AI 3603 Artificial Intelligence: Principles and Techniques

By: 518030910380 Xiaolin Bu
519030910089 Jiaxin Song

HW#: 3

December 15, 2021

I. INTRODUCTION

A. Purpose

The goal of this lab is to implement particle filters for state estimation. The key point is to generate a set of particles in the space, update the particles that are closer to the real state with a higher weight, then regenerate a set of particles according to the weight.

In this assignment, robot DR20 runs in a previously known map. We also know the robot's moving linear and angular velocity. This lab is divided into three parts:

1. In the first part, we know the range of initial pose of the robot. It's a **local** problem!
2. In the second part, we have no knowledge of the initial pose of the robot. It's a **global** problem!
3. In the third part, the robot can randomly move to a unknown position in the environment. It's a **more global** problem!

B. Equipment

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

- OS: Fedora 34
- python 3.8
- CoppeliaSim 4.2

II. LOCAL POSITION TRACKING

A. Description

In the local position tracking problem, we implement Monte Carlo Localization (MCL) algorithm. The basic procedures of MCL are followed:

- **Initial Step.** In the local problem, the initial position of the robot is known. So the algorithm begins with a set of particles generated randomly around the initial position of the robot (denoted as X_0).

- **Update Step.** At time t, the i_{th} $(0 \leq i \leq 4)$ sensor on the robot is used to measure the distances to the obstacle in direction $\theta + i \times \frac{\pi}{4} - \frac{\pi}{2}$ when the bias angle of the robot is θ. Then the five sensors record these distances in vector $data$.

  For each particle: $d_{t,i} = [x_{t,i}, y_{t,i}, \theta_{t,i}]^T, (1 \leq i \leq NP)$, suppose there is a sensor in that particle and calculate the distances to the obstacles in five directions (denoted as $data_i$).Assign the weight $w_i = \frac{1}{||data_i - data||_2}$ to the particle.

  Then we normalize the weight of all particles and use the normalized weight of each particle as the probability that the particle is just at the position of the robot.

- **Re-sample Step**: Choose NP indexes: $id_1, ..., id_{NP}$ according to the last step's probability. Then $d_{t+1,i} = [x_{t,id_i} + rand(0, 0.1), y_{t,id_i} + rand(0, 0.1) + \theta_{t,id_i} + rand(0, \pi)]^T$

B. Implementation

The main difficulty of local MCL is how to assign weight to each particle. Our method is to measure the five distances to obstacles for each particle (as the data for robot) and compare it with the data obtained from the sensor of the robot. Since the map is known, we save the obstacle boundaries as 24 segments $l_{x,i}, l_{y,j}, 1 \leq i \leq 12, 1 \leq j \leq 12$. They respectively represent the left, right, upper and lower boundaries of the obstacles.

For particle $d_{t,i}$ and sensor j , suppose the particle is located at the outside of all obstacles. Consider the intersection of the 24 segments and the ray: $y = x_{t,i} + tan(\theta + j \times \frac{\pi}{4} - \frac{\pi}{2})x, (x \geq x_0)$ and find a intersection point that is closest to the particle. This is the distance from the particle to the nearest obstacle in the direction of the j_{th} sensor. After the information about the particles is obtained, we calculate its Manhattan distance d with the data of the real robot. We use $\frac{1}{d^2}$ to represent the weight. Code to calculate the distance is:

```python
def get_x_nearest_obstacle(x, theta):
    if is_right_direction(theta):
        return int(x*2+1)/2.0
    else:
        return int(x*2)/2.0

def find_cloest_x_obstacle(x, y, theta):
    k = math.tan(theta)
    pos_x = get_x_nearest_obstacle(x, theta)
    while pos_x <= 5.0 and pos_x >= 0:
        pos_y = y + k * (pos_x - x)
        for range_y in obstacles_x[pos_x]:
            if pos_y >= range_y[0] and pos_y <= range_y[1]:
                return pos_x, pos_y
        pos_x += 0.5 if is_right_direction(theta) else -0.5
    return 100, 100
```

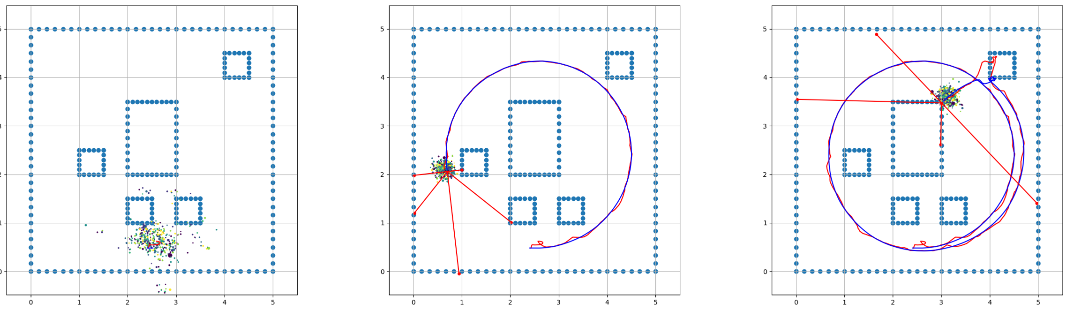
```

def find_cloest_obstacle(x, y, theta):
    obstacle_1_x, obstacle_1_y = find_cloest_x_obstacle(x, y, theta)
    distance1 = math.sqrt((obstacle_1_x - x)**2 + (obstacle_1_y - y)**2)
    obstacle_2_x, obstacle_2_y = find_cloest_y_obstacle(x, y, theta)
    distance2 = math.sqrt((obstacle_2_x - x)**2 + (obstacle_2_y - y)**2)
    return distance1 if distance1 < distance2 else distance2


def detect_obstacles(x, y, theta):
    distance = np.array([find_cloest_obstacle(
        x, y, mod2pi(theta + i * math.pi/4)) for i in range(-2, 3)])

    return distance
```

Otherwise, if the particle is inside a obstacle, the weight of that particle is directly assigned zero since the robot cannot go into a obstacle and the particle should not be in there. The weight vector is normalized after all weights are generated. Also, because the speed and the sensor of the robot has some randomness and noise, we add noise to the speed of the particles when making the particles moving along with the robot.

Another difficulty is the particle resampling according to the weight of the previous particles. We randomly choose a particle in the old particles according to the weight distribution, i.e. with a higher weight, the particle is more likely to be chosen. Then, the particle is added by a random bias (from -0.1 to +0.1) to generate a new particle.

C. Result Visualization and Analysis

The figure on the left shows the initial particles, the middle one shows the particles during updating and the right one shows the final state of the particles. The red and blue curves respectfully represent the estimated path and the true path, where the estimated path is the weighted average of the particles' position. The radius of a particle in the plot represents the particle's weight. (Colors of the particles are random).



(a)Initial Particles (b)Updating Particles (c)Final Particles

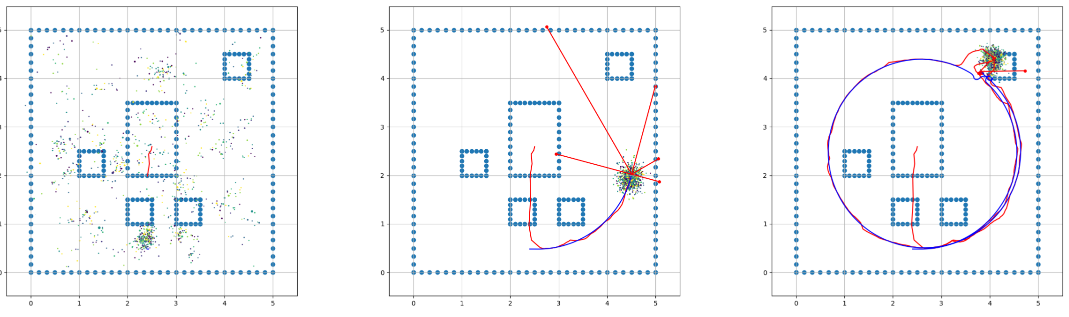
FIG. 1: Local MCL Result Visualization

Analysis It can be seen that the particles are initially generated around the robot and the estimated path converges very quickly. When the robot moves, the particles also move along the path and the estimated path fits the real path well. In Figure 1 (b), the robot hits the wall, yet the particles cannot figure out which wall does the robot hit because data in some directions is missing, hence they gather at a closest wall.

III. GLOBAL POSITION TRACKING

In this section, we have no knowledge about the initial pose of the robot. Due to this reason, the initial particles X_0 are distributed uniformly in the whole map.

We generate 1000 particles and our algorithm in Local MCL still works fine for Global MCL. This is because the particles distributed around the initial position of the car can get a higher weight, then the algorithm is more likely to generate particles around these particles in re-sample phase. The results can be seen below in Figure 2.



(a)Initial Particles (b)Updating Particles (c)Final Particles

FIG. 2: Reward Decay Visualization

Analysis It can be seen that the initial particles are distributed throughout the map due to none knowledge about the initial pose of the robot. The initial estimated position is in the middle of the map as the start point of the red line shows, because all the particles are distributed normally with the same weight. Although the initial estimated position is far from the real position of the robot, the algorithm also converges fast. The particles gather and find a true path in about 6 to 8 iterations.

Also, we can see from the radius of the particles that the particles with a higher weight is distributed around the real robot. However, the weight of the particles that are near to the real position can also be small. This is because the particles may have a wrong orientation despite the right position.

IV. (BONUS) KIDNAPPED ROBOT PROBLEM

In this section, the robot may be move randomly rather than always follows a fixed path. Therefore, particles may not be able to keep up with the movement of the robot if we always use the previous particles each time to re-sample. This is why we need the ACML algorithm.

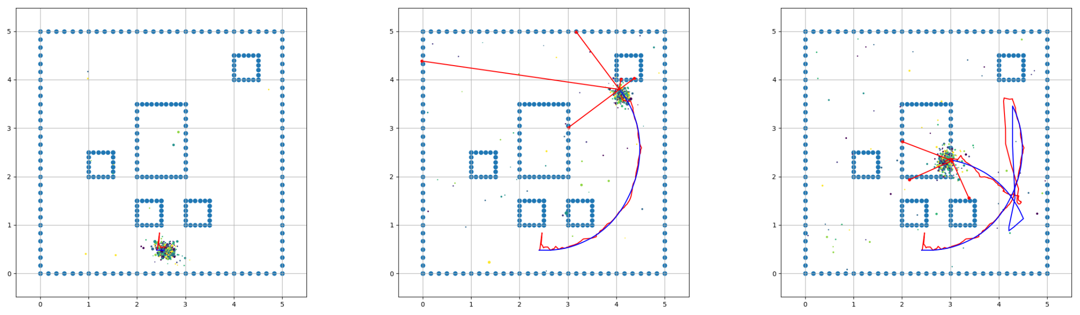
The key idea of ACML algorithm is to generate some random particles at a random probability during the re-sample phase each time.

A. Implementation

I set $\alpha_{slow} = 0.7$ and $\alpha_{fast} = 0.3$ and $w_{slow} = w_{fast} = 0$. At each iteration, we still set $||data_{t,i} - data||_2$ as the weight for each particle $x_{t,i}$ and choose $\frac{\sum_{i=1}^{NP} ||data_{t,i} - data||_2}{NP}$ as w_{arg}.

```python
def pf_localization(px, pw, data, u):
    """
    Localization with Particle filter. In this function, you need to:
    """
    global w_slow, w_fast
    global alpha_fast, alpha_slow

    w_avg = 0
    t1 = time.time()

    for ip in range(NP):
        #  Prediction step: Predict with random input sampling
        ud = randomize_input(u)
        par = px[:, ip].reshape(3, 1)
        pred = motion_model(par, ud)
        px[:, ip] = pred.reshape(3)

        pw[0][ip] = 1 / \
            np.linalg.norm(detect_obstacles(
                px[0][ip], px[1][ip], px[2][ip]) - data)
        #  Update steps: Calculate importance weight
        w_avg = pw.sum()/NP

    pw = pw / pw.sum()  # normalize
    w_slow += alpha_slow * (w_avg - w_slow)
    w_fast += alpha_fast*(w_avg - w_fast)
    x_est = px.dot(pw.T)

    print("w_avg", w_avg, "alpha_slow", alpha_slow, "alpha_fast", alpha_fast)
    # Resample step: Resample the particles.
    px, pw = re_sampling(px, pw)

    print("The time used for each iteration:", time.time()-t1, " s")
    return x_est, px, pw


def re_sampling(px, pw):
    """
    Robot generates a set of new particles, with most of them generated around
        the previous particles with more weight.
    """
    ### START CODE HERE ###
    global w_slow, w_fast

    indices = np.array([i for i in range(NP)])
    pw_indices = np.random.choice(indices, NP, replace=True, p=pw[0])
    random_indices = np.random.choice(indices, max(
        0, (int)(NP*(1.0-w_fast/w_slow))))
    proposol_particles = np.array(
        [[px[j][pw_indices[i]] + (np.random.rand()-0.5)/5 for i in range(NP)]
            for j in range(3)])
    print(w_fast, w_slow, len(random_indices))
    for idx in random_indices:
        proposol_particles[0][idx] = np.random.uniform(
            0, 5)
        proposol_particles[1][idx] = np.random.uniform(
            0, 5)
        proposol_particles[2][idx] = np.random.uniform(0, math.pi)
    ###  END CODE HERE  ###
    return proposol_particles, pw
```

B. Result Visualization



(a)Initial particles (b)Updating particles (c)Move the robot halfway

FIG. 3: ACML algorithm when $\alpha_{slow} = 0.7$ and $\alpha_{fast} = 0.3$

Analysis In FIG 3, we show the status of initial particles, updating particles and particles after the robot move a distance. The particles still keep in with the robot after the robot move a distance, which achieves the aim of ACML algorithm. Moreover, we also find some meaningful scenes during the ACML running. We find that there are some random particles almost every moment. The number of random particles will decrease when the robot‘s route is smooth while it will increase when the robot suddenly turns or moves a long distance.

a. Some Limitations Though the particles track the robot well in these three cases, there are still some counterexamples that ACML algorithm can not solve:

- *The final particles may not converge to one group!* When the robot suddenly move a long distance, the final particles may converge to several groups. This is because these particles may encounter the similar terrains and the algorithm will converge locally.

- *It's difficult to select α_{slow} and α_{fast}.* If we set a small value to α_{slow} and α_{fast}, ACML will be less likely to choose random particles, which will make the particles cannot track the robot's movement. However, if we set a large value to α_{slow} and α_{fast}, some small changes will result in large changes to particles.

V. CONCLUSION

In this lab, we focus on Local MCL, Global MCL and ACML. The problems they targeted are more and more generalized and the approaches they used are similar which only need some modifications. They can all track the car well in their respective situations. However, due to the limitations of the algorithm and our implementation, we meet some problems when the robot hit the wall or suddenly move a long distance. We expect we can learn more general algorithm to track the robot more precise.