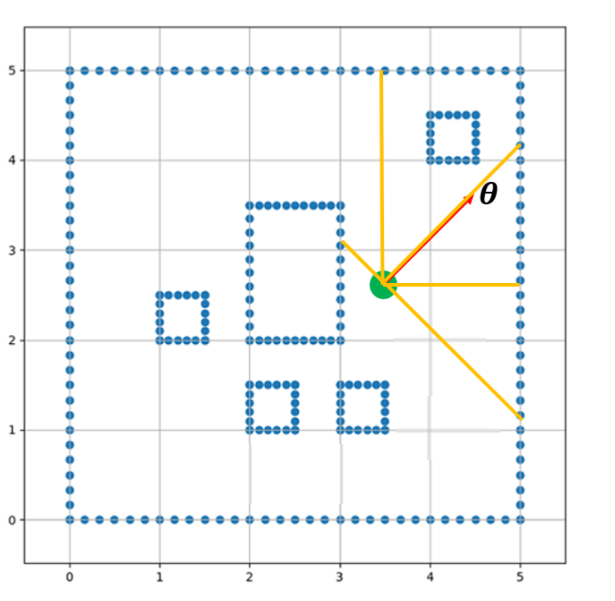


FIG. 5: Distance in 5 directions.

2. Weight Definition of Each Particle

After we obtain the distance between particle and obstacles, we sum the absolute value of the difference for each entry. If the sum is small, then our particle is more similar to the real position. Thus, we use the reciprocal of the sum to be the weight, which is

$$weight = \frac{1}{\sum abs(data' - data)}$$

It suggests that this weighted algorithm performs well in our experiment. The last step is to normalize all the weight to make them sum to 1, but the normalization is done for us and we don't need to do it again.

D. Particles to Be Re-sampling

This step aims at generating a set of new particles, with most of them generated around the previous particles with more weight. We follow the method provided in the class.

1. Generate New Particles

- Calculate the weight into the possibility line.
- Give a random value $v \in [0, 1]$.
- Find the correct interval and corresponding particle.
- Generate the correct particle.

To explain our method, we use one figure to present it. For each particle generation, we find the interval that suits the value v and generate the correct particle. Therefore, we generate NP particles for each re-sampling.

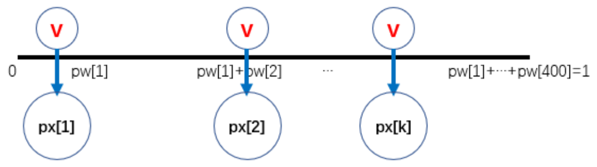


FIG. 6: The method to generate particles.

2. Set New Weight for Each Particle

The weight is already reflected in the number generated by the method above. Therefore, we want the weight to be equivalent, and thus we take average of all the weight. The strength of this method can ensure that we can just pay attention to one part of the parameters. No matter px or pw, we just need to consider one part of them.

```python
pw =np.zeros((1, NP)) +1.0 /NP # Particle weight
```

IV. RESULT VISUALIZATION AND ANALYSIS

A. Real Robot Path and Estimated Path

The real robot path has been plotted, we just need to plot the estimated path. After resampling, we assign each particle with the same weight, and the number of particles will actually represent the weight, so we can just sum them up and get the average position as our estimate path.

```
# Initialize the prediction history.
h_p_true =np.array([[pos[0]],[pos[1]]])

# For each iteration:
h_p_true =np.hstack((h_p_true, np.array([[sum(px[0,:])/400.0],[sum(px[1,:])/400.0]])))
plt.plot(np.array(h_p_true[0, :]).flatten(), np.array(h_p_true[1, :]).flatten(), "orange")
```

B. LiDAR Points on Map

To plot the LiDAR points, we use almost the same strategy as calculating the similarity, but it may be easier. To get the LiDAR point, We just calculate the new angle, and plus the distance in x axis and y axis.

```
angle =[- 0.5 *math.pi, -0.25 *math.pi, 0, 0.25 *math.pi, 0.5 *math.pi]
xx, yy =[], []
for i in range(5):
    x, y =pos[0], pos[1]
    x +=data[i] *cos(ori +angle[i])
    y +=data[i] *sin(ori +angle[i])
    xx.append(x)
    yy.append(y)
plt.scatter(xx,yy,color ='r')
```

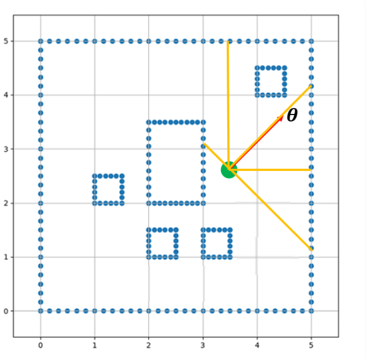


FIG. 7: LIDAR distance calculation.

C. Particles and Weight

Since after resampling, each particle has the same weight, but the same position may have repeat points, the more repeat points, the higher weight it will get, so we assign each particle with a different weight according the number of same points. But it may not be obvious in the video since all the particles are near each other.

V. DISCUSSION

A. Error to Be Inputted

To realize the goal of **scattering points at random**, we add error in our status modification. For example, we add error in the input control commands $u = (v, w)$, data observe and the particle moving modification px. Actually, it leads to a **large enough** circle which is similar as the random throwing. If we don't add the error, we can only get one single cluster point (the cluster point has 400 points in the place) and it violates the rule of **random throwing particles**.

```
u_error =np.random.randn(NP,2)@R
data_error =np.random.randn(NP*5,1)@Q
# Give the next step state
tmp_px =motion_model(tmp_px,u +np.array([u_error[ip]]).T)
# Update steps: Calculate importance weight
dis =[abs(obstacle(tmp_px, angle[i])-data[i] -data_error[ip*5+i])**2 for i in range(5)]
# Generate particles:
tmp_px[0][i], tmp_px[1][i], tmp_px[2][i] =px[0][j] +Q[0] *(-1 +2 *np.random.random()), px[1][j] +Q[0] *(-1 +2
                                                            *np.random.random()), px[2][j]
```

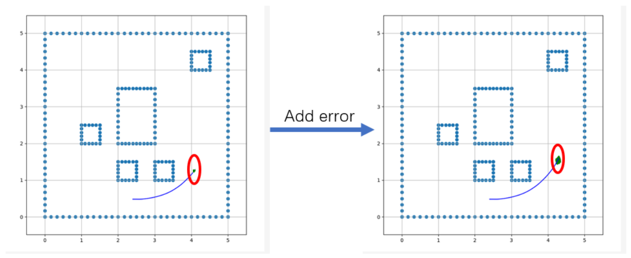


FIG. 8: The result when adding error.

From the position of LiDAR points, we can find that there exist some error, which suggests that the LiDAR is not so accurate. Thus, it is necessary for us to add some Gaussian Noise to the model.

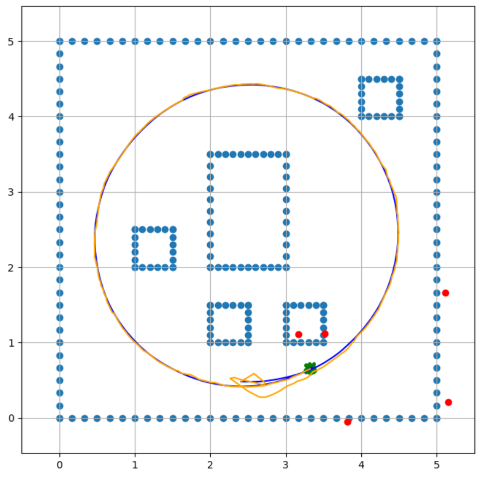


FIG. 9: The result of LiDAR points.

B. Reduce Time Cost in Each Iteration Step

Actually, the calculation algorithm of distance from the particle to the boundary or obstacles we design is an inefficient algorithm.

Our algorithm to find the distance goes like:

```
def obstacle(px, angle, step =0.05):
    # return the shortest distance when meeting obstacles
    x, y, theta, dis =px[0][0], px[1][0], px[2][0], 0
    # heng = x+d*cos(theta)  -90 -45    +0  +45    +90
    # zong = y+d*sin(theta)  -90 -45    +0  +45    +90
    while(notinroom(x,y)):
        x +=step *cos(theta +angle)
        y +=step *sin(theta +angle)
    return ((px[0][0]-x)**2 +(px[1][0]-y)**2)**0.5
```

We go the smallest step along the direction and find the distance. The figure is presented below:

However, we can find that this algorithm efficiency mainly depends on the places and distances. For example, the distances are different in the 5 directions in the graph. Thus, the step we choose will have a great influence on the efficiency. We firstly initialize *step* = 0.05 and leads to nearly 1 second in each iteration step. Next, we initialize *step* = 0.1 and leads to half cost in each iteration step. It leads to nice results.

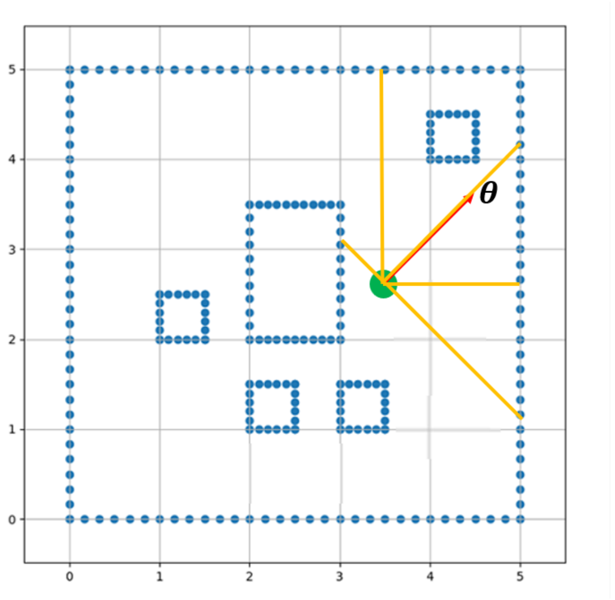


FIG. 10: Particle distance calculation.

```
# When step = 0.05
The time used for each iteration: 0.9836703300476074 s
The time used for each iteration: 1.2397853374481201 s
The time used for each iteration: 1.1779476642608643 s
# When step = 0.1
The time used for each iteration: 0.4428312778472900 s
The time used for each iteration: 0.5445635318756104 s
The time used for each iteration: 0.4637753963470459 s
```

C. Strengths

1. **Effective Algorithm to Calculate the Similarity**

   We design an algorithm to calculate the similarity between the particle and the real position, which simulates the work of LIDAR, try to move a small step in the specific direction. And finally we will get the distance infomation which is accurate and powerful to represent the weight of each particle. And since the obstacle will not be far away, we can always get the distance fast.

2. **Introduction of Gaussian Noise to Make it More Realistic and Robust**

   Since the real world is not so accurate as we assume, we add some Gaussian Noise based on the sigma Q and sigma matrix R, put it into the velocity of each particle. Thus, we add some random property to them and the model can adjust to different situation.

3. **Intuitive Visual Representation**

   We use a large amount of graphs in our experiment and in the final report, which helps you to realize it well and intuitively. Also, it records a lot of information of the robot and the particles so that it can help us to improve the whole model.

4. **Strong Scalability to High Requirement**

   Since the effective of our algorithm, we don't need to change a lot to realize the global version. Also, thanks to the efficiency, we can control our running time within 1 seconds in each step.

D. Weaknesses

1. **Running Time Depends on Step Size**

   We fix the step size to 0.1 since it is just enough to judge the obstacle for this specific map, and the accuracy is enough since we also add some Gaussian Noise to the update step. However, if we run it in a big map with higher accuracy requirement, we have to make the step size small, which will lead to more time cost.

2. **Not So Accurate If the Robot Hits the Wall**

   If the robot hits the wall at some moment, the particle will still move based on the previous strategy. Thus it cannot represent the current position of the robot accurately until the robot back to its previous track or after one circle.

VI. CONCLUSION

Monte Carlo Localization(MCL) is an algorithm that begins with a set of random hypotheses about where the robot might be all over the map and in any heading. Each of these hypotheses is called a particle, as this technique derives from the general technique called Particle Filtering. In this section, you will implement the MCL algorithm for the autonomous robot DR20. The map and the simulated environment is given. We can update the particle’ s probability based on the expected and the observed readings.

Nowadays we should notice that MCL has made great progress with the development of devices and algorithms. Algorithms have played a more and more important role in helping agents to perfectly finish the tasks. A preliminary analysis of this homework project is of great significance to our understanding of the high-tech frontier technology.

1. **Task 1: Local MCL**

   The initial pose of the robot is assumed to be known. Since the uncertainties are confined to region near the actual pose, this is considered to be a local problem.

2. **Task 2: Global MCL**

   In contrast to local position tracking, global localization assumes no knowledge of initial pose. It subsumes the local problem since it uses knowledge gained during the process to keep tracking the position.

3. **Task 3: Global Realtime MCL**

   We meet the requirements of real-time (up to 1 second interval per iteration). Calculate the time spent each time for robot localization.

All in all, this laboratory gave us an insight into how algorithms work in reality and we hope to be able to apply them to the reality.