Project 2: Where Am I

Date: December 25th, 2018

Menghong Feng

1 Abstract

Localization and navigation play an eseential role in robotic system. In this project we use ROS packages to create a mobile robot inside a provided map in Gazebo and RViz simulation. Adaptive Monte Carlo Localization (AMCL) and Navigation Stack algorithms are implemented as ROS packages to navigate mobile robot to its goal position.

2 Introduction

In the rest of report we will go through a complete process of simulating robot's localization and navigation tasks. In Background section we wil introduce three popular localization algorithms and compare their features. Model configuration section discusses how to design a robot in Unified Robot Description Format (URDF) file as well as its essential gazebo plugin packages. Results section shows robot's proformance in its localization and navigation task. Dicussion section states the failure and success made in this project. Future work section will provide suggestion for future research to enhance robot's performance.

3 Background

There are two popular algorithms used to localize robot's position: Kalman Filter and Monte Carlo Localization. These two algorithms do well to determine robot's position and orientation with respect to its global environment by using noisy data collected from the sensors. In the next two subsection we will discuss these two algorithms in details and compare their advantages and disadvantages.

3.1 Kalman Filter

Kalman filter is an estimation algorithm that is very popular in control system such as self-driving car, mobile robots, and so on. It can take data with a lot of uncertainty or noise in the measurement and return a very accurate estimate of real value. Kalman filter acts as a Gaussian distribution that is defined by its mean μ and variance σ^2. Mean value μ represents an average state (position and velocity) of a robot in environment, and variance σ^2 represents the uncertainty among robot's sense and move update. The Gaussian distribution and its value can be expressed and calculated as below:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

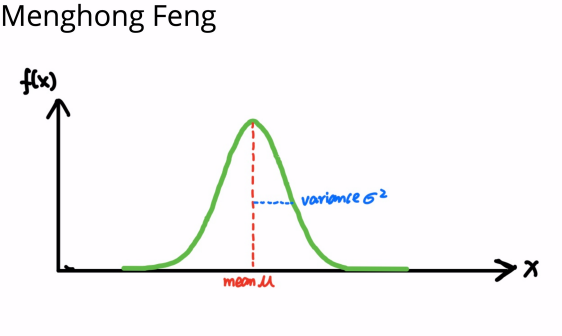

Figure 1: 1D Gaussian

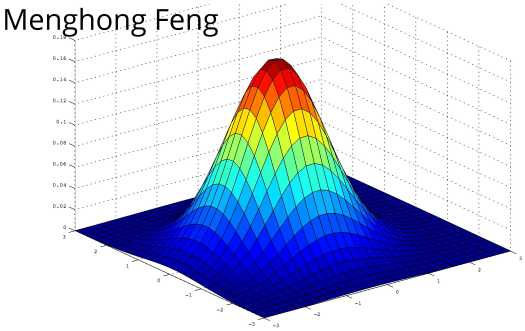

Figure 2: 2D Gaussian

In order to accurately localize a robot's position, Kalman filter forms a cycle to measure and update its state. The main steps in this cycle are called measurement update and state prediction. Measurement update takes noisy measurement to update current state (velocity and position). State prediction uses the updated state to predict what next state will be. Kalman filter repeats this measurement and predication steps to localize the car as it's moving. The examples below show mathematical process for 1D and 2D Kalman filter, respectively.

In 1D Kalman filter, we first start with an initial guess or use previous state to represent robot's location depicted as a red Gaussian distribution (μ_1, σ_1^2) below. Measurement update then gathers more information about the robot's surroundings depicted as green Gaussian distribution (v_1, r_1^2) and refine our location prediction as blue Gaussian distribution (μ_2, σ_2^2). In this step the measured location has a smaller variance r_1^2 due to its increased certainty compared to initial guess or previous state. The refined state is calculated based previous state (red curve) and measured state (blue curve) and formed as a new Gaussian distribution (blue curve) with even higher certainty σ_2^2. This process can be mathematically expressed as:

$$\mu_2 = \frac{r_1^2\mu_1 + \sigma_1^2 v_1}{r_1^2 + \sigma_1^2}$$

$$\sigma_2^2 = \frac{1}{\frac{1}{r_1^2} + \frac{1}{\sigma_1^2}}$$

With the best estimation of current state (blue curve), the state prediction takes action uncertainty (purple curve) into consideration to predict next state depicted as orange curve. For example, let's say we move to the right side at a certain distance and motion itself has its own uncertainty depicted as purple curve. The new state location has new mean value and an increased uncertainty depicted as orange curve. This new state is then served as the prior distribution passed to next cycle iteration for new measurement update. This process can be mathematicallg expressed as:

$$\mu_3 = \mu_2 + v_2$$

$$\sigma_3^2 = \sigma_2^2 + r_2^2$$

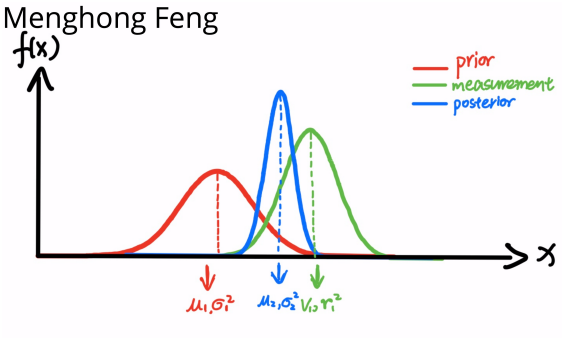

Figure 3: Step 1: Measurement Update

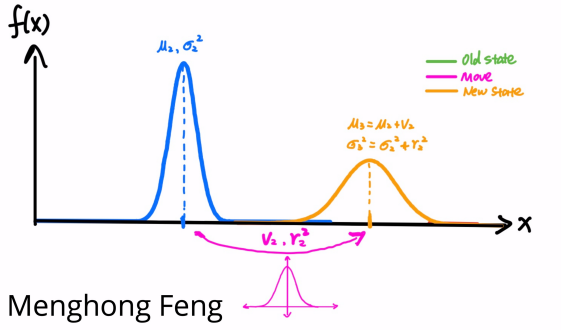

Figure 4: Step 2: State Prediction

For 2D Kalman filter, a 2D Gaussian distribution is defined over the space and it is represented as contour lines. The mean of 2D Gaussian is a vector μ located at the centroid of contour lines. Similarly, while a 1D Gaussian has a single value for the variance, a 2D Gaussian has a covariance 2 $\times$ 2 matrix represented by an uppercase Sigma Σ. An N dimensional Gaussian will have a covariance matrix with a size N $\times$ N. The covariance Σ defines the spread of Gaussian as indicated by the contour lines.

$$\mu = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & \sigma_y^2 \end{bmatrix}$$

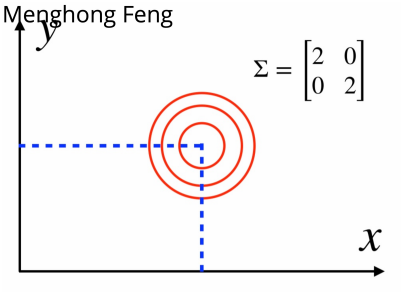

Figure 5: Equal Variance

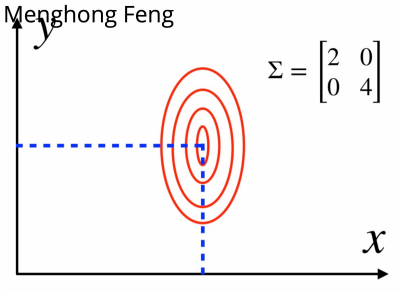

Figure 6: Unequal Variance

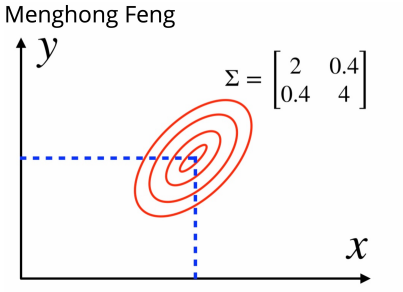

Figure 7: Correlation

In the covariance matrix Σ, diagonal terms (σ_x^2, σ_y^2) represent the variance and off-diagonal values ($\sigma_x\sigma_y$, $\sigma_y\sigma_x$) represent the correlation terms, which are always symmetrical and equivalent($\sigma_x\sigma_y$=$\sigma_y\sigma_x$). When x and y variances are both small and equal to one another, the 2D Gaussian representation will be circular as shown in figure 5. In such a case, we are equally certain about the location of robot in each dimension. When x and y variances are not equal to one another, meaning we are certain about the location of one dimension but uncertain for another dimension, such case of scenario will result in an oval shape for 2D

Gaussian representation as shown in figure 6. Both cases mentioned above have zero correlation terms. When correlation terms are not zero and variances in each dimension are not equal, the oval shape of 2D Gaussian representation tilt to certain angle. In such correlation case, once we get information about one dimension, it will reveal the information of another dimension due to correlation.

Correlation is an essential concept in multidimensional Kalman filter. The state has to be obversable in 1D Kalman filter. In multidimensional Kalman filter, however, hidden state variables that cannot be directly measured are allowed to exist. The value of hidden state can be inferred from observable state due to their correlation. In order to explain how does 2D Kalman filter work, we will go through an example here. Assume we'd like to track robot's observable position x and hidden velocity $\dot{x}$. Robot's position x and velocity $\dot{x}$ are linked through a formula expressed as:

$$x' = x + \dot{x}\Delta t$$

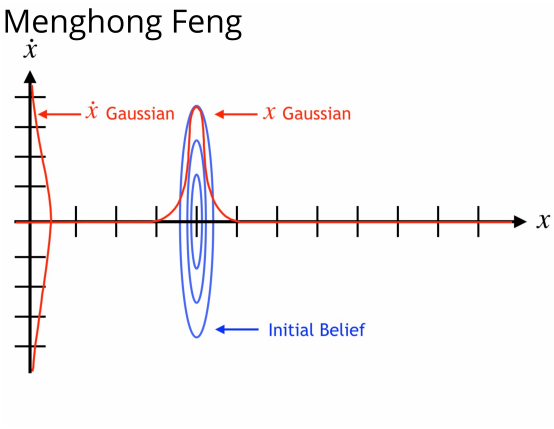

Figure 8: Step 1: Initial Belief

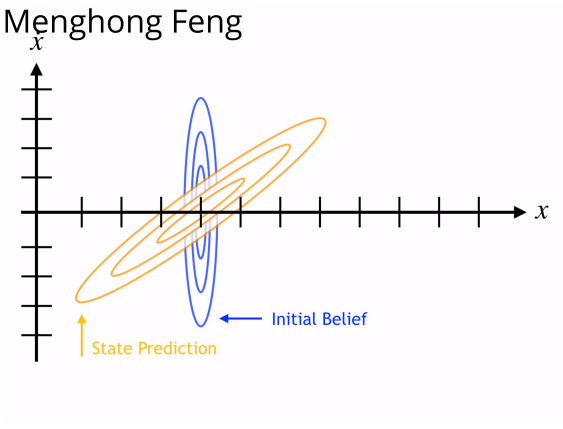

Figure 9: Step 2: State Prediction

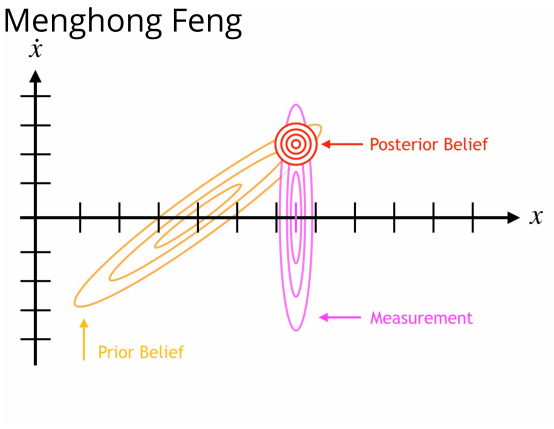

Figure 10: Step 3: Measurement Update

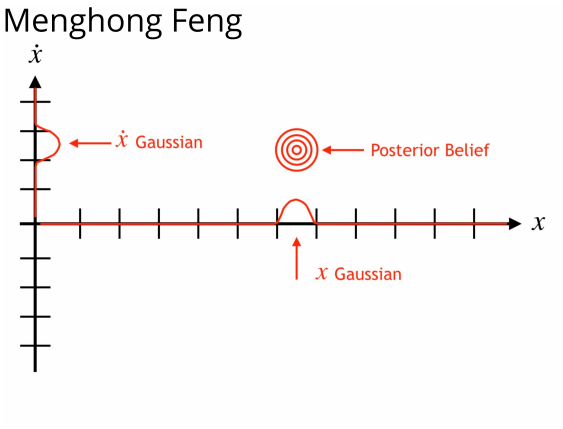

Figure 11: Step 4: Posterior Belief

As shown in figure 8, we first have an initial guess about robot's location but have no idea for its velocity. In such case robot's velocity has much large uncertainty than its position, showing a wide range of $\dot{x}$ Gaussian representation compared to the narrow x Gaussian distribution. In step 2 of state prediction, a new Gaussian distribution can be derived from the relationship between the hidden variable $\dot{x}$ and observable variable x. The new Gaussian represents the correlation between velocity $\dot{x}$ and location x. In step 3 of measurement update, the state prediction in previous step becomes a prior belief and a new measurement is updated to show robot's current location. When the measurement update is applied to prior belief, a very small posterior belief can be formed in the region where prior belief and measurement update superposed together. As shown in step 4, this posterior belief has very narrow Gaussian distribution for both robot's location and velocity, meaning the value of hidden variable $\dot{x}$ can be revealed by applying Kalman filter even though it is not directly measured.

State Predication:

$$x' = Fx$$

$$P' = FPF^T + Q$$

Measurement Update:

$$y = z - Hx'$$

$$S = HP'H^T + R$$

Calculation of Kalman Gain:

$$K = P'H^T S^{-1}$$

Calculation of Posterior State and Covariance:

$$x = x' + Ky$$

$$P = (I - KH)P'$$

In order to code Kalman filter in C++, we have to first go through the mathematical expression. As shown above, in state predication step posterior state x' is obtained through the multiplication of state transition function F and the prior state x expressed as $x' = Fx$. In reality, the equation should also account for process noise, but process noise is a Gaussian with a mean of 0 so the update equation for the mean need not include it. On the other hand, when the state x is predicated by transition function F, its covariance P will be affected by the square of F plus the uncertainty Q due to process noise. In matrix form, it can be expressed as $P' = FPF^T + Q$.

In measurement update step, the measurement residual y is calculated by subtracting the expected measurement based Hx' from actual measurement z, expressed as $y = z - Hx'$. H is denoted as measurement function. Similarly, the equation $S = HP'H^T + R$ maps the state prediction covariance P' into the measurement space and adds the measurement noise R. S is denoted as updated covariance during measurement action.

The Kalman Gain K determines how much weight should be placed on the state prediction x, and how much on the measurement update y, which is expressed as $x = x' + Ky$. For example, if K is large then the weight of measurement update y is large. If K is small then the weight of state prediction x increases. Finally, the last step in the Kalman Filter is to update the new state's covariance P using the Kalman Gain K, which is expressed as $P = (I - KH)P'$. A Kalman filter loop is continued in real time to estimate accurate state from inaccurate initial estimates.

3.2 Extended Kalman Filter (EKF)

There are two assumptions must be met in order to apply Kalman filter to solve robot's localization problem:

- Motion and measurement models are linear
- State space can be represented by a unimodal Gaussian distribution

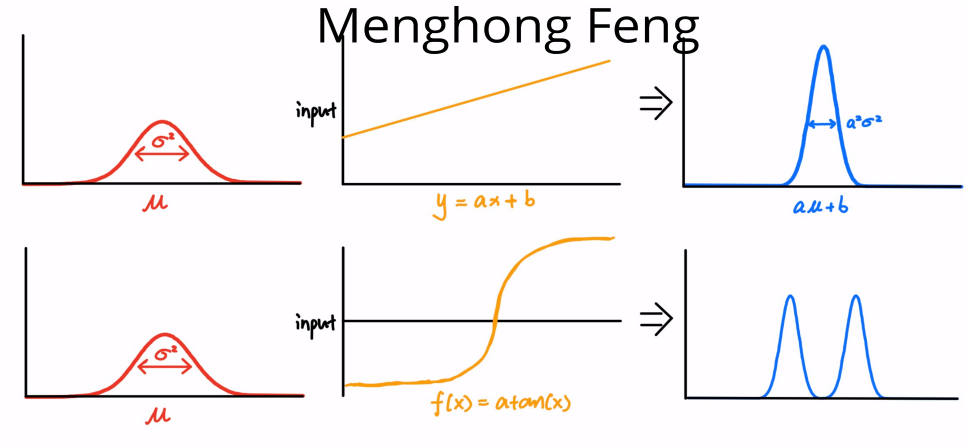

Figure 12: Non-Gaussian Distribution

However, robot motion in real world can be nonlinear, as shown in figure 12, and thus cannot be solved by Kalman filter. In order to solve nonlinear localization problem, the extended Kalman filter (EKF) is proposed to replace normal Kalman filter. EKF implements Talyor series to approximate a nonlinear function in small interval by a linear function. The linear estimation is only good for a small section of the function. The mean μ can be updated with a nonlinear function,

$$\mu \xrightarrow{\text{f(x)}} \mu'$$

but variance cannot be updated by a nonlinear function since it will result in a non-Gaussian distribution which is much more computationally expensive to work with. In order to update variance, the Extended Kalman Filter linearizes the nonlinear function $f(x)$ over a small section and calls it F. This linearization, F, is then used to update the state's variance:

$$P \xrightarrow{\text{F}} P'$$

The linear approximation can be obtained by using the first two terms of the Taylor Series of the function centered around the mean:

$$F = f(\mu) + \frac{\delta f(\mu)}{\delta x}(x - \mu)$$

To linearize functions with multiple dimensions, multi-dimensional Taylor series with first two terms is introduced here:

$$T(x) = f(a) + (x - a)^T Df(a)$$

where $Df(a)$ is the Jacobian matrix which holds the partial derivative terms for the multi-dimensional equation.

$$Df(a) = \frac{\delta f(a)}{\delta x}$$

The mathematical expression for Extended Kalman Filter are listed blow. The equation on the left side of arrow is of Kalman Filter, while the equation on the right side of arrow is of EKF. The transition function F and measurement function H in EKF is Jacobian matrix.

State Predication:

$$x' = Fx \rightarrow x' = f(x)$$

$$P' = FPF^T + Q$$

Measurement Update:

$$y = z - Hx' \rightarrow y = z - h(x')$$

$$S = HP'H^T + R$$

Calculation of Kalman Gain:

$$K = P'H^T S^{-1}$$

Calculation of Posterior State and Covariance:

$$x = x' + Ky$$

$$P = (I - KH)P'$$

3.3 Monte Carlo Localization

Monte Carlo Localization (MCL) is a very popular localization algorithm in robotics. MCL uses measurement data collected from range sensors to keep track of robot pose and implement particles to localize robot. Therefore, MCL can also be referred as Particle Filter Algorithm. In an example below we will show the process of how MCL work.

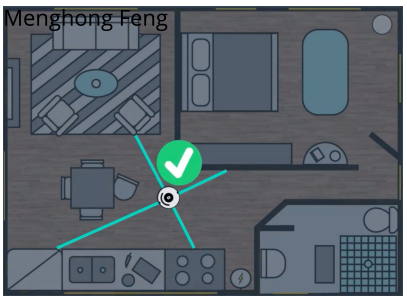

Figure 13: Sense Surrounding

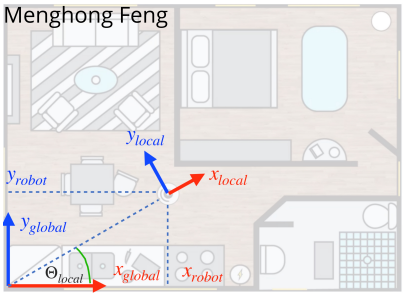

Figure 14: G & L Coordinates

Figure 15: Virtual Particles

Figure 16: Assign Weight

Figure 17: Resampling

Figure 18: Localization Converge

The robot has on-board range sensors to sense nearby obstacles such as walls and objects. In the example shown above, the current robot pose is represented by x_{local} & y_{local} coordinates and orientation Θ_{local} all with respect to the global coordinate frame. With the MCL algorithm particles are initially spread randomly and uniformly throughout entire map. The particles mentioned here are virtual elements to resemble a robot pose in the map. Just like the robot, each particle has $x_{particle}$ & $y_{particle}$ coordinates and orientation $\Theta_{particle}$. These particles represent the hypothesis of where the robot might be. In addition to three-dimensional vector, each particle is assigned a weight. The weight of a particle is the difference between the

robot's actual pose and the particle's randomly guessed pose. The particle. The smaller the difference the larger the weight. A particle with large weight is more likely to survive from resampling process. Finally, after several iteration of MCL, particles will converge and estimate robot's pose.

3.4 Comparison Between MCL and EKF

EKF and MCL are both popular localization algorithms that have been widely implemented in robot. However, each of them also has its own advantage and disadvantage in certain circumstance. The table below compares their features with one another.

Menghong Feng	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✔	✔✔
Efficiency(time)	✔	✔✔
Ease of Implementation	✔✔	✔
Resolution	✔	✔✔
Robustness	✔✔	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodal Continuous

Figure 19: Comparison Between MCL and EKF

As shown in the table, the main difference between MCL and EKF is that MCL can represent non-Gaussian distribution while EKF is restricted with a linear Gaussian state-based assumption. Therefore, MCL can solve a much greater variety of practical problems since in real world not all models can be represented as linear Gaussian distribution. The second main advantage for MCL over EKF is that MCL allows user to control the computational memory and resolution of solution by changing the number of particles distributed randomly and uniformly throughout the map. However, though MCL is easier to implement compared to EKF, EKF is less expensive in computation and memory storage in exchange of its robustness.

4 Model Configuration

Building robots in simulation is a very valuable skill for a roboticist or a robotics software engineer. In this subsection we will go through the details of mobile robot design in Unified Robot Description Format (URDF) file and its corresponding Gazebo plugin file. The urdf code and Gazebo plugin file can be found in *here*.

The mobile robot is deisgned to has 6 degrees of freedom (DOF) expressed by the pose (x, y, z, Roll, Pitch, Yaw) and is controlled by a pair of rolling wheels. It is equipped with a camera and a laser rangefinder (Hokuyo) to help robot sense the environment. Gazebo plugins are created to enable robot's moving and sensing functionalities defined in urdf file. Since we have a two-wheeled mobile robot, we will use a plugin that implements a Differential Drive Controller. This plugin takes information such as wheel seperation, joint names and so on to calculate and publish the robot's odometry information to the defined topics. In this task, we send velocity commands to robot so that it can move to a specific direction. For the camera and Hokuyo senser we use preexisting plugins and define topics where they will publish information or data.

In this project, three URDF models are built to meet rubric requirement. As shown below, the first robot is the Udacity bot provided in the lesson. The second and third robots are made by author. Though an attempt has been made to create a four-wheel robot, the Differential Drive Controller plugin seems to work better for two-wheel robot. Therefore, only the first and thrid robot are launched in Gazebo simulation environment. The URDF files for all three robots can be found in here.

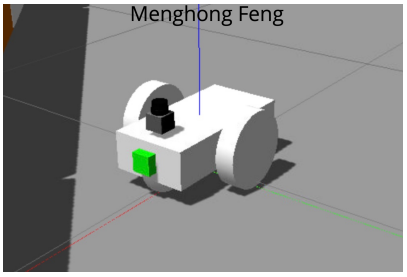

Figure 20: Udacity Robot

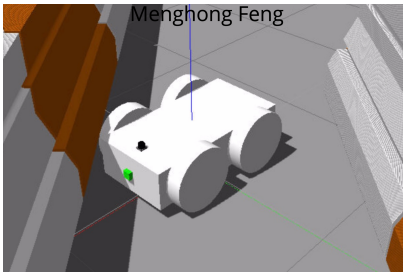

Figure 21: Four-Wheel Robot

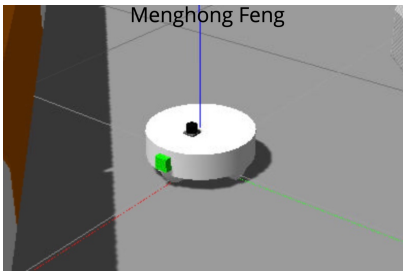

Figure 22: Roller Robot

In this project we use a map created by Clearpath Robotics. The map files can be found in *here*, and the world file can be found in *here*. The mobile robot model and map are shown below.

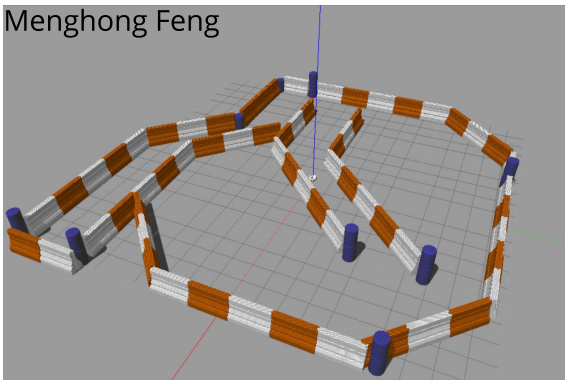

Figure 23: Map in Gazebo

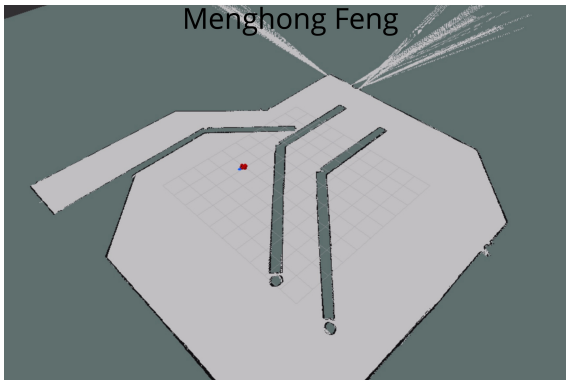

Figure 24: Map in Rviz

5 Results

Before we present the simulation results for Udacity robot and roller robot created by author, let's first discuss how to justify if the localization algorithm has been well-tuned or not. When robot is simulated in Gazebo, we can visualize its position uncertainty by adding PoseArray elements in RViz. The PoseArray in Rviz depicts a certain number of particles, represented as arrows, around the robot. The position and the direction the arrows pointing in represent an uncertainty in the robot's pose. If the algorithm is well-tuned and robot is certain for its location, the number of arrows will reduce and arrows' distribution will be narrowed down in a small region. For example, figure 25 and 26 represent robot's initial and posterior position, respectively. Since the initial guess for robot's position is random and highly uncertain, the number of arrow is large and arrows' distribution is dispersive. After robot begins to move and takes measurement to its surrounding, the algorithm is able to accurately localize robot's position and thus the number of arrow is reduced.

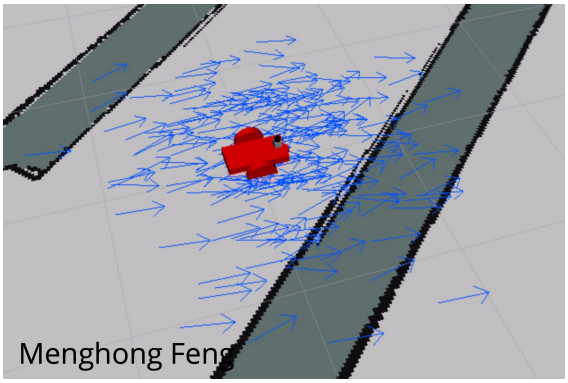

Figure 25: Initial Position

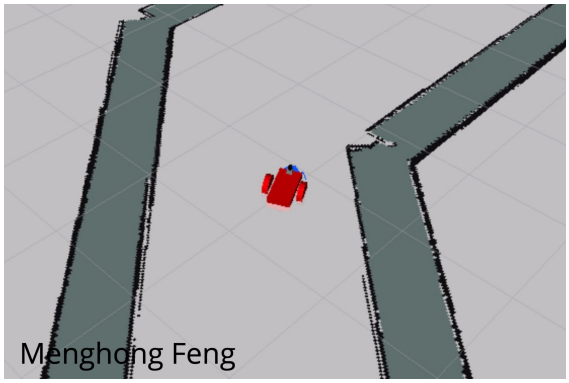

Figure 26: Posterior Position

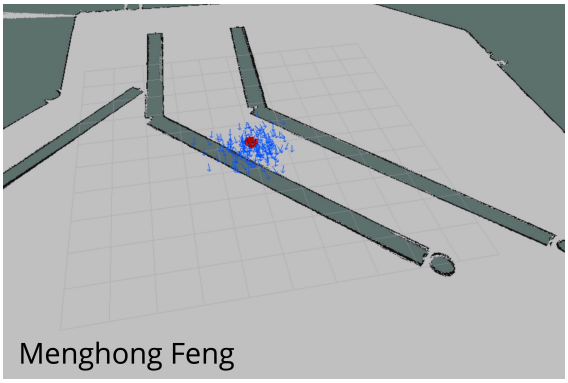

Figure 27: Udacity's Robot Initial Position

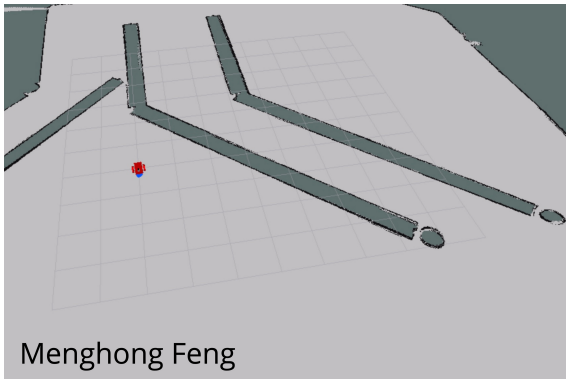

Figure 28: Udacity's Robot Final Position

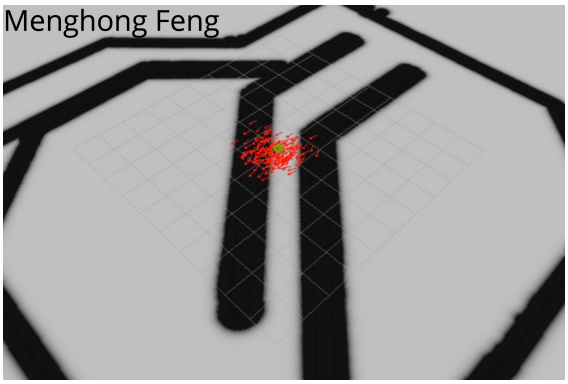

Figure 29: My Robot Initial Position

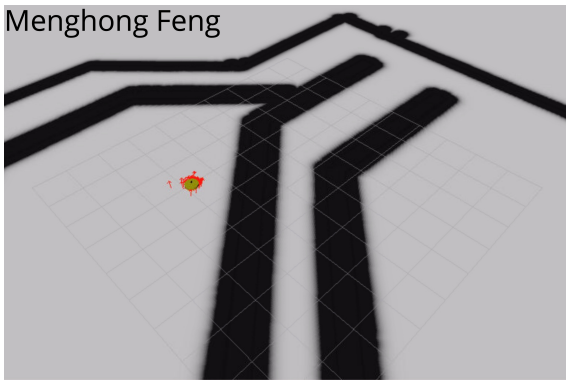

Figure 30: My Robot Final Position

In this project, both Udacity robot and roller robot created by auhtor are successful to navigate themselves to destination as shown above. As shown in figures below, the number of PoseArray has been reduced considerably from robot's initial position to its final destination, demonstrating that algorithm has been well-tuned in both cases. The parameters for both robots are shown in the table below.

AMCL Launch Package		
	Udacity Robot	My Robot
min particles	10	10
max particles	200	200
initial pose x, y, z	0,0,0	0,0,0
odom model type	diff-corrected	diff-corrected
odom alpha 1,2,3,4	0.010	0.010

In AMCL launch package, the maximum and minimum number of particles define how accurate and how computational intense the model will be. The higher the number of particles the more accurate the localization is. However, adding more particles will also require more computational source. Observation has been made that intense calculation also increase the latency between each itertation. In this study we use the range of particles defined in provided Udacity robot since it works fine in previous and my robot model. The rest of parameters define the location and orientaion of robot, which are not changed either.

Costmap Common Parameters		
	Udacity Robot	My Robot
obstacle range	2.5	2.0
raytrace range	3.0	5.0
transform tolerance	0.3	0.2
robot radius	0.25	0.4
inflation radius	0.5	0.3

In Costmap Common Parameters table, we decrease the value of obstacle range and inflation radius. Inflation radius defines how thick the map wall is and obstacle range determines how close the robot can reach to the wall before obstacle is detected. The reduction in both parameters give robot more space to navigate. Also the observation has been made that increasing these values will narrow down the movement space for robot, while decreasing these values too much will let robot come too close to the wall before robot can detect it, which both case letting kidnapping problem to occur more easily. Transform tolerance is tuned to balance model accuracy and latency, and raytrace range is used to clear and update the free space in costmap as robot moves.

Base Local Planner		
	Udacity Robot	My Robot
holonomic robot	false	false
yaw goal tolerance	0.05	0.05
xy goal tolerance	0.1	0.1
sim time	1.0	1.0
meter scoring	true	true
pdist scale	0.5	0.5
gdist scale	1.0	1.0
max vel x	0.5	0.5
max vel y	0.1	0.1
max vel theta	2.0	2.0
acc lim theta	5.0	5.0
acc lim x	2.0	2.0
acc lim y	5.0	5.0
controller frequency	15.0	20.0

In Base Local Planner table, most parameters value are kept same as in Udacity robot case, except we increase the controller frequency to allow robot respond quickier but also add a bit of latency.

6 Discussion

My robot is able to navigate itself to target location. However, the kidnapped situation occurs more frequently for my robot compared to Udacity robot. Escaping from kidnapped situation sometimes depends on where the robot is kidnapped and how close it is near the wall, which can be solved by tuning the obstacle range and inflation radius. My robot's overall performance is less than Udacity robot so that it will takes longer time to reach the target location. There are many factors can cause this result. One thing is adding more weight to your robot can actually decrease its movement. Also in my robot case the free space to allow robot's movement is increased, robot may takes longer time to explore the region before it can reach the goal.

AMCL can tackle the kidnapping problem by re-gathering to correct direction. However, the observation is found that AMCL may fail to help robot escape from kidnapping if robot is too close to the wall. Robot might need to rotate itself to adjust its pose for particle re-gathering and sometimes even go back a little bit to escape from kidnapping, which may be the case if wall thickness is decreased.

In industrial application one of the most famous example is Amazon robotics. Amazon uses robots in warehouse to move and send packages. Localization algorithms must be widely used in Amazon robot. Also iRobot vacuum cleaner robot also use localization algorithm to localize and navigate itself.

7 Future Work

My robot has much space to improve. First of all, the occur freqency of kidnapping issue must be reduced. By adding more particles and tuning the parameters may solve this problem, but also increase the latency and computational cost. Another improvement can be made is we can add additional sensors to collect the data with less noise, which can be used to improve robot's localization ability and also might increase the latency since more data need to process this time. Ideally if we have a more powerful labtop we may increase the accuracy without worry too much about accuracy and time trade-off. NVIDIA Jetson Tx2 is a power embedded system that can be used to run the operation in the future.