RoboND – Localization *'Where am I?'* Project

Abstract

Robot localization techniques are used to find the current location and orientation of a mobile robot in some environment. In this paper, the topic of localization is discussed and applied in simulation using ROS, Gazebo, and RViz to successfully navigate two modeled robots with a camera and a laser sensor through a given map to reach a goal position and orientation. The paper begins with explaining the two most common localization techniques, the Kalman filter and the particle filter, then the results of using the Adaptive Monte Carlo Localization, or AMCL algorithm which is based on Particle filtering are discussed on both robots. Also, the robots' configurations are discussed, and the parameters used to improve localization for both robots are explained.

Introduction

Robot localization is the process of determining where a mobile robot is located with respect to its environment. Localization is one of the most fundamental competencies required by an autonomous robot as the knowledge of the robot's own location is an essential precursor to making decisions about future actions. In a typical robot localization scenario, a map of the environment is available, and the robot is equipped with sensors that observe the environment as well as monitor its own motion. The localization problem then becomes one of estimating the robot position and orientation within the map using information gathered from these sensors. Robot localization techniques need to be able to deal with noisy observations and generate not only an estimate of the robot location but also a measure of the uncertainty of the location estimate.

In this project the Adaptive Monte Carlo Localization method is used in ROS to estimate the robot's pose while it navigates its way to reach a goal position in a given map in Gazebo and RViz simulation.

Background

There are multiple localization methods such as: Markov Localization, Grid localization, Kalman Filter localization, and Monte Carlo Localization. The concept of localization for mobile robots can be demonstrated through the below diagram.

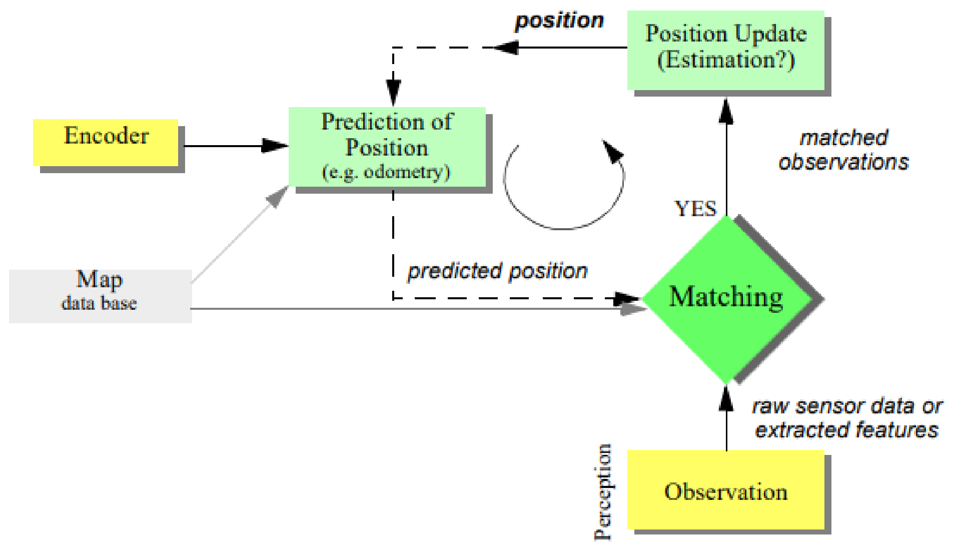


Figure 1. General schematic for mobile robot localization.

For this project, only the Kalman filter localization and the Monte Carlo Localization (Particle Filtering) will be discussed.

Kalman Filter

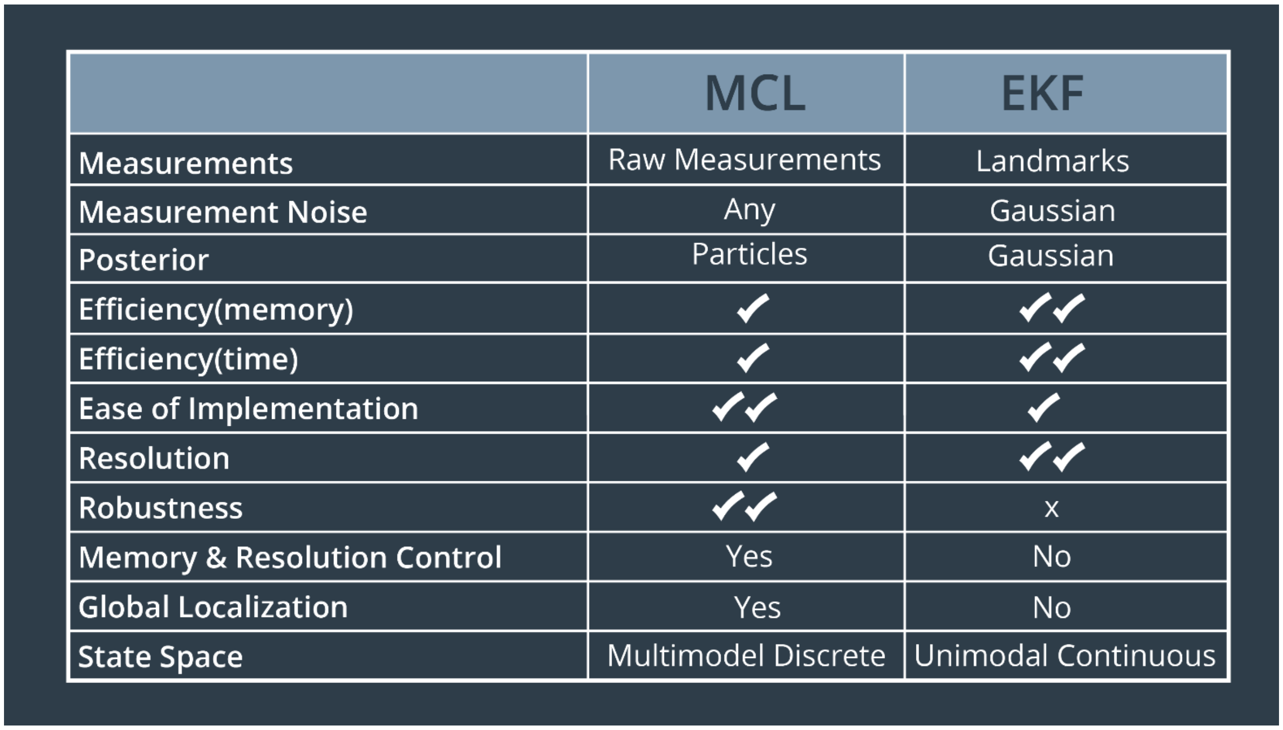
The Kalman filter-based techniques are based on the assumption that uncertainty in the robot's position can be represented by a unimodal Gaussian distribution. Kalman filter-based techniques have proven to be robust and accurate for keeping track of the robot's position. By allowing robots to perform sensor fusion, which is pulling and combining data from multiple sensors to obtain an accurate estimate of the measured value. However, this approach cannot deal with multi-modal densities typical in global localization. The other limitation is that the initial posture must be known with Gaussian uncertainty at most. Therefore, the Kalman filter cannot be applied to most robotics problems. However, by implementing an Extended Kalman Filter (EKF), one can linearize non-linear problems, in order to convert the result into a Gaussian distribution.

Particle Filter

The Monte Carlo Localization (MCL) also known as the Particle filter algorithm is the most popular localization algorithm in robotics. This algorithm uses particles to localize the robot. Each particle has a position and orientation, representing the guess of where the robot is positioned. These particles are re-sampled every time the robot moves by sensing the environment through range-finder sensors, such as lidars, sonars, RGB-D cameras, among others. After a few iterations, these re-sampled particles eventually converge with the robots pose, allowing the robot to know its location and orientation. The MCL algorithm can be used for both Local and Global Localization problems and is not limited to linear models.

Comparison

Although you can estimate the pose of almost any robot with accurate sensors with the Extended Kalman Filter (EKF) algorithm, the Monte Carlo Localization (MCL) algorithm has many advantages over the EKF algorithm. The MCL algorithm is easier to program and setup than the EKF algorithm, making it better for robot implementation, debugging, and community support. The EKF algorithm is normally restricted by a Linear Gaussian state space assumption; whereas MCL can be used to represent any model. This makes the MCL algorithm helpful since the world cannot always be modeled by Gaussian distributions. Furthermore, with the MCL algorithm you can control the computation memory and resolution of the solution by tuning the particle quantity distributed randomly around the map (See Fig. 2).



	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 2. EKF VS MCL.

Results

Two mobile robots were built in simulation, a camera and a laser sensor were added to them, and the robots were integrated with Gazabo and RViz by developing a ROS package for the robots.

The visuals of both robots are shown in the below figure. A) is the robot model of the “udacity bot” mobile robot and B) is the robot model of the “my bot” mobile robot.

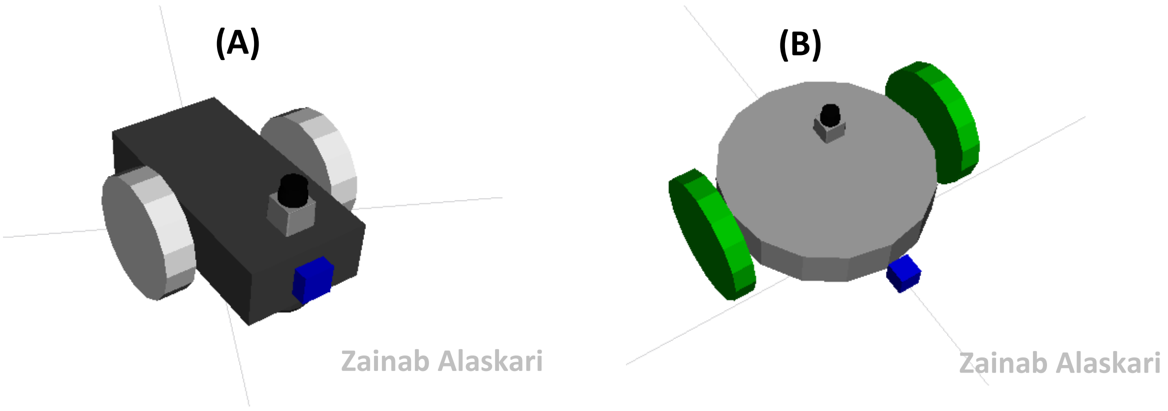


Figure 3. (A) Robot model for "udacity bot". (B) Robot model for "my bot".

The Adaptive Monte Carlo Localization (AMCL) package and the Navigation Stack package were integrated to allow for the robots to navigate and localize themselves in a given map.

After tuning the different parameters corresponding to these ROS packages, both the “Udacity bot” and “my bot” reached the goal position successfully.

The results show that both robots’ particles were uniformly spread out around the robots at the start of the simulation and then as the robots started to navigate their way to the end position, the particles had a tight grouping pointing towards the orientation of the goal.

The below figure shows both robots at their initial position.

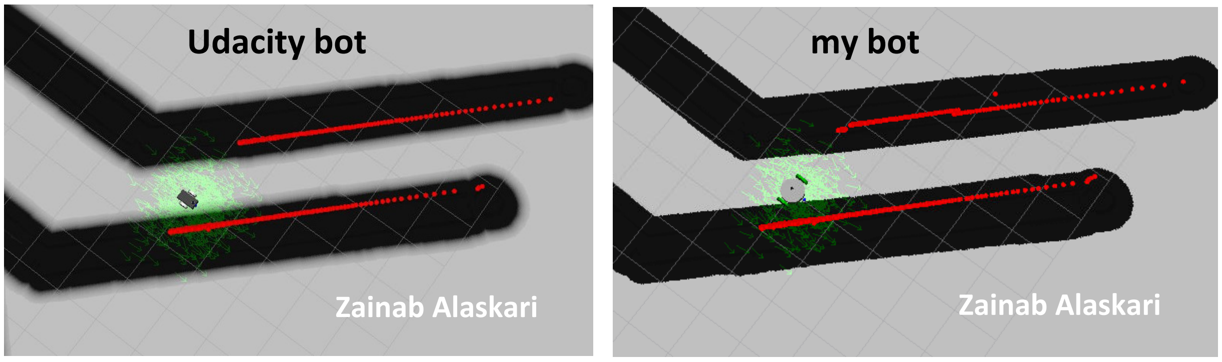


Figure 4. Robots at their initial position.

The below figure shows both robots as they navigate their way through the environment.

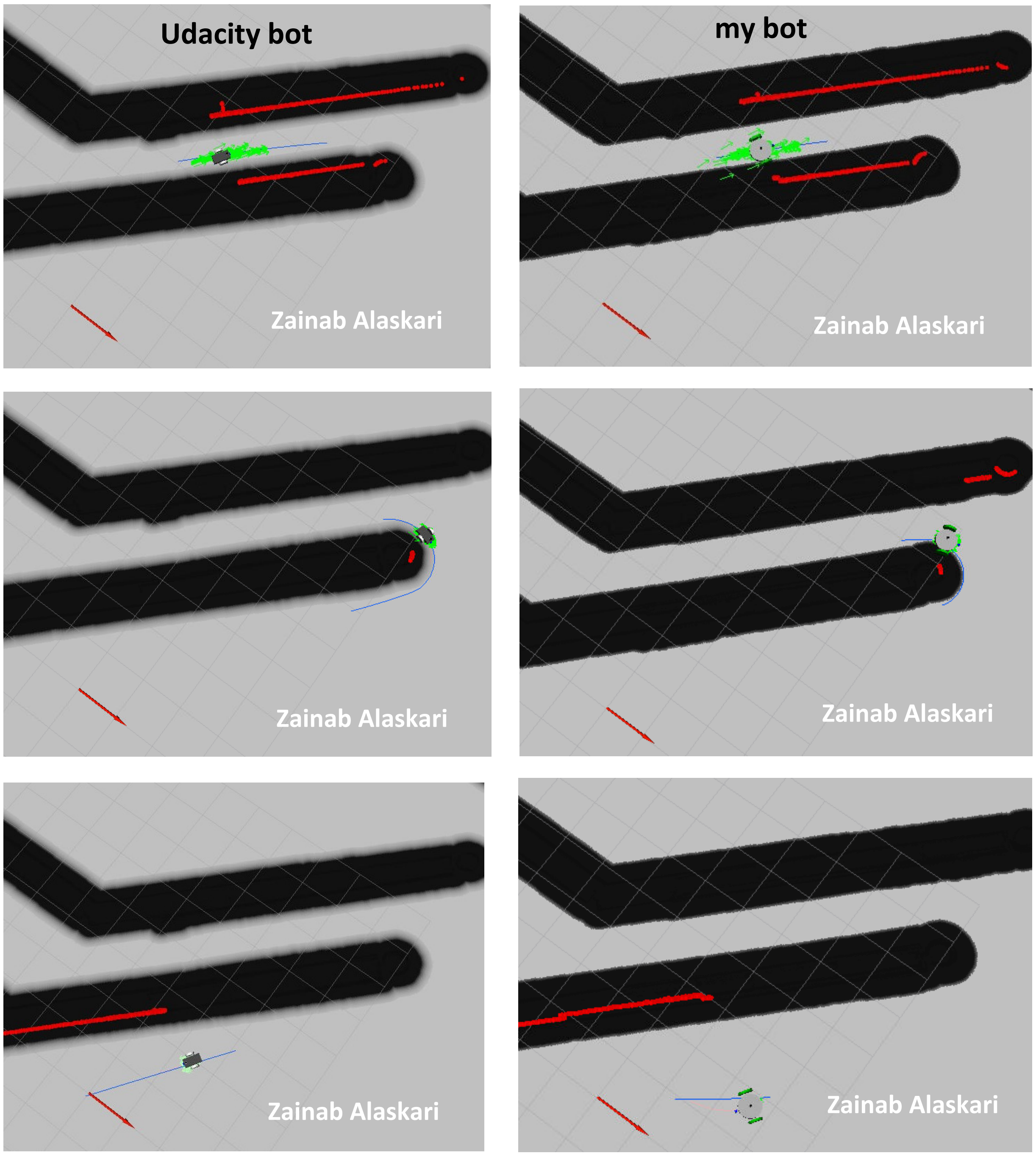


Figure 5. Both robots following their trajectory path to reach the goal position.

The below figure shows both robots at the goal position.

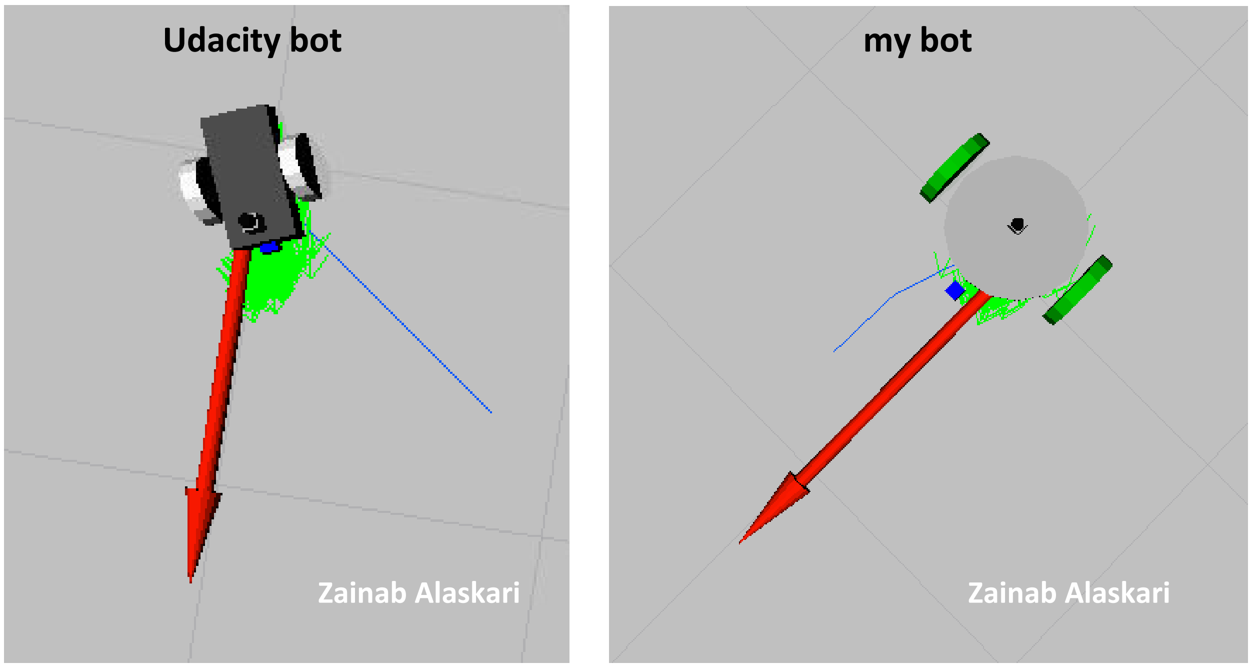


Figure 6. Robots at goal position.

As seen from the above figure, the “my bot” was oriented better than the “udacity bot”. The “udacity bot” reached the goal position within 2 minutes, while the “my bot” reached the goal position within 35 minutes!

It took longer for the “my bot” to reach the goal position because it moved much slower due to the control loop missing its desired rate of 20Hz. For some reason the navigation stack for “my bot” required more processing power and memory than the navigation stack for the “udacity bot”.

Model Configuration

Udacity bot

To enhance the performance of the “udacity bot”, the following parameters were tuned/ added:

In the amcl.launch file:

- The minimum number of particles was chosen to be 100 particles and the maximum was chosen to be 500 particles. The more particles the more accurate the localization performance will be but at the cost of additional compute resources. Since, the “udacity bot” performed very well at these values in terms of localization and in the robot’s simulated movement, where the robot moved smoothly and fast in the simulation. Therefore, this particle range was found to be suitable.

  ```
  <param name="min_particles" value="100"/>

  <param name="max_particles" value="500"/>
  ```

- The transform tolerance was set to 0.35. It is the time with which to post-date the transform that is published, to indicate that this transform is valid into the future.

  Both the amcl and move_base packages or nodes require that this information be up-to-date and that it has as little a delay as possible between the three maps transforms (the world map, the global costmap, and the local costmap). Once the tf was tuned at this value the three maps in RViz were able to be visualized without any issue.

  ```
  <param name="transform_tolerance" value="0.35"/>
  ```

- For tuning the laser parameters, the likelihood_field laser model type was chosen because it is usually more computationally efficient and reliable for the given environment. The laser max distance was set at 3.0 m because it was seen that range is enough for the robot to detect and avoid obstacles in the given environment.

  ```
  <param name="laser_model_type" value="likelihood_field"/>

  <param name="laser_likelihood_max_dist" value="3.0"/>
  ```

- The initial pose estimate was set to zero to coincide with the location of the robot at start-up.

  ```
  <param name="initial_pose_x" value="0.0"/>

  <param name="initial_pose_y" value="0.0"/>

  <param name="initial_pose_a" value="0.0"/>
  ```

- The next parameters modified, and the most important in terms of localization accuracy, were the odom_alpha parameters. These parameters specify the expected noise in odometry's rotation estimate. Odom_alphas 1-4 are the only parameters tuned since the robot uses a differential drive system.

  Odom_alphas 1-4 were set to 0.0008 as it was noticed that reducing these values resulted in a tighter packed set of particles as the robot moved towards the goal position.

  ```
  <param name="odom_alpha1" value="0.0008"/>

  <param name="odom_alpha2" value="0.0008"/>

  <param name="odom_alpha3" value="0.0008"/>

  <param name="odom_alpha4" value="0.0008"/>
  ```

In the base_local_planner_prams.yaml file:

The trajectory planner is responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan.

The yaw goal tolerance was set to 0.05, this is the acceptable tolerance allowed for the calculated orientation of the robot. This value should have been decreased further so that the robot can be better oriented at the goal position. The xy goal tolerance was set to 0.1 to reduce the positioning errors. The the global path (pdist_scale) compliance was increased to 3.5 to help the robot from not deviating too much from the global path.

```
yaw_goal_tolerance: 0.05

xy_goal_tolerance: 0.1

pdist_scale: 3.5
```

In the cost_map_common_params.yaml file:

The transform tolerance was set to 0.35. the robot’s radius was added and it was set at 0.25 m.

The obstacle range was set to 3.0 m, which means the robot can detect any obstacle within 3m from the base of the robot. This value was found to be suitable because, no falsely detecting obstacles were found and it did not add much computational costs.

raytrace_range was set to 3.0m. This parameter is used to clear and update the free space in the costmap as the robot moves.

The inflation radius was set to 0.5m, which conformably fits the robots size. This parameter defines a padding that is added to obstacles. The navigation planer then takes the padding into account when calculating the global path.

The robot’s radius was also added to make sure that the robot is not going to bump into the walls.

```
obstacle_range: 3.0

raytrace_range: 3.0

transform_tolerance: 0.35

robot_radius: 0.25

inflation_radius: 0.5
```

In the global_costmap_params.yaml file:

The update and publish frequencies were reduced to 10 Hz because the map was not getting updated fast enough. The update loop was taking longer than the desired frequency rate of 50 Hz which made the robot stop in the simulation.

The width and height of the global cost map were reduced drastically to 10 m to release computational resources. The resolution was increased from its default value of 0.05 to 0.08, which was a good enough value to reduce computational load while keeping the preferred accuracy.

```
  update_frequency: 10.0

  publish_frequency: 10.0

  width: 10.0

  height: 10.0

  resolution: 0.08
```

In the local_costmap_params.yaml file:

The local costmap size was reduced to 5m x 5m to successfully navigate around the corner at the end of the corridor. This is because the goal creates a huge influence over the local costmap. This resulted in the robot getting pulled away from the calculated global path. Reducing the size of the local costmap to approximate to the corridor width solved this problem.

```
  update_frequency: 10.0

  publish_frequency: 10.0

  width: 5.0

  height: 5.0

  resolution: 0.08
```

My bot

As for the model configuration of the "my bot", this robot's chassis base was chosen to be in cylindrical shape with a radius of 0.25 m. This robot's wheels had a slightly bigger radius than the "udacity bot" wheels. This robot total radius was 0.35 m which is bigger than the "udacity bot" radius. The shape and size of the robot was changed in such manner to allow for a comparison between the two robots in terms of their localization performance.

The location of the camera sensor was in front of the "my bot" in the same manner that it was placed in the "udacity bot". The laser scanner sensor was placed slightly higher on top of the "my bot" because it was noticed that when it was directly placed on top of the chassis of the robot, it would scan the wheels of the robot and would consider them as obstacles.

Testing the robot with same parameters

When this robot model was tested using the same AMCL and navigation parameters as the "udacity bot" it was noticed that the robot was not following the global path, and it was noticed that the robot was heading straight to the goal position wanting to climb over the obstacle as if it wasn't seen by it.

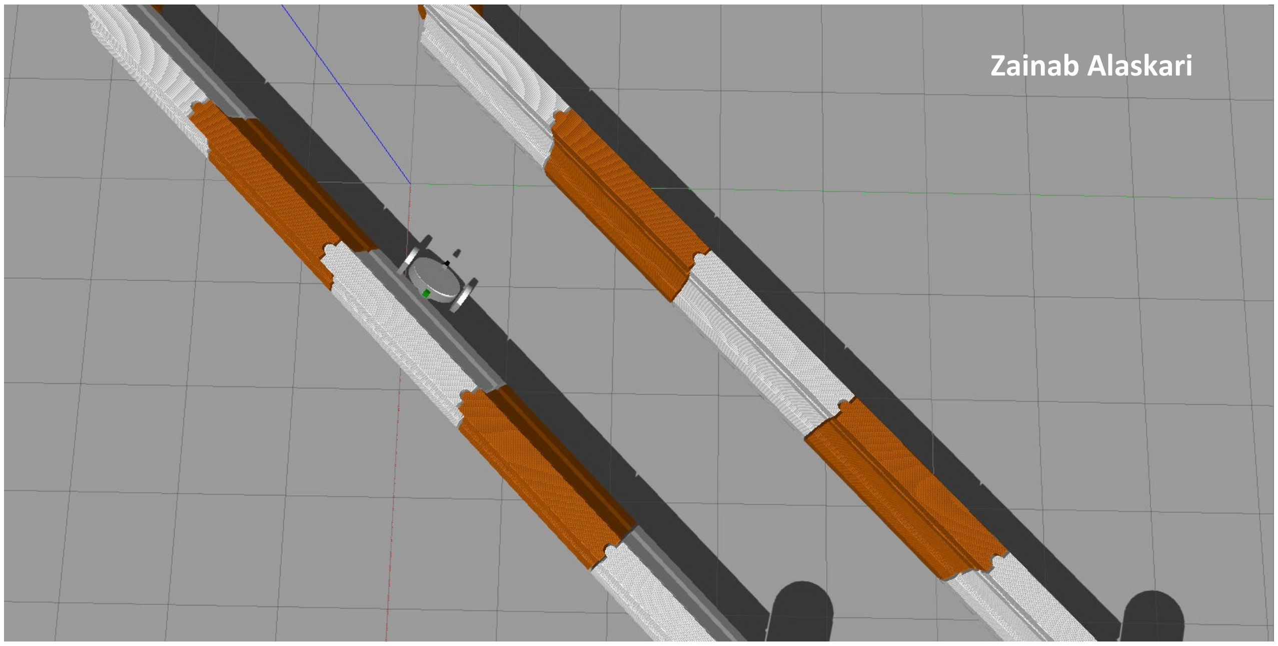
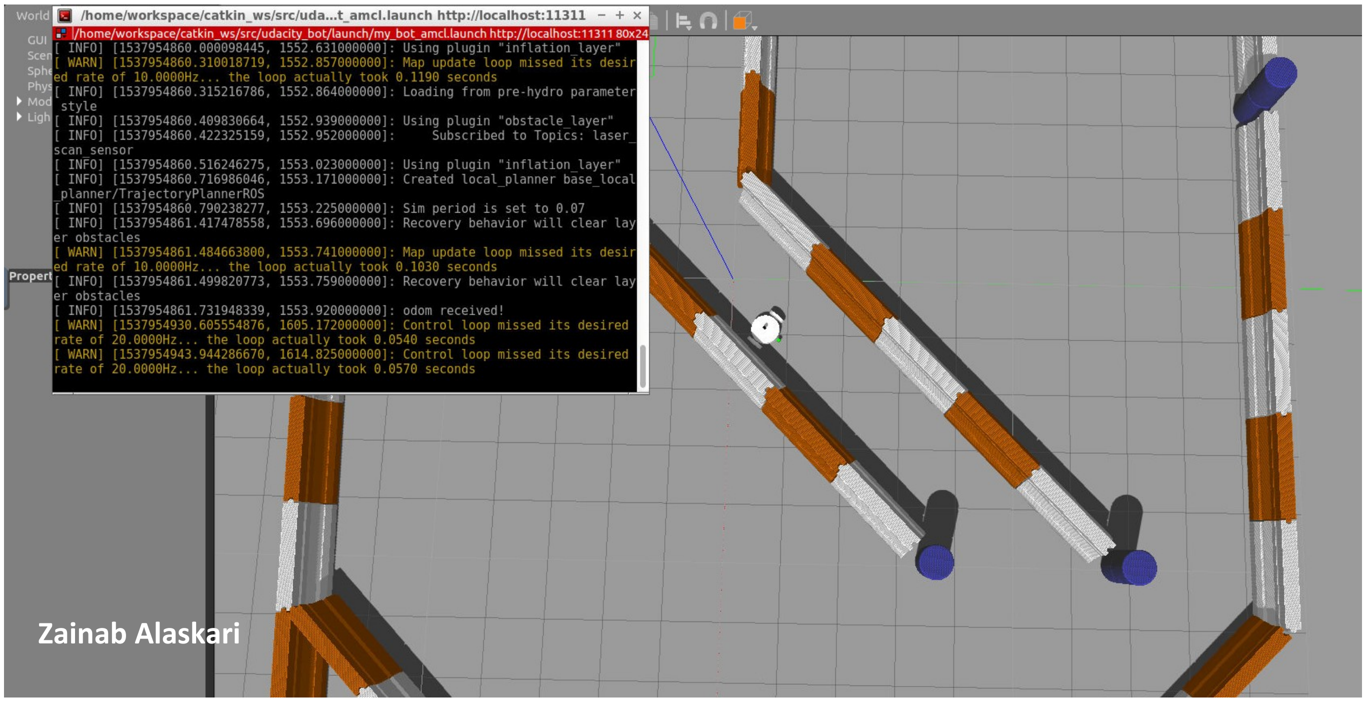


Figure 7. The my bot before tuning its parameters was bumping into the maze wall trying to reach to the goal position.

After tuning the navigation parameters

After tuning the right parameters, it was seen that the navigation of the robot improved significantly; the robot wasn't pumping into obstacles and it was following the correct path.



```
/home/workspace/catkin_ws/src/uda...t_amcl.launch http://localhost:11311
/home/workspace/catkin_ws/src/udacity_bot/launch/my_bot_amcl.launch http://localhost:11311 80x24
[ INFO] [1537954860.000098445, 1552.631000000]: Using plugin "inflation_layer"
[ WARN] [1537954860.310018719, 1552.857000000]: Map update loop missed its desir
ed rate of 10.0000Hz... the loop actually took 0.1190 seconds
[ INFO] [1537954860.315216786, 1552.864000000]: Loading from pre-hydro parameter
 style
[ INFO] [1537954860.409830664, 1552.939000000]: Using plugin "obstacle_layer"
[ INFO] [1537954860.422325159, 1552.952000000]:     Subscribed to Topics: laser_
scan_sensor
[ INFO] [1537954860.516246275, 1553.023000000]: Using plugin "inflation_layer"
[ INFO] [1537954860.716986046, 1553.171000000]: Created local_planner base_local
_planner/TrajectoryPlannerROS
[ INFO] [1537954860.790238277, 1553.225000000]: Sim period is set to 0.07
[ INFO] [1537954861.417478558, 1553.696000000]: Recovery behavior will clear lay
er obstacles
[ WARN] [1537954861.484663800, 1553.741000000]: Map update loop missed its desir
ed rate of 10.0000Hz... the loop actually took 0.1030 seconds
[ INFO] [1537954861.499820773, 1553.759000000]: Recovery behavior will clear lay
er obstacles
[ INFO] [1537954861.731948339, 1553.920000000]: odom received!
[ WARN] [1537954930.605554876, 1605.172000000]: Control loop missed its desired
rate of 20.0000Hz... the loop actually took 0.0540 seconds
[ WARN] [1537954943.944286670, 1614.825000000]: Control loop missed its desired
rate of 20.0000Hz... the loop actually took 0.0570 seconds
```

Figure 8. Robot successfully following the path avoiding obstacles after tuning.

No parameters in the amcl.launch file were changed for this robot model. The only changes were in the following files:

In the base_local_planner_prams.yaml file:

Since the robot was deviating too much from the global path as if attempting to head straight to the goal. The gdist_scale which is the goal influence was reduced to 0.2 and the pdist_scale which is the global path compliance was set to 1.5.

```
  pdist_scale: 1.5

  gdist_scale: 0.2
```

In the cost_map_common_params.yaml file:

The robot radius size was changed to 0.35 m to reflect the new model size, also, the inflation radius was increased to 0.4. This is to take into consideration the new size of the robot in navigation planning and hence avoiding obstacle collision when navigating the path.

```
robot_radius: 0.35

inflation_radius: 0.4
```

In the global_costmap_params.yaml file

Update and publish frequencies were reduced to 8 Hz and the resolution was reduced a bit to 0.05 to free up some computational power.

```
update_frequency: 8.0

publish_frequency: 8.0

width: 10.0

height: 10.0

resolution: 0.05
```

In the local_costmap_params.yaml file:

Update and publish frequencies were reduced to 8 Hz, the resolution was reduced a bit to 0.05, and map size was reduced to 3 x3 m to free up some computational power.

```
update_frequency: 8.0

publish_frequency: 8.0

width: 3.0

height: 3.0

resolution: 0.05
```

Discussion

both the Udacity robot and the custom robot successfully follow the global path completing the navigation with a fairly precise final position and orientation to the goal due to the confident location estimate of each robot's position.

It took a very long time to test the parameters and tune them specially for the custom robot because it always took a lot of processing time to reach the goal postion. Both robots would be believed to behave similarly due to their similar mass and the small difference in size and shape. It was a surprise for me that changing the chassis shape would dramatically change the performance of the robot.
It was very hard trying to make the custom robot perform smoothly and fast in the simulation, and it was still not achieved due to time limitation where no further testing was made to enhance the performance of the custom robot.

It is believed that by freeing up unnecessary computation power the performance can be enhanced by:

- Decreasing the maximum particles number significantly.
- Decreasing the maps resolutions.
- Reducing the maps size further.
- Reducing obstacle range.

The most difficult thing about tuning was knowing from where to start. Knowing which parameters to start from and understanding them fully was key to enhance the robots’ performance. The most significant tuning parameters were:

- The local cost map size. Reducing it made all the other parameters work.
- The update and publish frequencies. Reducing them allowed for the maps to be updated fast enough.
- The transform tolerance. This value set the maximum amount of delay or latency allowed between transforms and hence all the multiple coordinate frames are transformed in time and successfully.
- The inflation radius. Setting this value correctly helps the robot avoid obstacles successfully.

As for the kidnapped robot problem, it must be pointed that the robot would fail to localize when moved to a different location. This is due to the fact that AMCL is a bayesian algorithms that holds an internal belief of the world that is difficult to change in case of abrupt changes in the environment.

One way to go about moving the robot from place to place would be to reset the AMCL internal state, back to a uniform distribution of particles, when placing the robot on a new location. This would reduce the kidnaped robot to a global localization problem.

MCL/AMCL in terms of industry is best implemented in controlled environments such as warehouses and factories where it is required for robots to navigate through a known map of the environment to do certain tasks.

MCL/AMCL could be implemented in amazon warehouses where the shelfs would represent the maze walls and the map of the place would be static, the people or other operating robots would be the obstacles. This robot’s task would be to carry stock around the expansive warehouse floors and group together all the individual items needed for a specific order.

MCL/AMCL could also be used with robot used for servicing in hospitals or malls where the environment is closed and known.

Future Work

The more accuracy required in localization the more the computational power needed, and hence it all depends on the nature of the localization problem and figuring out if accuracy or processing time is more important.

Problems that could require high accuracy in localization could be a pick and place problem where the final pose of the robot is more important compared to the processing time. The processing time can still be enhanced by using powerful processors.

Problems that would need great processing time could be related human servicing as was mentioned above, such robots can be used in malls and hospitals to help humans and so they need to be able to have a fast response. Drones also need to have fast responses to navigate smoothly though the sky.

This tradeoff between accuracy and processing time was best shown in the demonstrated results of both robots. Where the udacity robot had a low processing time and fairly good accuracy. On the other hand, the custom robot had a high processing time and very precise localization results.

To further increase the accuracy in both models the following could be implemented at the expense of processing time:

- Increase maximum particle numbers
- Add more sensors to the robots, such as lidars, ultrasonic, IMU sensors, etc.
- Lower the yaw goal tolerance and xy goal tolerance.
- Increase map resolution.
- Reduce odom alpha parameters.
- Reduce base size area.

To decrease processing time the opposite of what is mentioned above must be done as was mentioned in the discussion section.