ECE 484: Principles of Safe Autonomy Spring 2024

Written by: ECE 484 Team

MP 3: Filtering and Localization *Website* Due Date: ***March 22, 2024***

1 Introduction

You are stuck in ECEB and want to find your way out. However, now you don't know your 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.

Same as the previous MPs, this MP is divided into two different parts. In the first part of this MP (section 2), you will be working on some theoretical problems about particle filter and localization which can help you better understand the algorithm we talked about during the lecture. In the second part of this MP (section 3), 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.

For part one, you will solve Problems 1-3. For part two, you will need to submit your modified lidarProcessing.py, maze.py, particle_filter.py code and the solution to Problems 4-8. Name all the group members and cite any external resources you may have used in your solutions. More details for submission are given in Section 4. 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 20
- ROS Noetic
- ros-noetic-ros-control
- ros-noetic-effort-controllers
- ros-noetic-joint-state-controller
- ros-noetic-ackermann-msgs

2 Written Problems

Problem 1 (20 points)**.** Following is derivation steps for the Bayes filter update rule

$$P(x|y,z) = \frac{P(y|x,z)P(x|z)}{P(y|z)} \tag{1}$$

using the conditional probabilities. **Fill in the blanks**. *Hint: The box with same number should have the same answer.*

$$\begin{aligned} P(x|y,z) &= \frac{P(x,y,z)}{\boxed{1}} \\ &= \frac{P(x,y,z)}{\boxed{2}P(z)} \\ &= \frac{\boxed{3}P(x,z)}{\boxed{2}P(z)} \\ &= \frac{\boxed{3}\boxed{4}P(z)}{\boxed{2}P(z)} \end{aligned} \tag{2}$$

Problem 2 (50 points)**.** An autonomous robot is navigating through a 1D world represented by five positions. The robot uses the particle filter to keep track of its position.

(a) **Probabilistic Motion Model** The robot has a prior belief of being in position $x = 3$ with probability 0.4, $x = 2$ with probability 0.5, and $x = 4$ with probability 0.1. Compute the posterior probabilities for its location if a control input potentially moves the robot one position to the right with a transition probability of 0.8, and there's a 0.2 probability it remains in the same position.

(b) **Particle Weight Update** The positions $[2, 4]$ have a unique landmark. The robot has a sensor that detects these landmarks with a probability of 0.8 when it's at the correct position, and gives a false positive (detects a landmark when there isn't one) with a probability of 0.1 at other positions. The initial positions of the particles are $[1, 3, 4]$. A control input moves the robot one position to the right with a 0.7 probability, and there's a 0.3 probability it remains in the same position.

"After the movement, the robot's sensor detects a landmark."

Update the particle weights to reflect this sensor measurement.

Problem 3 (30 points)**.** Recall in MP0 you proved safety of Automatic Emergency Braking scenario using invariants. Now let's add localization into the equations. Imagine a scenario that a connected autonomous vehicle and a pedestrian's phone can share information about their locations wirelessly. The localization methods on the vehicle and on the phone both rely on GPS. However, due to the differences in the receiver size and performance, the two have different accuracies. Similar to MP0, let's assume the vehicle starts from origin and cruises down a straight road in the x-direction. The car has an initial velocity $v_0 = 5m/s$, and the deceleration rate $a_b = 5m/s^2$. Assume the vehicle has no reaction delay and infinite sensing distance thanks to the connectivity. The pedestrian is still static. Only this time, you don't have ground truth for the locations of the vehicle and the pedestrian. Let's model the sensor noise of the GPS receivers using additive Gaussian noise model. The measurement y_{car} of the car GPS can be modeled as $y_{car} = x_{car} + n_{car}$, where x_{car} is the true location of the vehicle and n_{car} is the Gaussian noise. Similarly, $y_{human} = x_{human} + n_{human}$. And from the receiver information, we know that $n_{car} \sim N(0, 1.5)$ and $n_{human} \sim N(0, 6)$. **For this question, you don't need to do formal invariant proofs.**

(a) You would like to take no chances and make sure the car is 100% safe. Fortunately, you know the GPS sensor noises are truncated at a maximum absolute value, i.e., $|n_{car}| \leq 15$, $|n_{human}| \leq 60$. (We will ignore the normalization of distribution for this question.) Now given y_{car} and y_{human}, what's the minimum safe braking distance needed between the two to ensure safety?

(b) Suppose we use our conservative safe braking distance from (a), but now have perfect position estimates, i.e., $y_{car} = x_{car}$, $y_{human} = x_{human}$, what is the final distance between the vehicle and the pedestrian? Is this policy useful in practice? Explain in 20 words.

(c) Now you want to take some risks. Instead of using the maximum value truncation, you instead use your knowledge in probability that 99.7% of the noise fall into 3σ of the $N(0, \sigma^2)$ distribution. Now let's do the same analysis again, what's the minimum safe braking distance between y_{car} and y_{human} that you must brake to ensure safety? Is it better?

3 Implementing MCL to localize in ECEB environment

3.1 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

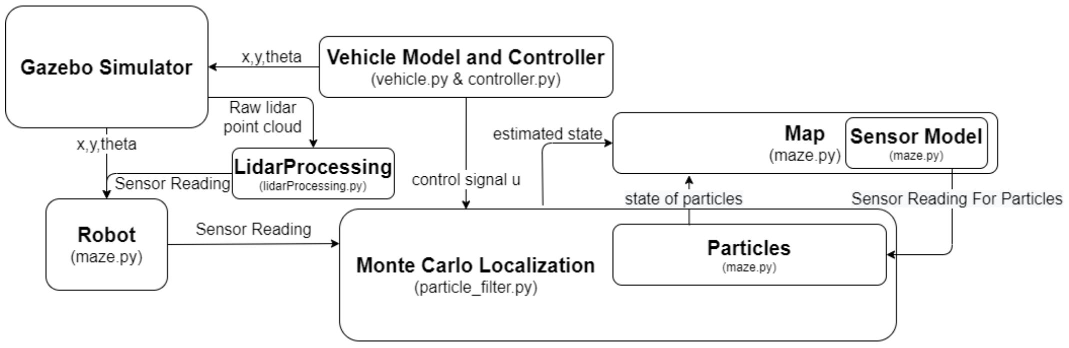


Figure 1: Architecture of this MP. Map, Robot, Particles module are located in maze.py

3.1.1 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. In addition, it will send the current state of the vehicle (position and orientation) to the robot module so that the actual position of the vehicle can be visualized on some map for debugging.

In addition, the Lidar sensor on the simulated GEM vehicle will constantly publishing the raw point cloud Lidar reading from the Gazebo simulator.

3.1.2 LidarProcessing

This module is located in the lidarProcessing.py file. This module takes the raw point cloud data from the Lidar simulator in Gazebo and process the point cloud data so they can be used in other parts of this MP. The provided LidarProcessing module will only look at 4 directions, (front, rear, left, right) directions of the of the vehicle and will return the distance between the vehicle and wall in those directions. In this MP, you will need to modify this module to extend the features of Lidar. Some detailed explanation about how this module works will be discussed in section 3.2.4.

3.1.3 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.

3.1.4 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 that contains a model for the sensor.

The robot class stores the state of the actual vehicle in the Gazebo simulator. It is a derived class of the particle class. Different from the particles, instead of having a sensor model, the robot will get sensor readings from the real sensor from the LidarProcessing module.

3.1.5 Maze

This module is located in the `maze.py` file. This module encodes the map of the environment that the vehicle stays in. The actual and estimated state of the vehicle will be displayed in this map.

In addition, the maze has a model of the Lidar. Same as the simulated Lidar on the GEM car, the sensor model will return the distance between particle and wall in front, rear, left and right direction. In this MP, you will need to modify the sensor model to extend the feature of it. Details about how the sensor model works will be discussed in section 3.2.4.

3.1.6 Monte Carlo Localization

This module is located in the `particle_filter.py` file. This is the main module you will be 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.

3.2 Development instructions

3.2.1 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 hold 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.

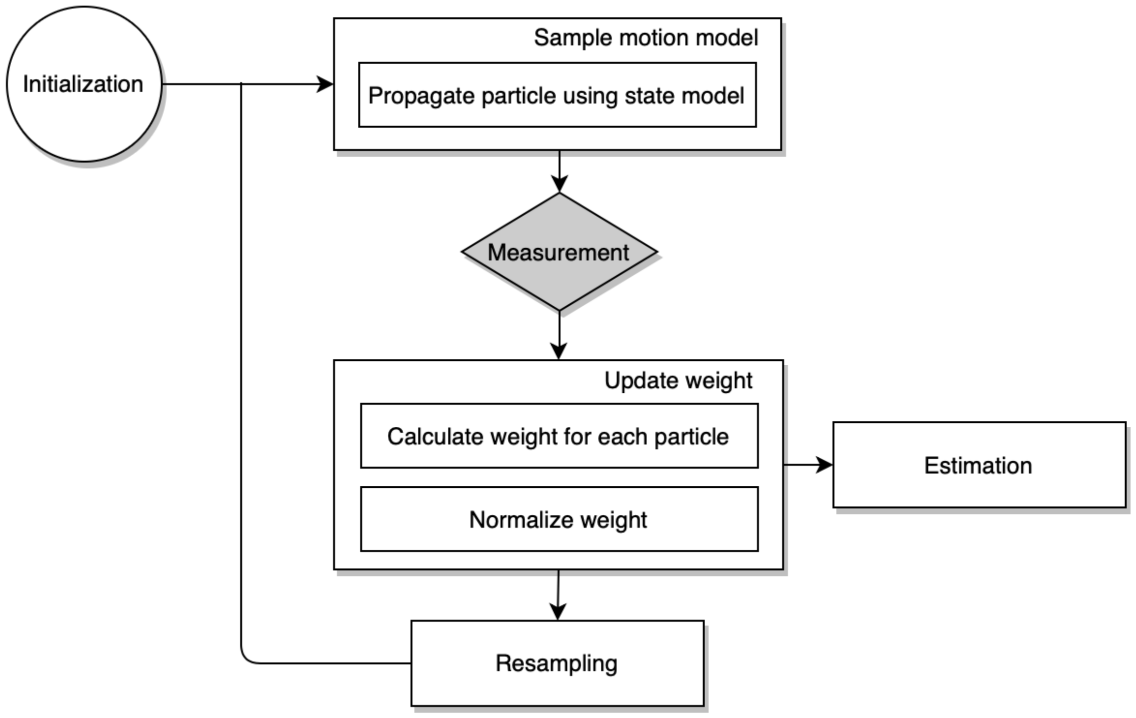


Figure 2: Overall flowchart of a MCL based algorithm.

3.2.2 runFilter

The algorithm will be implemented in the runFilter function. The steps in this function are straightforward. You only need to constantly loop through the steps as shown in 3. Suppose $p = \{p1 \dots pn\}$ are the particles representing the current distribution:

```
def runFilter
    while True:
        sampleMotionModel(p)
        reading = vehicle_read_sensor()
        updateWeight(p, reading)
        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.

3.2.3 sample motion model

In this part of the algorithm, each particle needs to predict its new location based on the actuation command (control input) from the real vehicle.

For this MP, the vehicle dynamics is given as

$$\begin{aligned} \dot{x} &= vcos(\theta) \\ \dot{y} &= vsin(\theta) \\ \dot{\theta} &= \delta \end{aligned} \tag{3}$$

x, y, θ are the x position, y position and orientation of the vehicle. Velocity v and steering angle δ are the control signal calculated by the controller.

In this MP, the vehicle module will constantly publish the control signal ([v, δ]) in ROS topic `/gem/control`. The particle filter will subscribe to that topic and record the control signal in list `self.control`. The time step 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 with example for how to use the integrator can be found here. In this case, the initial condition for each particle is the current state of that particle. You should perform integration through the whole list 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.

3.2.4 Sensor model and weight updates

Sensor model 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.

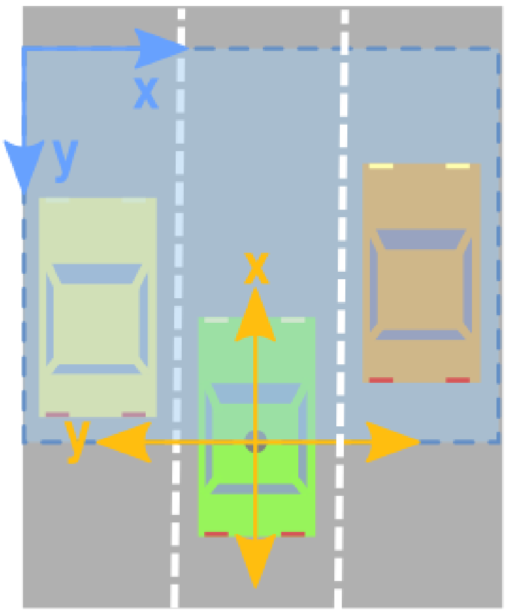


Figure 3: The picture shows the reference coordinate frame of the vehicle. The provided LidarProcessing will measure distance in front, right, rear, left direction. Picture from here

The sensor reading of the simulated vehicle is coming from the Lidar. The Gazebo simulator will constantly publish raw point cloud data from the simulated lidar. Detailed explanation about point cloud data can be found here. The LidarProcessing module will subscribe to the raw point cloud data. It will then process the data and calculate the distance between the vehicle and the wall in front, right, rear and left direction in other words, 0, 270, 180, 90 degree with respect to the vehicle frame as shown in figure 3. Detailed explanation about how the point clouds data are processed can be found here.

To run the MCL algorithm it's also necessary to have the sensor reading for each particle. The sensor model will based on the python map of the environment, and can provide sensor reading for each particle. Same as the Lidar on the simulated vehicle, the sensor model can measure the distance between particle and wall in front, right, rear and left direction. In the current implementation of sensor model, the distance are obtained by counting in step of 1 at each direction until reach the wall.

To have a more accurate simulation, both the simulated Lidar on the vehicle or the senor model have a sensing limit. Both the sensor on vehicle and sensor model has a certain effective range and it cannot recognize obstacles that is not within its range. If in the direction, no obstacle is sensed within sensor limit, the returned sensor reading will be the sensor limit. `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 3.4.

For part of this MP, you are supposed to extend the provided LidarProcessing and sensor model. You will need to add 4 additional measurement directions (total of 8 measurement directions) to the Lidar and sensor model and compare the performance with using only 4 meausurement directions.

Update weight 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 recommend using a Gaussian Kernel to calculate the likelihood between the two sensor readings. You can compute the difference between the sensor reading from Lidar on the vehicle and readings from sensor model for each particle and feed the result into the Gaussian kernel to get the weight value. You can also implement any weight assignment method you prefer. Please note, the weights need to be normalized before getting into the resampling step.

3.2.5 Resampling particles

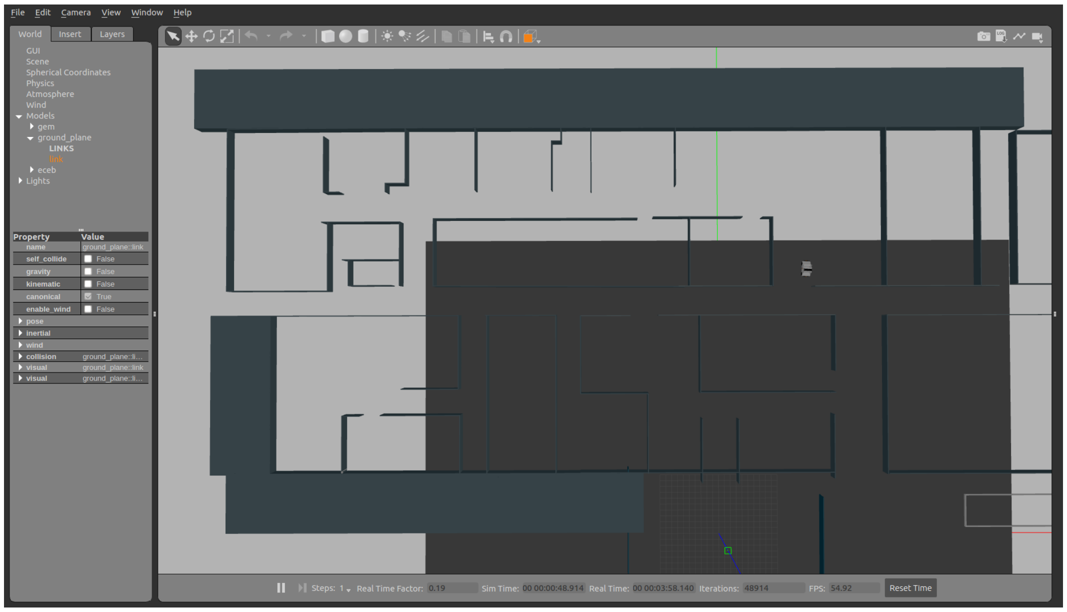
In this part of the code, you are supposed to implement function `resampleParticle()` to resample particles according to the weight calculated from the section above to replace old set of particles. The newly sampled particles will be more likely to reside around the particles that have higher weight. 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 and 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.

Feel free to explore other resampling method that can have better performance. Some additional resampling algorithm can be found here.

3.3 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.



Property	Value
name	ground_plane::link
self_collide	False
gravity	False
kinematic	False
canonical	True
enable_wind	False
pose	
inertial	
wind	
collision	ground_plane::li…
visual	ground_plane::link
visual	ground_plane::li…

Steps: 1 Real Time Factor: 0.19 Sim Time: 00 00:00:48.914 Real Time: 00 00:03:58.140 Iterations: 48914 FPS: 54.92

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

More specifically, we choose a rectangle region with width 120, height 75, and bottom left corner of the region is at position $(x, y) = (-85, 45)$ in Gazebo. 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. A point $p = (x, y)$

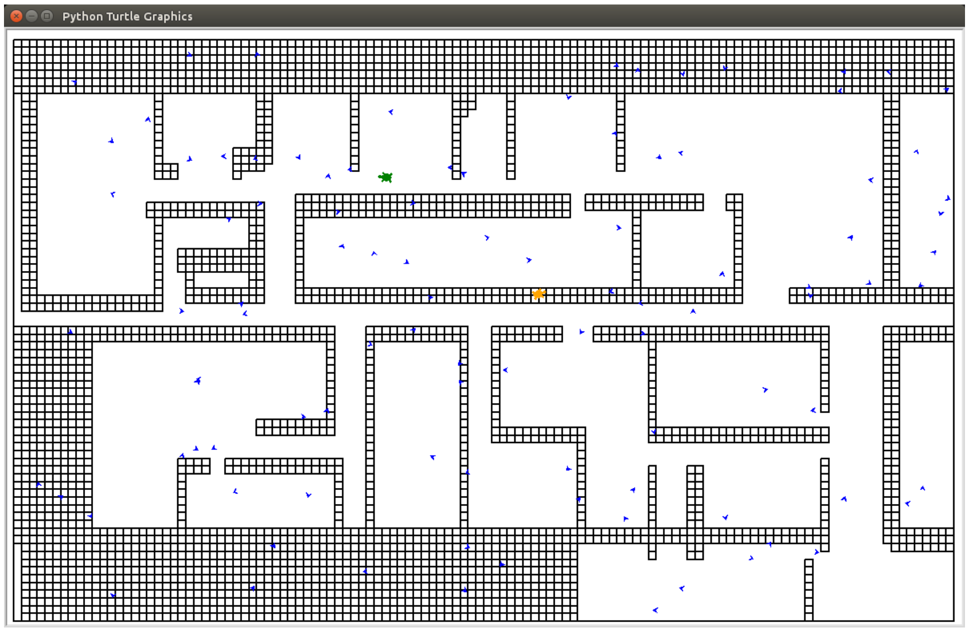


Figure 5: Maze map in bird's eye view.

in the Gazebo simulator can be converted to a point $\bar{p} = (\bar{x}, \bar{y})$ in python map as shown below:

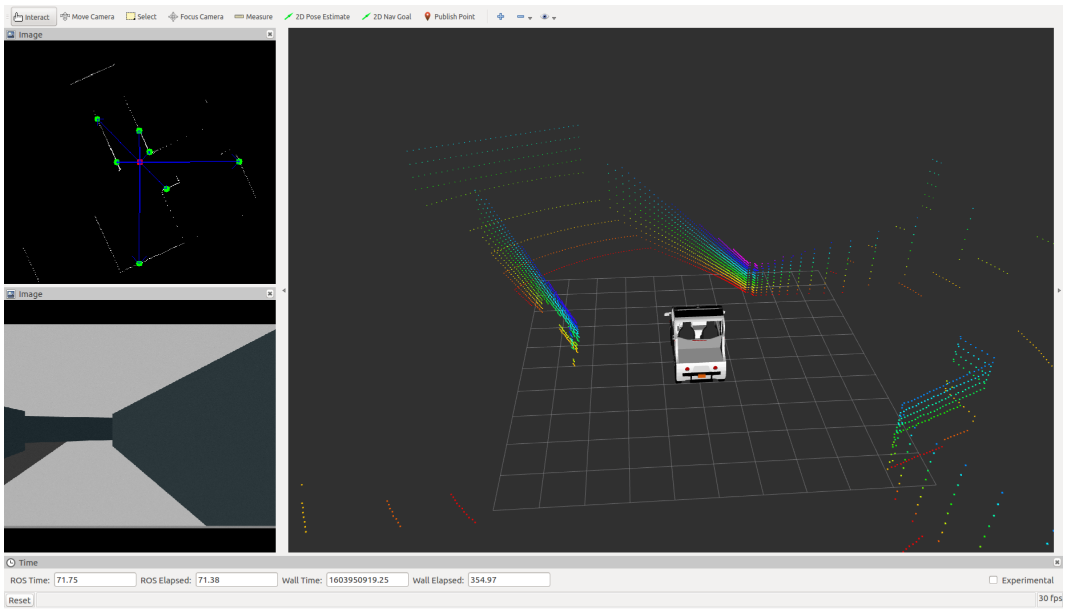
$$\begin{aligned} \bar{x} &= x + 85 \\ \bar{y} &= y - 45 \end{aligned} \tag{4}$$

3.4 Running experiment

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

```
roslaunch mp3 gem_vehicle.launch
```

By running this command, you will be able to see the Gazebo window as shown in figure 4 and the rviz window as shown in figure 6. For the rviz window, the window on the right shows the visualization of the raw point cloud data from the Lidar on the simulated vehicle. The window at bottom left shows the image from the camera on the vehicle. The window at up left shows a visualization of the Lidar bird-eye view. The distance between vehicle and wall at each measurement direction and the sensed position of wall is annotated on the bird-eye view. The annotation will not be displayed if no wall is sensed within sensor limit at that measurement direction.



ROS Time: 71.75 ROS Elapsed: 71.38 Wall Time: 1603950919.25 Wall Elapsed: 354.97

Figure 6: The rviz window. Note that the Lidar processing code shown here have 8 measurement directions.

With Gazebo started, you should start running the vehicle in the environment using command

```
python3 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

```
python3 set_pos.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

```
python3 main.py --num_particles 1000 --sensor_limit 15 --measurement_noise True
```

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 15.
3. measurement_noise – If enabled, there is a 50% chance the LiDAR measurement will be absent at each time step. Then particle weights will be uniformly distributed. By default, this parameter is set to False.

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).

3.5 Report

For problem 4-9, (1) plot the error in position estimation (euclidean distance between actual position and the estimated position) and orientation estimation as a function of algorithm iterations. Since particle filtering is a randomized algorithm, run several instances of the same setup and plot the average error for each time. (2) Provide a link to the video and include it in the report.

Problem 4 (30 points)**.** Using 4 measurement directions, keep the sensor limit constant at 15, run your algorithm with number of particles 500, 1000, 1500. How does changing the number of particles influence the estimation accuracy, converging speed and computational cost of the algorithm? Record a video of one of the three runs. The video should include the turtle map window.

Hint: Algorithm runtime may vary with number of particles. Modify the vehicle control frequency in vehicle.py to align with particle filter updates for convergence:

```
    rate = rospy.Rate(10)  # 10 Hz
```

Recommended frequencies are 10 Hz for 500 particles, 8 Hz for 1000 particles, and 4 Hz for 1500 particles, adjustable according to computing power.

Problem 5 (30 points)**.** Using 4 measurement directions, keep the number of particles constant at 1000, run your algorithm with sensor limit 15, 20, 25. How does changing sensor limit influence the estimation accuracy, converging speed and computation cost of the algorithm? Record a video of one of the three runs. The video should include the turtle map window.

Problem 6 (10 points)**.** Does the particle filter you implemented performs evenly well through the whole environment after converging? More specifically, does the your particle filter have larger prediction error in some regions of the environment then other regions? If yes, can you explain why this is happening?

Problem 7 (20 points)**.** Modify the LidarProcessing module and the sensor model so that they can make measurements in 8 directions. Run your algorithm with number of particles 1000 and sensor limit 20. How does having more sensor data influence the estimation accuracy and converging speed of the algorithm? Record a video of the run. Besides the turtle map window, the video should include the RViz window of the sensor measurements.

Problem 8 (5 bonus points)**.** Activate the measurement noise as outlined in Section 3.4. This will result in occasional absence of the Lidar measurements. Run your algorithm with particles 1500 and sensor limit 25. How does the missing lidar measurement influence the estimation accuracy and converging speed? Record a video of the run. The video should include the turtle map window.

Problem 9 (5 bonus points)**.** Modify the distribution of your initial particles to be within the top-right quadrant. Run your algorithm with particles 500 and sensor limit 15. How does the initial distribution of the particles influence the estimation accuracy and converging speed? Record a video of the run. The video should include the turtle map window.

Problem 10 (10 points)**. Demo** For this MP, you will need to demo your code to the TAs in lab sessions on **October 27**. The TA will primarily check if the particle filter can converge within a reasonable number of iterations and track the position of the vehicle closely. Note that you will also need to show that your filter can expand the sensor measurement from 4 directions to 8 directions. Also, prepare to be asked related questions.

4 Report and Submission

Problems 1-3 must be done individually (Homework 3). Write solutions to each problem in a file named `hw3_<netid>.pdf` and upload the document in Canvas. Include your name and netid in the pdf. You may discuss solutions with others, but not use written notes from those discussions to write your answers. If you use/read any material outside of those provided for this class to help grapple with the problem, you should cite them explicitly.

Problems 4-9 can be done in groups of 2-4. Each group should write a report that contains the solutions, plots, and discussions. This report should be submitted to Canvas **per group** with filename `mp3_groupname.pdf`. Please include links to the video and your solution code in the report.