Robot localization in a mapped environment using Adaptive Monte Carlo algorithm

Mohamed Ewis

1 Introduction:

This project aims to utilize ROS packages to accurately localize a mobile robot inside a provided map in the Gazebo and RViz simulation environments.

2 Background:

2.1 Localization problem:

Localization is one of the main capabilities a robot should possess, this allows the robot to identify its location and be able to move to a certain goal. In localization, there are three main challenges:

a) Local Localization (position tracking):
This is the easiest localization problem. In this problem, the robot knows its initial pose and the problem is to keep track of the robot's pose as it moves.

b) Global Localization:
In this problem, the robot's initial pose is unknown, and the robot tries to determine its pose relative to the ground truth map; the uncertainty for this type is greater than for local localization.

c) The kidnapped robot problem:
This is just like the global localization problem, except that the robot may be kidnapped at any time and moved to a new location on the map.

2.2 Localization algorithms:

There are several algorithms for a robot to perform localization such as Kalman Filter and Monte Carlo Localization (Particle filter).

2.2.1 Kalman Filter:

The Kalman Filter is an estimation filter and very prominent in controls. It is very good at taking in uncertainty or noise in the measurements in real time and providing accurate estimate of actual variables such as position or velocity of a robot. The Kalman Filter algorithm is based on two assumptions:

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

These assumptions limit the applicability of the Kalman Filter as most mobile robot scenarios will execute nonlinear motions.
The Extended Kalman filter (EKF) addresses these limiting assumptions by linearizing a nonlinear motion or measurement function with multiple dimensions using multidimensional Taylor series.

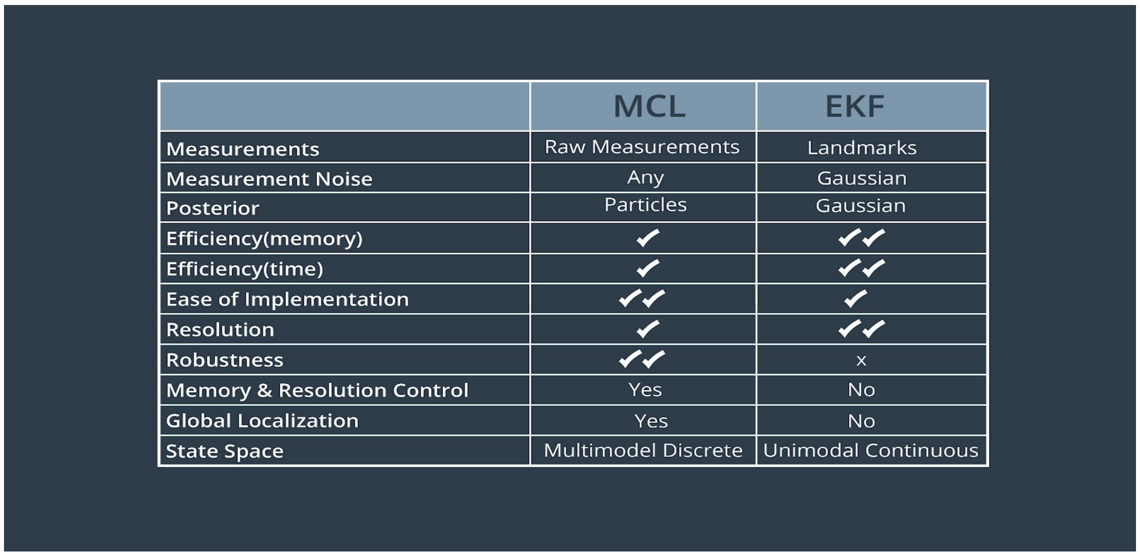
2.2.2 Monte Carlo Filter:

The Monte Carlo Localization (MCL), known as the Particle filter localization 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.

2.2.3 Comparison:

Kalman Filters have limiting assumptions of a unimodal Gaussian probability distribution and linear models of measurement and motion. Extended Kalman filters relax these assumptions but at the cost of increased mathematical complexity and greater CPU resource requirements. Monte Carlo Localization does not have the limiting assumptions of Kalman filters and is very efficient to implement.

MCL vs EKF:



	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

Fig. 1: EKF vs MCL comparison

3 Simulations:

The simulations show the actual performance of the robots. The simulations are done in an environment using Gazebo and RViz tools for visualization. The Navigation Stack as derived from the Willow garage NavStack can be visualized as follows:

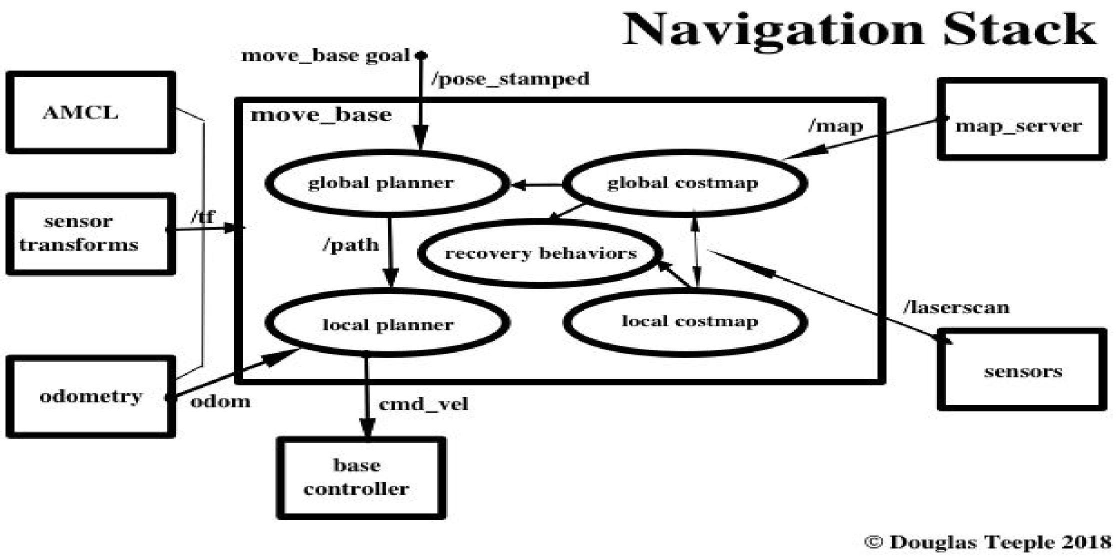


Fig. 2: Navigation Stack

- The whole simulation process was carried out on both UdacityBot and EwisBot. The algorithm and map visualization process were done in RViz. We can also see the robot in action inside the Gazebo simulation environment.

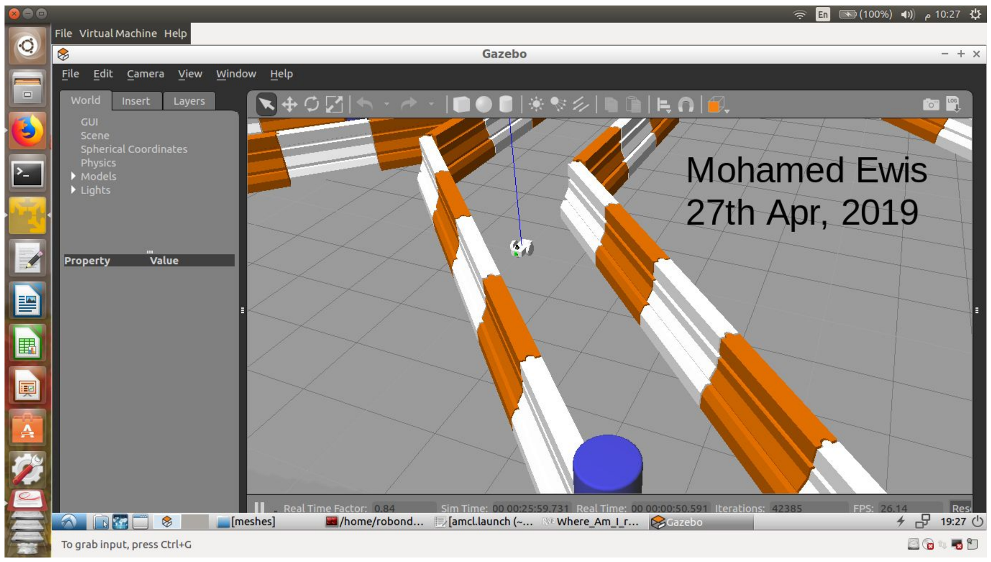


Fig. 3: Robot in Gazebo environment

- At the start of simulation, for both bots, particles are very dispersed indicating great uncertainty in the robot position. At this point, sensors have not yet provided any information regarding the location. Following figure shows the great uncertainty of the robot’s position:

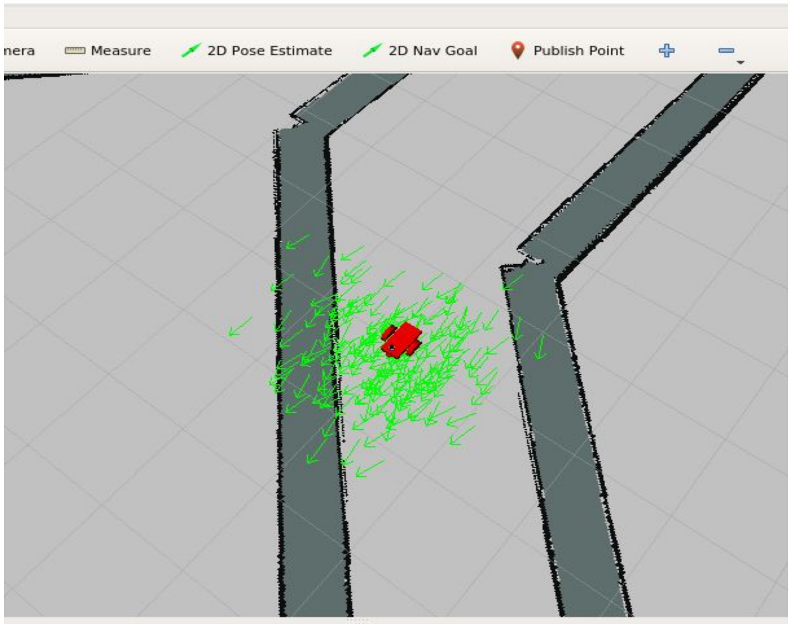


Fig. 4: High uncertainty in robot position

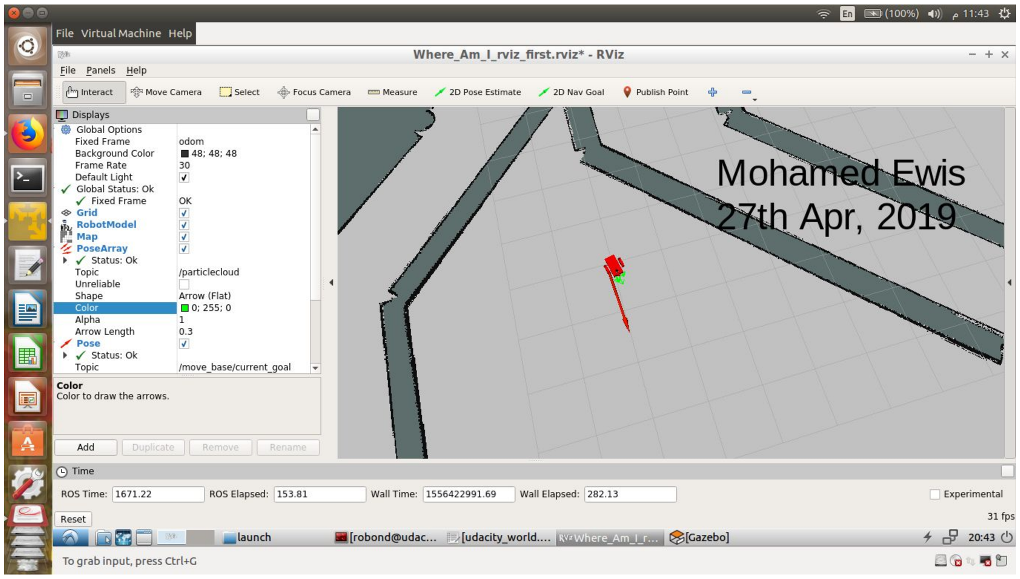
- After the simulation is started, the localization process starts from taking the sensor measurements and gradually improve. After the algorithm converges, the particles effectively depicts the pose of the robot in the map, thus making the robot successfully navigate through the maze and reach the goal state.

3.1 Achievements:

For this project, two robots were deployed in the simulation environment. The benchmark model or UdacityBot and the custom made model or EwisBot.

3.1.1 UdacityBot:

The UdacityBot initially started towards the north of the map as from its local costmap, it calculated the path to the goal from its starting position to be shorter than any other paths. But it discovered the presence of an obstacle and the impossibility of reaching the goal through that path. Then it turned around and reached the goal through the second most shortest path. Following figure shows UdacityBot at the goal position:



Displays	
Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
Map	✓
PoseArray	✓
✓ Status: Ok	
Topic	/particlecloud
Unreliable	☐
Shape	Arrow (Flat)
Color	🟩 0; 255; 0
Alpha	1
Arrow Length	0.3
Pose	✓
✓ Status: Ok	
Topic	/move_base/current_goal

Color
Color to draw the arrows.

Time

ROS Time: 1671.22 ROS Elapsed: 153.81 Wall Time: 1556422991.69 Wall Elapsed: 282.13 Experimental

31 fps

To grab input, press Ctrl+G

Fig. 5: UdacityBot at the goal position

3.1.2 EwisBot:

The EwisBot also has similar behavior and started towards north of the map, before discovering the impossibility of that route and then it reached the goal position through the same route UdacityBot has taken. Following figure shows EwisBot at the goal position:

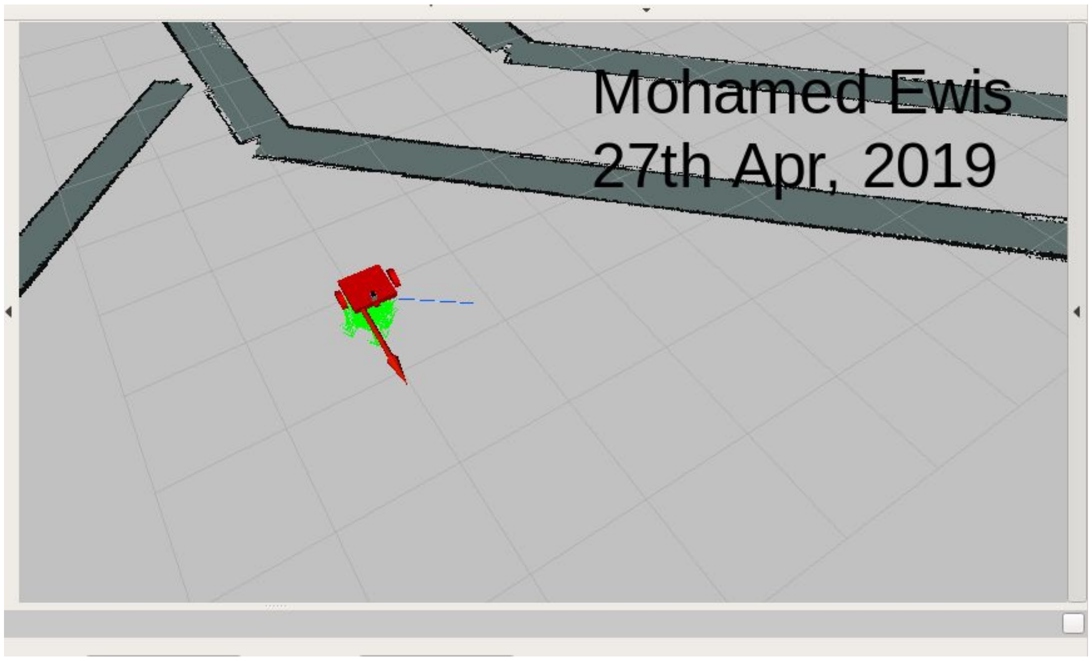


Fig. 6: EwisBot at the goal position

3.2 Benchmark Model - UdacityBot:

3.2.1 Model design:

The robot's design considerations included the size of the robot and the layout of the sensors as detailed below:

3.2.1.1 Maps:

The Clearpath *jackalrace.yaml* and *jackalrace.pgm* packages were used to create the maps.

3.2.1.2 Meshes:

The laser scanner which was used in the robot for detecting obstacles is the Hokuyo scanner. The mesh *hokuyo.dae* was used to render it.

3.2.1.3 Launch files:

Three launch files were used. They are as follows:

1. ***robot_description.launch***: This launch file defines the *joint_state_publisher* which sends fake joint values, *robot_state_publisher* which sends robot states to tf and *robot_description* which defines and sends the URDF to the parameter server.

2. ***amcl.launch***: This file launches the AMCL localization server, the map server, the odometry frame, the move_base server and the trajectory planner server.

3. ***udacity_world.launch***: This launch file contains the robot_description.launch file, the gazebo world and the AMCL localization server. It also spawns the robot and launches RViz.

Following figure depicts the connection graph between the discussed nodes:

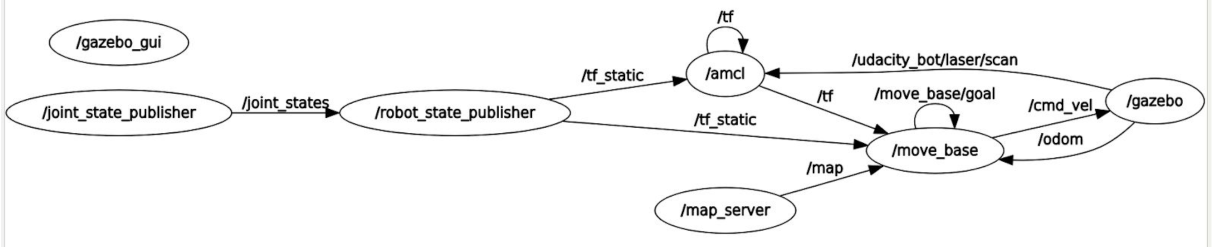


Fig. 7: Node relations

3.2.1.4 Worlds:
Two worlds are defined:

1. Udacity world: This is the original blank world where the robots were created and prototyped. This defines the ground plane, the light source and the world camera.
2. jackal_race world: This world defines the maze.

3.2.1.5 URDF:
The URDF files defines the shape and size of the robot. Two files were used:

1. *udacity_bot.xacro*: Provides the shape and size of the robot in macro format. For the UdacityBot, a fixed base is used. A single link, with the name defined as ”chassis”, encompassed the base as well as the caster wheels. Each link has specific elements, such as the inertial or the collision elements. The chassis is a cuboidal (or box), whereas the casters are spherical as denoted by their ”geometry” tags. Each link (or joint) has an origin (or pose) defined as well. Every element of that link or joint will have its own origin, which will be relative to the link’s frame of reference. For this base, as the casters are included as part of the link (for stability purposes), there is no need for any additional links to define the casters, and therefore no joints to connect them. The casters do, however, have friction coefficients defined for them, and are set to 0, to allow for free motion while moving. Two wheels were attached to the robot. Each wheel is represented as a link and is connected to the base link (the chassis) with a joint. For each wheel, a ”collision”, ”inertial” and ”visual” elements are present. The joint type is set to ”continuous” and is similar to a revolute joint but has no limits on its rotation. It can rotate continuously about an axis. The joint will have it’s own axis of rotation, some specific joint dynamics that correspond to the physical properties of the joint like ”friction”, and certain limits to enforce the maximum ”effort” and ”velocity” for that joint. The limits are useful constraints in regards to a physical robot and can help create a more robust robot model in simulation as well. For the UdacityBot, two sensors were used. A camera and a Laser range-finder (hokuyo sensor).

2. *udacity_bot.gazebo*: This file was included as the URDF file is unable to make the robot take pictures with the camera or detect obstacle with the Laser range-finder. This file contains 3 plugins, one each for the camera sensor, the hokuyo sensor and the wheel joints. It also implements a differential drive controller.

3.2.2 Packages Used:
A ros package called udacity_bot was designed for this project. The structure of this package is shown below:

- config
- images
- launch
- maps
- meshes
- src
- urdf
- Worlds

- This package, along with the AMCL and the navigation stack packages were crucial for a complete simulation of a mobile robot performing and successfully solving the localization problem. Following table describes UdacityBot setup instructions:

UdacityBot Body		
Part	**Geometry**	**Size**
Chassis	Cube	0.4*0.2*0.1
Back and front casters	Sphere	0.0499(radius)
Left and right wheels	Cylinders	0.1(radius), 0.05(length)
Camera sensor	Link origin Shape-size Joint origin Parent link Child link	[0, 0, 0, 0, 0, 0] Box - 0.05*0.05*0.05 [0.2, 0, 0, 0, 0, 0] Chassis Camera
Hokuyo sensor	Link origin Shape-size Joint origin Parent link Child link	[0, 0, 0, 0, 0, 0] Box - 0.01*0.01*0.01 [0.15, 0, 0.1, 0, 0, 0] Chassis Hokuyo

TABLE 1: UdacityBot setup instructions

3.2.3 Parameters:

To obtain most accurate localization results, several parameters were added, tested and tuned. The parameters were iteratively tuned to see what works best for the UdacityBot. **The AMCL parameters** were tuned as follows:

- The min and max particles parameters were set to 25 and 200 in order to prevent over-usage of CPU. Increasing the max_particles did not improve the robot's initial ability to find itself with certainty.
- The transform_tolerance was one of the main parameters to tune. The tf package, helps keep track of multiple coordinate frames, such as the transforms from these maps, along with any transforms corresponding to the robot and its sensors. 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 these transforms. The maximum amount of delay or latency allowed between transforms is defined by the transform_tolerance parameter. It was finally set to 0.2 for the cost-maps and 0.2 for the amcl package.
- The laser model parameters like laser_max_beams, laser_z_hit and laser_z_rand, were kept as default as the obstacles were clearly detected in the local cost-maps with them as is.

- The odom_model_type was kept as diff-corrected as this mobile robot followed a differential drive. There are additional parameters that are specific to this type - the odom_alphas (1 through 4). These parameters define how much noise is expected from the robot's movements/motions as it navigates inside the map. They were kept at 0.005, 0.005, 0.010 and 0.005 respectively obtained from trial-error method. Table 2 depicts the AMCL parameters used.

AMCL Parameters	
Parameter	**Value**
odom_frame_id	odom
odom_model_type	diff-corrected
transform_tolerance	0.2
min_particles	25
max_particles	200
initial_pose_x	0.0
initial_pose_y	0.0
initial_pose_a	0.0
laser_z_hit	0.95
laser_z_short	0.1
laser_z_max	0.05
laser_z_rand	0.5
laser_sigma_hit	0.2
laser_lambda_short	0.1
laser_model_type	likelihood_field
laser_likelihood_max_dist	2.0
odom_alpha1	0.005
odom_alpha2	0.005
odom_alpha3	0.010
odom_alpha4	0.005

TABLE 2: AMCL parameters

The move base parameters were tuned as follows:

- The obstacle_range parameter was modified to have a greater value of 1.5. This parameter depicts the default maximum distance from the robot (in meters) at which an obstacle will be added to the cost-map.
- The raytrace_range parameter was modified to a higher value of 4.0. This parameter is used to clear and update the free space in the cost-map as the robot moves.
- Two parameters in the global and local cost-maps were also changed. They are:

1. update_frequency: This value was set to 10.0. This is the frequency in Hz for the map to be updated.
2. publish_frequency: This value was also set to 10.0. This is the frequency at which the map will be published on the display.

- The_yaw_goal_tolerance was updated to a value of 0.1. This parameter depicts the tolerance in radians for the controller in yaw/rotation when achieving its goal. The xy_goal_tolerance was updated to a value of 0.2. This is the tolerance in meters for the controller in the x-y distance when achieving the goal. Both these parameters were doubled to allow for additional flexibility in trajectory planning. The transform_tolerance was set to 1.25. Table 3 depicts the move_base parameters used.

move_base Parameters	
Parameter	**Value**
yaw_goal_tolerance	0.1
xy_goal_tolerance	0.2
obstacle_range	1.5
raytrace_range	4.0
inflation_radius	0.65
robot_radius	0.3
update_frequency	10.0
publish_frequency	10.0

TABLE 3: move base parameters

3.3 Personal Model - EwisBot:

3.3.1 Model design:
The EwisBot has a similar structure as UdacityBot, but it has a square base and is bigger in size. The laser sensor was also moved to the front of the robot.

Following table describes EwisBot setup instructions:

EwisBot Body		
Part	**Geometry**	**Size**
Chassis	Cube	0.4*0.4*0.1
Back and front casters	Sphere	0.05(radius)
Left and right wheels	Cylinders	0.1(radius), 0.05(length)
Camera sensor	Link origin Shape-size Joint origin Parent link Child link	[0, 0, 0, 0, 0, 0] Box - 0.05*0.05*0.05 [0.2, 0, 0, 0, 0, 0] Chassis Camera
Hokuyo sensor	Link origin Shape-size Joint origin Parent link Child link	[0, 0, 0, 0, 0, 0] Box - 0.1*0.1*0.1 [0.15, 0, 0.1, 0, 0, 0] Chassis Hokuyo

TABLE 4: EwisBot setup instructions

3.3.2 Packages Used:
Same packages as UdacityBot.

3.3.3 Parameters:
The AMCL parameters were kept the same as UdacityBot, but significant changes were made in the move_base parameters. The move_base parameters were tuned as follows:

- The obstacle_range parameter was modified to have a greater value of 4.0. This parameter depicts the default maximum distance from the robot (in meters) at which an obstacle will be added to the cost-map. This was done in a trial-and-error method. The basic intuition behind increasing this parameter was as the EwisBot was larger in size, it moved slower. So additional time was wasted if it was unable to see an obstacle in moderate range and followed the path towards it only to find the region bounded by an obstacle. Hence, in order to increase its sight with respect to the cost-map, a higher value of this parameter was used.
- The raytrace_range parameter was same value of 4.0. This parameter is used to clear and update the free space in the cost-map as the robot moves.

- Two parameters in the costmap_common_params were also changed. They are:

1. update_frequency: This value was set to 10.0. This is the frequency in Hz for the map to be updated.
2. publish_frequency: This value was also set to 5.0. This is the frequency at which the map will be published on the display.

- The yaw_goal_tolerance was updated to a value of 0.01. This parameter depicts the tolerance in radians for the controller in yaw/rotation when achieving its goal. The xy_goal_tolerance was updated to a value of 0.05. This is the tolerance in meters for the controller in the x-y distance when achieving the goal. The transform_tolerance was set to 0.2 . Table 5 depicts the move base parameters used.

move_base Parameters	
Parameter	**Value**
yaw_goal_tolerance	0.01
xy_goal_tolerance	0.05
obstacle_range	4.0
raytrac_ range	4.0
inflation_radius	0.55
update_frequency	10.0
publish_frequency	5.0

TABLE 5: move base parameters for EwisBot

4 Results:

4.1 Localization Results:

4.1.1 Benchmark Model - UdacityBot:

The time taken for the particle filters to converge was around 5-6 seconds. The UdacityBot reaches the goal within approximately two minutes. So the localization results are pretty decent considering the time taken for the localization and reaching the goal. However, it does not follow a smooth path for reaching the goal. Initially, the robot heads towards the north as it was unable to add the obstacle over there in its local cost-map and hence it followed the shortest path to the goal. But soon, it discovered the presence of the obstacle and it changed its strategy to the next shortest route to reach the goal, i.e. head south-east, turn around where the obstacle ends and reach the goal.

- **Launch setup :**
- In a terminal type: roslaunch udacity_bot udacity_world.launch
- In a new terminal type: roslaunch udacity_bot amcl.launch
- in a new terminal run the following: rosrun udacity_bot navigation_goal

4.1.2 Personal Model - EwisBot:
The time taken for the particle filters to converge was around 30-40 seconds. The EwisBot reaches the goal within approximately 8 minutes. So here, a deterioration of the results is observed. This can be attributed to the heavier mass of the EwisBot, even though no wheel slippage was kept for both the robots in their respective URDFs.

- **Launch setup :**
- In a terminal type: roslaunch ewis_bot udacity_world_ewis.launch
- In a new terminal type: roslaunch ewis_bot amcl_ewis.launch
- in a new terminal run the following: rosrun udacity_bot navigation_goal

4.2 Technical Comparison:
EwisBot was significantly heavier than the UdacityBot. EwisBot has a square shape and also the Laser finder was moved to the front.

5 Discussion:

- UdacityBot performed significantly better than EwisBot. This might be attributed to the fact that EwisBot is considerably heavier than UdacityBot. This significantly changed EwisBot's speed and hence the time it takes to reach the goal also increased. EwisBot, which occupies more space than UdacityBot also had a problem navigating with a higher inflation radius as it thought the obstacles to be thicker than they really are and hence deducing that it has not enough space to navigate. As a result, sometimes EwisBot froze in one place trying to figure out an alternate route which took a long time. So, when the inflation radius was decreased, the performance of EwisBot improved considerably.

- The obstacle_range was also another important parameter to tune. EwisBot, being slower in speed naturally wastes a lot of time if it goes to the wrong direction only to find an obstacle there. So in order to be able to successfully add an obstacle to its cost-map from a significantly larger distance, this parameter was increased which resulted in significant improvement of EwisBot's performance.

- In the 'Kidnapped-robot' problem, one has to successfully account for the scenario that at any point in time, the robot might be kidnapped and placed in a totally different position of the world.

- MCL/AMCL can work well in any industry domain where the robot's path is guided by clear obstacles and where the robot is supposed to reach the goal state from anywhere in the map. The ground also needs to be flat and obstacle free, particularly in the cases where the laser range-finder is at a higher position and cannot detect objects on the ground.

6 Conclusion/Future Work:

- Both robots satisfied the two conditions, i.e. Both were successfully localized with the help of AMCL algorithm and both reached the goal state within a decent time limit. UdacityBot performed better than EwisBot which can be attributed to the heavier mass of EwisBot. This time difference was reduced significantly by increasing the obstacle range and inflation radius parameters. The main issue with both the robots was erroneous navigation as both of robots took a sub-optimal route due to the lack of knowledge of an obstacle beforehand. However, as the primary focus of this project was localization, this problem was ignored due to time constraint.

- Placement of the laser scanner can also play a vital role in the robot's navigation skill. Placing the scanner too high from the ground could result in the robot missing obstacles in the ground and then get stuck on it causing a total sensor malfunction. On the other hand, placing the scanner too low may prevent the robot from perceiving better as it gets more viewing range in this situation. So this would be another important modification to improve the robots.

- Future work would involve making both the robots commercially viable by working on making/tuning a better navigation planner.

- Also, the present model deployed only has one robot in a world at a time. In future, this project could be expanded to include multiple robots in the same world, each with the same goal or a different one.

References

[1] http://wiki.ros.org/navigation/Tutorials/Navigationbasic
navigation tuning guide," 2018.

[2]http://wiki.ros.org/costmap_2d