ECE 498SMA: Principles of safe autonomy	Spring 2020
Written by: Yangge Li, Ted Liu, Qichao Gao, Sayan Mitra	*MP 4: Particle Filter*
Course Website	*Due Date: 2020/4/3*

1 Introduction

You are stuck in ECEB and want to use the algorithm you implemented in MP3 to find your way out. However, now you don't know you exact location in the building. Fortunately, you have a detailed map of ECEB. Therefore you plan to use a particle filter to find your position.

In this MP, you will need to implement the Monte Carlo Localization (MCL) discussed during the lecture. You will have a vehicle constantly running in gazebo simulator and you need to find the position of the vehicle using the sensor reading from the vehicle and the map of the environment. The estimated position of the vehicle will be displayed on the map. You will need to play around with some parameters of the particle filter and check the effect of those parameters. This is passive localization, that is, the vehicle motion is controlled independently and you cannot change it for localization.

The provided code can be found at this git repo. All your code should be in the file `particle_filter.py` and you will have to write a brief report.

In the MP you will learn how to use particle filter to perform localization. This document gives you the first few steps to get started on implementing the particle filter algorithm. As usual, you may have to take advantage of online tutorials. Cite all resources in your report. **All the regulations for academic integrity and plagiarism spelled out in the student code apply.**

Learning objectives

- Particle filters
- Localization
- Measurement models

System requirements

- Ubuntu 16
- ROS Kinetic
- ros-kinetic-ros-control
- ros-kinetic-effort-controllers
- ros-kinetic-joint-state-controller
- ros-kinetic-ackermann-msgs

2 Module architecture

This section describes components that are important for this MP. However, in this MP you only need to implement some of the components. The components marked by * are not *required* for you to implement, but feel free to check them. In fact, as class project may be utilizing a similar simulation framework, learning about these components may help with your projects. The overall architecture of this MP is shown in figure 1. Detailed explanation for each of the components in the figure can be found in the following section

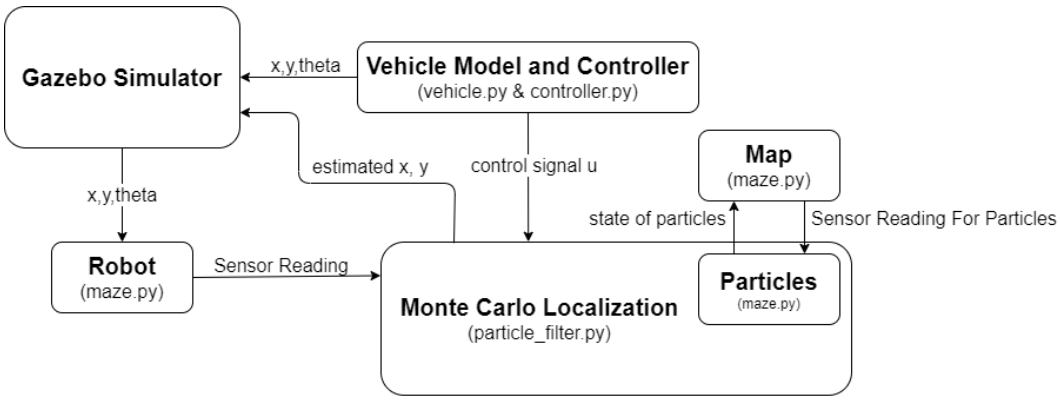

Figure 1: Architecture of this MP. Map, Robert, Particles module are located in maze.py

2.1 Gazebo simulator*

As usual, this MP will rely on the gazebo simulator. In the gazebo simulator, we have created an environment for the first floor of ECEB. Feel free to take nostalgic virtual tours of your favorite cafe and machine shops as you WFH. Gazebo will acquire the position and orientation of the vehicle from the vehicle model and controller module and display the state of the vehicle in the ECEB environment. The Gazebo simulator will constantly send the current state (position and orientation) of the vehicle to the robot module so the robot odule is able to send its sensor reading to MCL. As your particle filter calculates the location of the vehicle, it will show the estimated location in the gazebo simulator as a marker on the ground.

2.2 Vehicle model and controller*

These two modules are implemented in `vehicle.py` and `controller.py`. These two modules drive the vehicle constantly in the gazebo simulator through a series of waypoints by computing the current position and orientation of the vehicle and send the information to gazebo simulator. You should not modify these two modules.

2.3 Robot and Particle*

Robot and Particle are two classes defined in the `maze.py` file. The particle class, as its name suggests, defines the particle that is used in the particle filter algorithm. It has properties recording its x, y position, its orientation (heading), a weight and the map of the environment, which is the maze. The particle also has a model of the sensor on the robot that can calculate the distance between particle and nearby wall. Details about how the sensor model works will be discussed in section 3.1.3.

The robot class stores the state of the actual vehicle in the gazebo simulator. It is a derived class of the particle class. The robot will acquire its exact state from the gazebo simulator. The robot have a sensor on it that can measure the distance between the robot and nearby wall. However, the reading from the sensor

have noise and may not be accurate. Details about how the sensor model works will be discussed in section 3.1.3.

2.4 Map*

This module is located in the `maze.py` file. This module encodes the map of the environment that the vehicle stays in. The estimated state of the vehicle will be displayed in this map. Also, the sensor reading from the particles will be based on this map.

2.5 Monte Carlo Localization

This module is located in the `particle_filter.py` file. This is the main module you will working on. This module contains the implementation of the Monte Carlo Localization that is based on the sensor reading from the robot, sensor reading from each particle from the map, and the control signal from vehicle model and controller. In addition, the MCL will hold a list of particles. The output from this module is the estimated position of the vehicle in the ECEB environment. In this MP, the MCL resides in the `runFilter` function. You will need to implement this function together with some helper functions to do the calculations. Detailed guide on how to implement this module is given in the next section.

3 Development instructions

3.1 Monte Carlo Localization (MCL)

As already talked about during the lecture, given a map of the environment, MCL can be used to approximate the posterior probability distribution of current location based on motion models and measurement updates. The algorithm should holds a list of uniformly random generated particles in initialization. Then as the vehicle moving, the algorithm will shifts the particles to predict its new state after the movement. With the sensor reading from the vehicle, the particles are weighted and resampled based on how well the actual sensed data correlated with the predicted state. The algorithm will run iteratively and ultimately, most of the particles should converge toward the actual state of the vehicle.

3.1.1 runFilter

The algorithm will be implemented in the runFilter function. The steps in this function are straightforward. You only need to constantly loop through three steps as shown in 3.1.1. Suppose $p = \{p1 \dots pn\}$ are the particles representing the current distribution:

```
def runFilter
    while True:
        sampleMotionModel(p)
        updateWeight(p)
        p = resampleParticle(p)
```

The weighted sum of the position and orientation stored in particles will determine the estimated position and orientation of the vehicle and will be displayed on the map.

3.1.2 sample motion model

In this part of the algorithm, the robot needs to predict its new location based on the actuation command (control input) given, by applying the simulated motion to each of the particles.

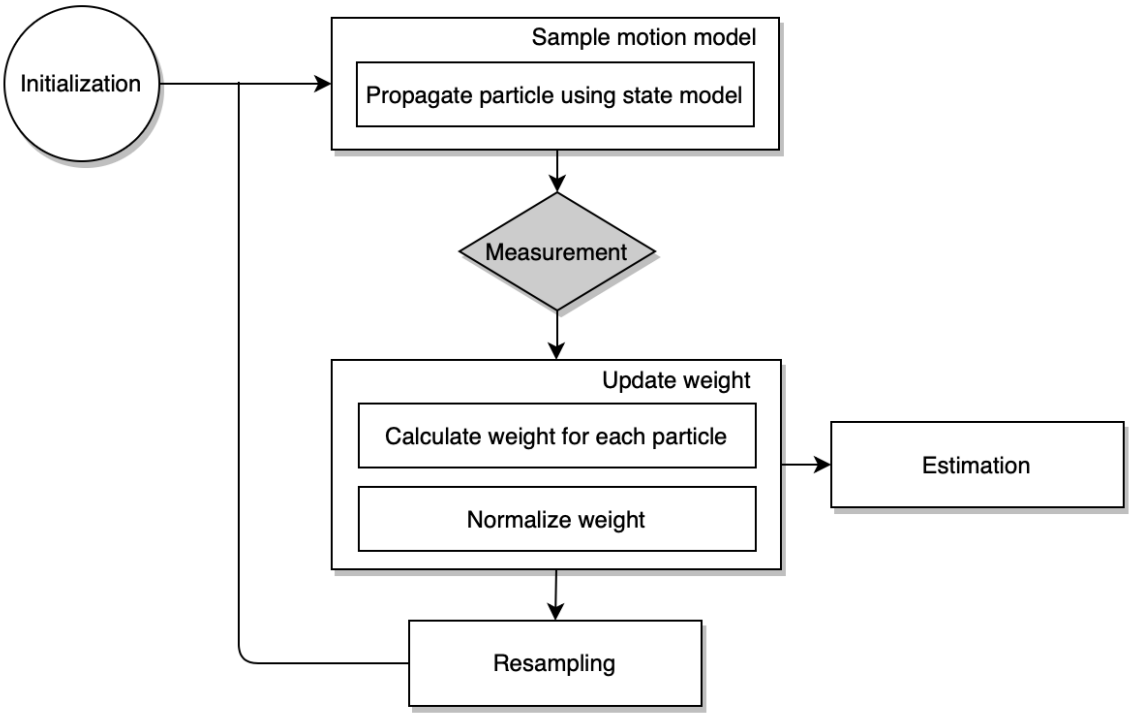

Figure 2: Overall flowchart of a MCL based algorithm.

For this MP, the vehicle have the same dynamics as MP2, which is given as

$$\begin{aligned} \dot{x} &= v_r cos(\theta) \\ \dot{y} &= v_r sin(\theta) \\ \dot{\theta} &= \delta \end{aligned} \tag{1}$$

Where v_r and δ are the control signal calculated by the controller.
In this MP, the vehicle will constantly publish the control signal (v_r and δ) in ros topic `/gem/control`. The particle will subscribe to that topic and record the control signal in list `self.control`. The time duration for each control signal is $0.01s$.
With all the above information, we are able to simulate the movement for each particle. We will use the numeric integrator package `ode` in the `scipy.integrate` package to perform the simulation. Detailed documentation for how to use the integrator can be found here. In this case, the initial condition for each particle is the current state of the particle. You should perform integration iteratively with the series of control input stored in `self.control` with time step 0.01. By doing this, you can properly predict the new location of the particle.

3.1.3 Sensor model and weight updates

After propagating particles with state model and control command, we now need to assign a new weight for each of them according to the measurement from the sensor.

There are four sensors installed on the front, back, left and right of the GEM module. The sensors measure its **perpendicular distances** to the closest walls in four directions in order of (front, right, back,

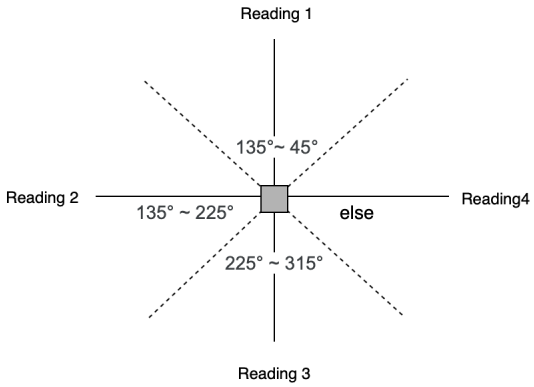

Figure 3: Sensor on GEM.

left) with respect to the local frame of the vehicle. To further simplify the problem, the orientation of the vehicle while reading sensor is quantized into four 90 degree region. For example, if the orientation of the vehicle is between -45 degree and 45 degree, it will be treated as 0 and if the orientation of the vehicle is between 45 degree and 135 degree, it will be treated as 90. Therefore, for example, if the heading of the vehicle is between 135 degree and 225 degree, `read_sensor` returns list [Reading 2, Reading 1, Reading 4, Reading 3]. In reality, sensor reading always have some noise, thus our robot measurement from `read_sensor` is not noise free and it contains some level of noise. To have a more accurate simulation, this mp also take sensor limit into account. Each sensor has a certain effective range and it cannot recognize obstacles that is not within its range. `sensor_limit` is an argument that can be changed when you run the python file. Details about how to change it will be discussed in section 3.3.

To assign the weights to the particles, we need to compare the similarity between the real sensor measurements and the particle sensor measurements. In this MP we will use `self.weight_gaussian_kernel` to calculate the likelihood between the two sensor readings. This function takes in two sensor readings, ideally one from the vehicle and the other from the particle, and a standard deviation value, default value set to 10. It will compute and return the weight of particle base on the Gaussian kernel. Please note, the weights need to be normalized before get into the resampling step.

3.1.4 Resampling particles

In this part of the code, you are supposed to implement function `resampleParticle()` to resample particles according to the weight calculated from the section above to replace old set of particles. The newly sampled particles will be more likely to reside around the partiles that have higher weight. It is recommended to first start implementing this function using multinomial resampling method:

1. Calculate an array of the cumulative sum of the weights.
2. Randomly generate a number determine which range in that cumulative weight array to which the number belongs.
3. The index of that range would correspond to the particle that should be created.
4. Repeat sampling until you have the desired number of samples.

Implement and compare at least **TWO different** resampling techniques then put your result in report. Feel free to declare any additional function in `particle_filter.py` for different resampling methods. For more information about particle filter resampling techniques you can checkout this paper [1].

3.2 Gazebo Environment and Map

In order to reduce the amount of computational power required for the particle filter, we restrict the vehicle to move in the north west corner of the ECE building (region around machine shop) as showing in figure 4. More specifically, we choose a rectangle region with width 120, height 75, and bottom left corner at position

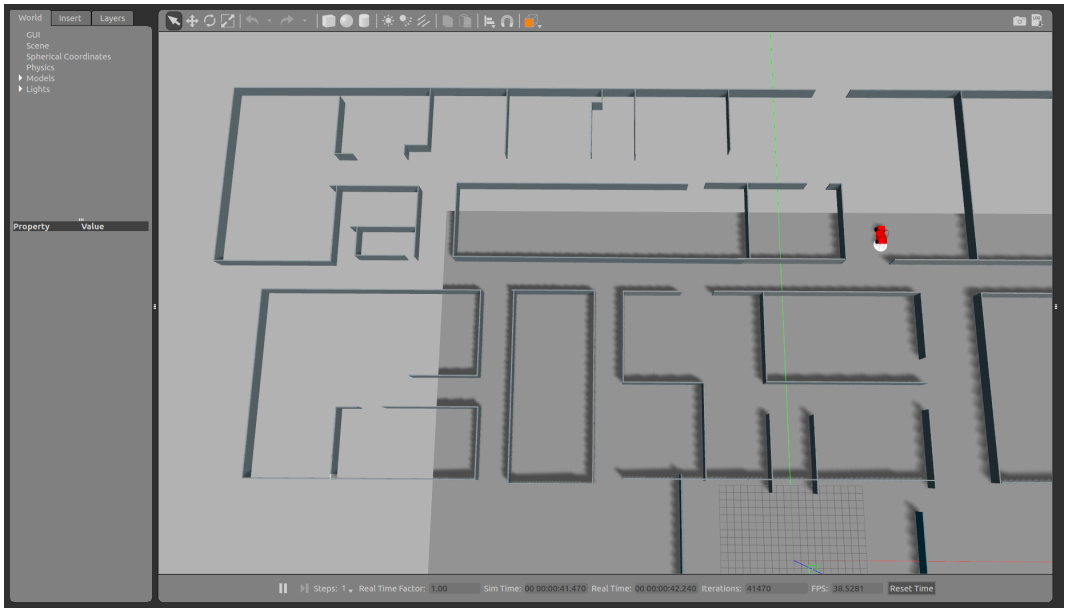

Figure 4: North west corner of ECEB. The vehicle is at its default location

$(x, y) = (-85, 45)$. Because of that, the map we have only represent that region as shown in fig 5. Therefore, all the particles we choose should from that region. Moreover, to better discretize the map to get better accuracy, each grid in the map is divided into 100 segments horizontally and vertically. Therefore, a point $p = (x, y)$ in the gazebo simulator can be converted to a point $\bar{p} = (\bar{x}, \bar{y})$ as shown below:

$$\begin{aligned} \bar{x} &= (x + 85) * 100 \\ \bar{y} &= (y - 45) * 100 \end{aligned} \tag{2}$$

3.3 Running experiment

For this MP, you should be able to start the gazebo environment using command

```
roslaunch mp4 mp4_eceb.launch
```

Then you should start running the vehicle in the environment using command

```
python vehicle.py
```

if the vehicle is at the default location when the gazebo is launch as shown in figure 4. If the vehicle is not at that starting location, you should first run

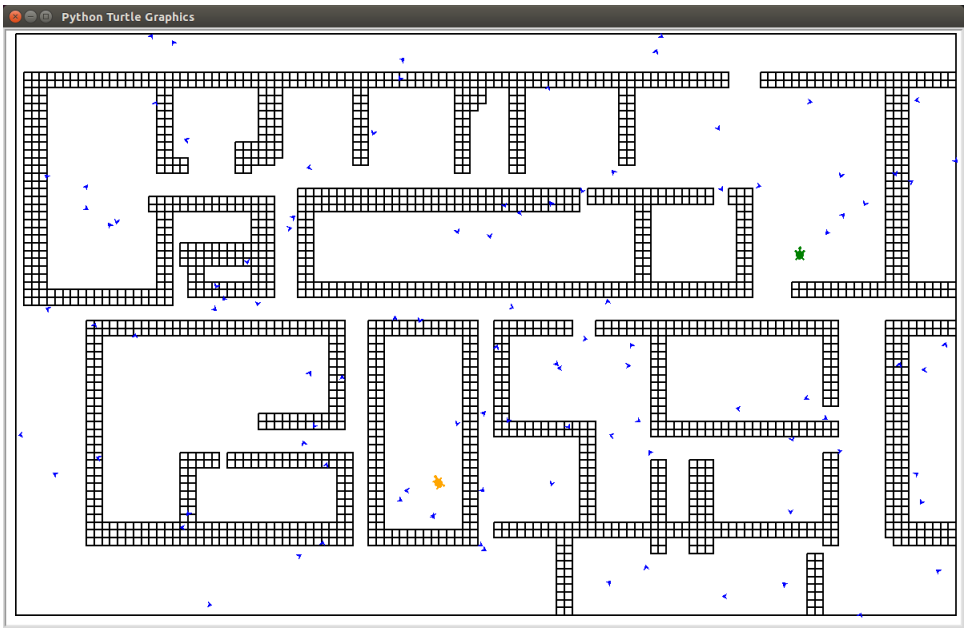

Figure 5: Maze map in bird's eye view.

```
python respawn.py
```

with **no argument** before running the vehicle or the vehicle may not be able to follow the waypoints. With the car running, you can now run the particle filter by using command

```
python main.py --num_particles 1000 --sensor_limit 5
```

The particle filter can take arguments

1. num_particles – This parameter defines the number of particles used in the particle filter. The default value of this parameter is 1000
2. sensor_limit – This parameter defines the distance the sensor can see. If the distance between the vehicle and a wall is beyond the sensor limit, the sensor will not be able to see the wall and therefore will return value equal to sensor limit. The default value of this parameter is 5.

When the Particle filter is running, you will be able to see a python turtle window pop up with the map in it as shown in figure 5. You should be able to see two turtles in the map, which corresponds to the actual and estimated state of the vehicle. The actual state of the vehicle is represented by the green turtle and the yellow turtle represents the estimated state of the vehicle. The estimated particles are represented by blue arrows. If you implementation is correct, you should be able to see the particles (blue arrows) converge to the actual position of the vehicle (green turtle).

4 Report

1. How is the number of particles influence the performance of the particle filter. Plot the error in position estimation (euclidean distance between actual x, y position and the estimated position) as a function of algorithm iteration, for three different choices of the number of particles. Since particle filtering is a randomized algorithm, run several instances of the same setup and plot the average error for each time.
2. Plot a similar figure for the error in the heading θ.
3. What is the trade-off between the estimation accuracy and the computational load, as the number of particles is changed.
4. How is the quality of the sensor influence the performance of the particle filter. Make plots for the position error for three different sensor limit values.
5. How do different resampling methods influence the particle filtering/localization?
6. How would you deal with the kidnapped robot problem in your code?
7. You may notice that the orientation of the estimated vehicle is sometimes opposite to the actual orientation of the vehicle when the vehicle is heading to the right ($\theta = 0$). Why does this happen? Hint: read function `show_estimated_location` in maze.py and see how the estimated orientation is calculated. **Bonus.** Can you find a general enough solution to this problem?

5 Submission instructions

Please record a video showing the behavior of the particle filter-based localization algorithm for two different settings for the number of particles. The video should show the gazebo simulation, the python map with vehicle position estimates side-by-side, with a plot of the position and heading errors. Please pack your report, your code and the video in a zip file and submit to Compass 2G by one of the members in your group.

6 Grading rubric

- 40%: The particle filter implementation: code and video
- 60%: Report
- 5%: Bonus point for solving orientation problem

References

[1] J. D. Hol, T. B. Schon, and F. Gustafsson. On resampling algorithms for particle filters. In *2006 IEEE Nonlinear Statistical Signal Processing Workshop*, pages 79–82, Sep. 2006.