Introduction to Localization

Youssef Khaky

Abstract—Robots, unlike us, do not have an intuitive sense of their location in a room. How can a robot know if it is at the corner of the room or at the center? This problem is known as the localization problem and is what discussed in this paper. Two different algorithms for localization will be discussed, Kalman Filter and the Monte Carlo Localization (MCL), however, only the MCL algorithm will be implemented on a simulated robot. Additionally, important parameters and ROS topics will be discussed in this paper.

1 Introduction

The field of smart robots is a fairly young field. One of the first problems that presents itself in this field is the problem of localization. From a noisy sensor data, how can we accurately estimate the position of our robot in a map? Researching localization will allow us to make advancements in the field of self driving car and autonomous robots. So far, by the use of this technology, companies were able to make products such as the autonomous vacuum cleaner. Other companies were able to use this technology to automate factory work and reduce costs of a production line by many folds. Further exploration into this field is needed as there is still a long way to go to make our localization algorithms more accurate and robust. As localization gets more accurate, we will be able to use robots in more sensitive applications such as in medicine. This paper presents nothing new, its purpose is to provide information about two popular algorithms. First being the Kalman filter approach. One of the first applications for this algorithm was for a NASA Apollo program, it was incorporated within Apollo's navigation system. The second algorithm is the Monte Carlo Localization algorithm which was discovered around the 1970s, it is also known as the particle filter. Both filters are recursive Bayesian estimators. The similarities and the differences between both estimators will be discussed in the paper. There is a project component to this paper were the MCL is implemented on a custom made Gazebo simulated robot. The AMCL package in the navigation stack was used to implement the algorithm.

2 Background

Deducing it's current position from a given map of the environment is known as the localization problem. Coupled with a mapping algorithm, a robot can perform SLAM, giving the robot the ability simultaneously localize itself and map it's environment, making the robot intrinsically able of knowing it's current position. There are different algorithms that can be used to tackle the localization problem, two of the famous ones are the EKF (Extended Kalman Filter) and the MCL (Monte Carlo Localization).

2.1 Kalman Filters

Kalman filter is used to converge towards the actual value of a measurement from a sequence of incoming noisy measurements. In our case, that noisy measurement is a distance measurement. The Kalman filter is a recursive algorithm. There are two stages to it, first is the measurement update, second is the state prediction step.

2.1.1 Measurement Update

During the measurement update, we gain knowledge about the environment which results in reducing the uncertainty of our state variables. We start our Kalman filter with an initial belief of what the measurement is along with an uncertainty paired with that belief, the initial belief and the measurement can be represented as a Gaussian distribution with the estimated value being the mean and the uncertainty being the variance. When we take a new measurement using the sensor, we use it to update our current belief and the variance. We use the following equation to update the mean in the case in 1D. It is analogous to current dividing between two resistors.

$$u' = \frac{r^2 u + \sigma^2 v}{r^2 + \sigma^2}$$

In case of higher dimensions, we represent the measurement in a form of a matrix.

$$z = H = \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ \theta \end{bmatrix}$$

To update the variance such that it includes the inaccuracy from the sensor, we use the following equation (in 1D), which is analogous to finding the equivalent resistance of two resistors connected in parallel.

$$\sigma^{2'} = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

In higher dimensions this is calculated using the following equation.

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

Where P' is the state covariance. That is calculated in the state transition step which will be discussed in the next section.

Figure 1 shows the posterior distribution calculated from the measurement distribution and the prior belief.

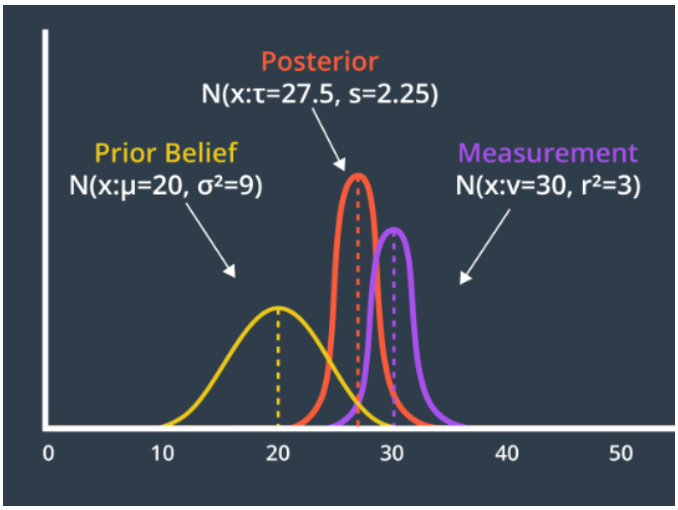

Fig. 1. Mean updated after a measurement update step [1]

2.1.2 State prediction

After a robotic movement, we need to estimate our current location while taking into account the motion that just happened. That motion is not accurate, the robot could have slowed down due to unexpected weather conditions or it may have run faster than expected. The wheels of the robot could have also slipped. These factors make our motion uncertain. To update our mean and variance from an uncertain motion, we use the following equation in the case of one variable.

$$u' = u_1 + u_2$$
$$\sigma^{2'} = \sigma_1^2 + \sigma_2^2$$

In the case of higher diminutions we use the following equations for updating the mean.

$$X' = FX$$

$$\begin{bmatrix} X \\ Y \\ \theta \end{bmatrix}' = F \begin{bmatrix} X \\ Y \\ \theta \end{bmatrix}$$

Were F is the State Transition Function, with dimensionality of nxn were n is the number of variables. To update the state covariance matrix we use the following equation.

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

Were Q represents the process noise.

Consider figure 2 showing motion update in 1D. The uncertain motion is used to update the prior belief. In this process, unlike measurement update, we lose information, this results in the uncertainty getting bigger, which is the case as represented by the posterior getting wider.

2.1.3 Kalman-Gain in Higher Dimensions

In case of 2 state variables or more, we use the kalman filter to determine which should we trust more, state prediction or measurement update. So it is expected that the Kalman-gain to be a function of both the sensor uncertainty S from the measurement update and motion uncertainty P from the state prediction.

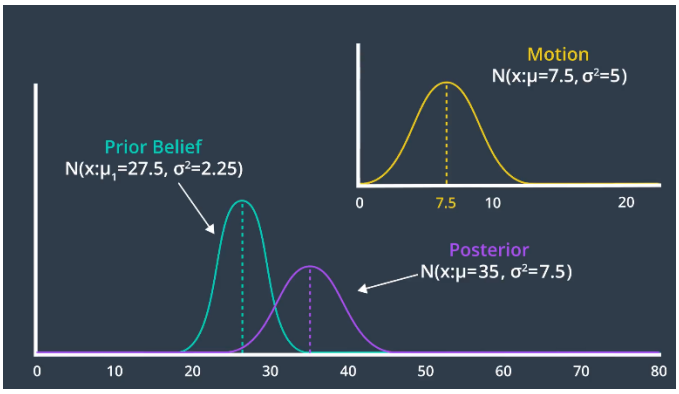

Fig. 2. motion update [1]

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

2.1.4 Linear Kalman Filter Vs Extended Kalman-Filter

What was described above is a linear Kalman filter. The linear Kalman filter operates on the bases of two assumptions.

- Motion and measurement models are linear. (The F Matrix is linear)
- State space can be represented by a unimodal Gaussian distribution.

These two assumptions are violated by most robots. Many robots do not have a linear state transformation matrix. However a linear state transformation is required to maintain the Gaussian shape of our estimation. To tackle that issue we linearize the non-linear state transformation using Taylor Series. That linearized state approximation is used to generate the variance of the new mean, thus maintaining the Gaussian nature of the estimation.

2.2 Particle Filters

The particle filter, (also known as the Monte Carlo Localization), is used to localize the robot using a provided environment map. A particle is a virtual element that resemble a robot. Each particle has a position and orientation. Initially the particles are distributed uniformly across the map such that there is no bias towards an initial position. i.e. the robot is equally likely to be at any position on the map.

Every particle has a weight attached to it, that weight gets re-evaluated after every evaluation step with the input being the LIDAR sensor's distance measurements. During the reevaluation step of the particles, some are removed, with the particles having the lightest weight being most likely to be removed. As the robot explores the map more, the particle_could uncertainty starts getting smaller. That is depicted by the reduction in the spread of the particles as the robot. The particles start to converge to one location in the map. Ideally there would be only one location representing the true location of the robot, however because the estimation is not 100% accurate, the particles do not converge to one point and remains uncertain to some degree.

Algorithm MCL(X_{t-1}, u_t, z_t):
$\bar{X}_t = X_t = \emptyset$
for $m = 1$ to M:
$x_t^{[m]}$ = motion_update($u_t, x_{t-1}^{[m]}$)
$w_t^{[m]}$ = sensor_update($z_t, x_t^{[m]}$)
$\bar{X}_t = \bar{X}_t + \left\langle x_t^{[m]}, w_t^{[m]} \right\rangle$
endfor
for $m = 1$ to M:
draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\propto w_t^{[m]}$
$X_t = X_t + x_t^{[m]}$
endfor
return X_t

Fig. 3. MCL algorithm having two loops, motion update and re-sampling process [1]

2.3 Kalman Filter VS MCL

The table below compares between the MCL algorithm with the Kalman filter.

	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. 4. table comparing MCL and EKF, [1]

3 Simulations

The robot used in this environment is a very basic model. It has two actuated wheels and two caster wheels. An RGBD camera and LIDAR as inputs. The map is obtained online and was designed by clear path robotics. There are two robot's used. The first one (Udacity model, main model) is the one shown in figure 5. The second one (My model, Modified model, second robot) is the same model except that it is shrunk by a factor of 2. It is shown in figure 6. The goal of using two robots is to compare between their success in localizing the robot. There is a reason why the second robot chosen is a shrunken version of the first robot, the next section discussed why.

3.1 Model design

The robot designed is the same robot provided by Udacity but is scaled down by a factor of two. The Udacity robot would often get stuck at turns, and it was thought that this happened due to the robot's size. Therefore, the new design with everything halved was done.

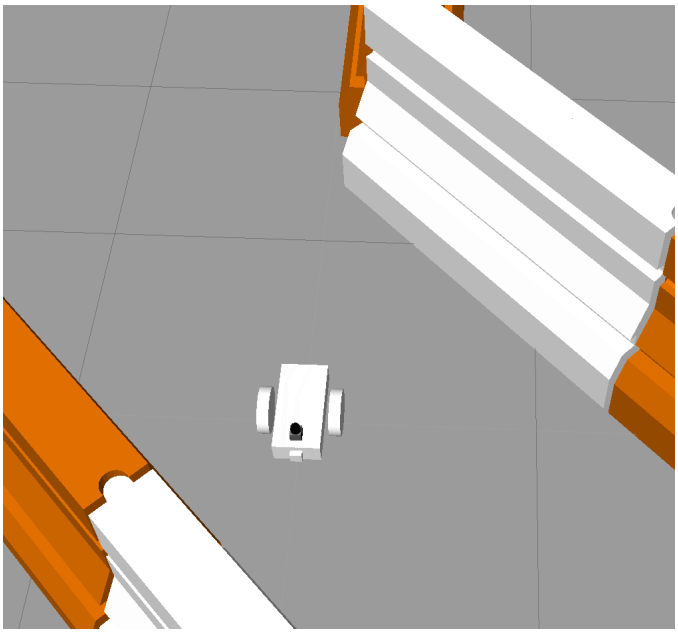

Fig. 5. Udacity's robot model

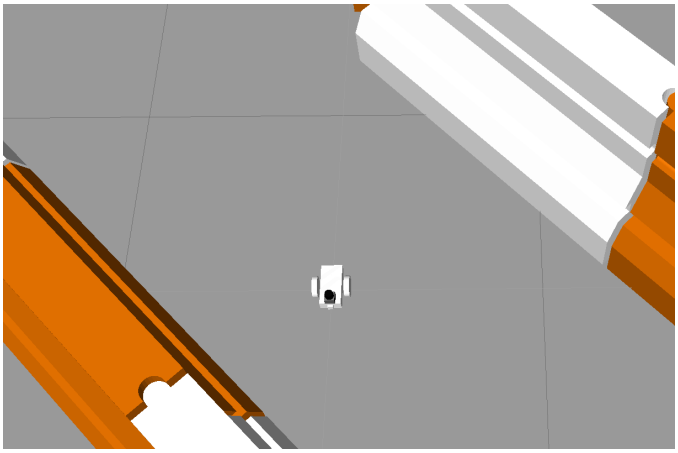

Fig. 6. Modified robot model with all dimensions halved

Another problem that motivated this design was that the wheels were getting detected as an obstacle, with the reduced size, that would no longer happen.

The table below shows the difference between the Udacity bot and the modified bot.

	Udacity Bot	Modified
Length of base	0.4	0.2
Width of base	0.2	0.1
Radius of wheel	0.1	0.05
wheel Separation	0.4	0.2

3.2 Parameters

There are many parameters to tune in the AMCL node and the move_base node, only few are discussed in this paper. There are 5 places where the parameters are defined in this project:

- AMCL launcher

- config/base_local_planner_params.yaml
- config/costmap_common_params.yaml
- config/global_costmap_params.yaml
- config/local_costmap_params.yaml

3.3 AMCL parameters

In the AMCL launcher, all the parameters pertaining to the particle cloud and localization are located. Those that are not defined explicitly, as showing in figure 7 are set to their default value.
Here is a list of all the parameters and their values:

- base_frame_id: robot_footprint
- beam_skip_distance: 0.5
- beam_skip_threshold: 0.3
- do_beamskip: false
- first_map_only: false
- global_frame_id: map
- gui_publish_rate: 10.0
- kld_err: 0.22
- kld_z: 0.99
- laser_lambda_short: 0.1
- laser_likelihood_max_dist: 1.5
- laser_max_beams: 47
- laser_max_range: -1.0
- laser_min_range: -1.0
- laser_model_type: likelihood_field
- laser_sigma_hit: 0.2
- laser_z_hit: 0.95
- laser_z_max: 0.0
- laser_z_rand: 0.05
- laser_z_short: 0.1
- max_particles: 5000
- min_particles: 500
- odom_alpha1: 0.2
- odom_alpha2: 0.2
- odom_alpha3: 0.2
- odom_alpha4: 0.2
- odom_alpha5: 0.2
- odom_frame_id: odom
- odom_model_type: diff
- recovery_alpha_fast: 0.0
- recovery_alpha_slow: 0.0
- resample_interval: 1
- restore_defaults: false
- save_pose_rate: 0.5
- tf_broadcast: true
- transform_tolerance: 0.5
- update_min_a: 0.2
- update_min_d: 0.25
- use_map_topic: false

Some of the important parameters set are, minimum particles (set to 50), maximum particles (set to 5000), update min d (0.25), update min a (0.2) and kld err.
Figure 22 in the appendix shows the dynamic reconfiguration menu in rqt.

3.3.1 Update min d and a

Referring back to the algorithm in figure 3, this parameter dictates how often the first loop (measurement update) will

```
<!-- Localization-->
<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <!--rosparam file="$(find my_amcl_launcher)/params/my_amcl
  <remap from="scan" to="udacity_bot/laser/scan"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="odom_model_type" value="diff"/>
  <param name="base_frame_id" value="robot_footprint"/>
  <param name="global_frame_id" value="map"/>
  <param name="min_particles" value="50"/>
  <param name="max_particles" value="5000"/>
  <param name="gui_publish_rate" value="10.0"/>
  <param name="kld_err" value="0.00005"/>
  <param name="update_min_d" value="0.25"/>
  <param name="update_min_a" value="0.2"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value=".001"/>
  <param name="laser_max_beams" value="60"/>
  <param name="laser_max_range" value="12"/>
  <!--param name="laser_z_hit" value="0.5"/-->
  <param name="laser_max_beams" value="5"/>
  <param name="laser_likelihood_max_dist" value="1.5"/>
</node>
```

Fig. 7. Showing the AMCL launcher explicitly defining some parameters

be executed. Setting this to high numbers will result in the robot loosing track of it's location. If it is set to low numbers, it should result in high computation cost as the loop is being repeated a lot and not enough computation resources is given to other functions such as re-sampling, generating cost maps, etc. However, no significant effects were seen from setting this to very low values, therefore the effects of this needs to be further explored.

3.3.2 Re-sampling interval

Referring back to the algorithm in figure 3, this parameter dictates how often the second (re-sampling) loop should take place. If this number is high, then a lot of bad estimations will lurk around for a long time. This should be increased only it computation is a concern, however the default value of 1 is good.

3.3.3 Number of particles

The more the particles the better the estimation, however the more the number of particles the more the computation. In figure 3 showing the algorithm, M represents the number of particles. Measurement update and re-sampling measurement loops have to be repeated M times each, therefore increasing the number of particles will linearly increase the computation.

3.3.4 laser_min_range and max_range

As the name suggests, values closer than min and further than max will not be considered for the localization. This is good is there is a part of the robot that blocks a beam from the LIDAR. It is not an actual obstacle therefore it should not be taken into consideration by the AMCL algorithm.

3.4 Move_base parameters

There are three main parameter sections to be configured for the move_base, here is the structure.

- move_base
  - Trajectory Planner
  - global_costmap
    - inflation_layer
    - obstacle_layer
    - static_layer
  - local_costmap
    - inflation_layer
    - obstacle_layer

Let's start with the move_base parameters

3.4.1 move_base

All these were not explicitly set, so these parameters are set to the default values:

- clearing_rotation_allowed: true
- conservative_reset_dist: 3.0
- controller_frequency: 20.0
- controller_patience: 5.0
- max_planning_retries: -1
- oscillation_distance: 0.5
- oscillation_timeout: 0.0
- planner_frequency: 0.0
- planner_patience: 5.0
- recovery_behavior_enabled: true
- restore_defaults: false
- shutdown_costmaps: false

3.4.2 Trajectory Planner Parameters

This is the list of all the parameters. Only few of them were explicitly set, the rest were set by default. Those who were explicitly set were set in the "base_local_planner_params.yaml" config file. The config file was called in the launch file under the move_base section using the following line:
rosparam file="$(find udacity_bot)/config/base_local_planner_params.yaml" command="load" />
List of all trajectory planner parameters and their values:

- acc_lim_theta: 3.2
- acc_lim_x: 2.5
- acc_lim_y: 2.5
- angular_sim_granularity: 0.025
- dwa: false
- escape_reset_dist: 0.1
- escape_reset_theta: 1.57079632679
- escape_vel: -0.1
- gdist_scale: 0.8
- heading_lookahead: 0.325
- heading_scoring: false
- heading_scoring_timestep: 0.1
- holonomic_robot: false
- max_vel_theta: 1.0
- max_vel_x: 0.4
- min_in_place_vel_theta: 0.2
- min_vel_theta: -1.0
- min_vel_x: 0.1
- occdist_scale: 0.01
- oscillation_reset_dist: 0.05
- pdist_scale: 0.6
- restore_defaults: false
- sim_granularity: 0.025
- sim_time: 1.7
- simple_attractor: false
- vtheta_samples: 20
- vx_samples: 20
- y_vels: ''

3.4.2.1 Holonomic and y_vels:

During this project the robot got stuck around turns very frequently, after some investigation it was noticed that move_base was sending y velocity commands to the robot which shouldn't have been the case because the robot can only be actuated in one linear direction. This must always be set to false unless the robot is capable of moving sideways.
The y_vels are the velocities that the robot will attempt to use.

3.4.2.2 Acceleration constraints:

Acceleration in theta,x and y dictates the maximum acceleration that the robot can take. It is a good idea to accelerate and decelerate slowly such that the plane of the LIDAR beam is always parallel to the floor plane. If robot accelerates or decelerates sharply, the robot will tilt long it's pitch (y axis) causing errors in the LIDAR measurements which may confuse the AMCL.
Also high accelerations will cause the robot to tilt and spin uncontrollably.

3.4.2.3 Velocity constraints:

Velocity in theta, x and y that is too high may cause the robot to do a measurement update very frequently and stress the resources. This is the same problem mentioned in section 3.3.1.

3.4.3 global costmap

Those parameters that are explicitly defined are defined in the "global_costmap_params.yaml" and "costmap_common_params.yaml". Those that aren't are assigned to their default value. Some of the parameters below are also found in the local costmap (as shown in the next section), that's why we use "common params" yaml file, so we define them explicitly only once.

- footprint_padding: 0.01
- footprint: '[]"

- height: 40
- origin_x: 0.0
- origin_y: 0.0
- publish_frequency: 30.0
- resolution: 0.05
- robot_radius: 0.3
- transform_tolerance: 0.3
- update_frequency: 10.0
- width: 40
- inflation_layer plugin
  - cost_scaling_factor: 0.0
  - enabled: true
  - inflate_unknown: false
  - inflation_radius: 0.55
- obstacle layer plugin
  - combination_method: 1
  - enabled: true
  - footprint_clearing_enabled: true
  - max_obstacle_height: 2.0
- static_layer plugin
  - enable

3.4.3.1 Dimensions, Origin and Resolution:
The hight and width defines the map size in m. Resolution sets the resolution of the map, meter/cell. X and Y origin set the map origin in global frame in meters.

3.4.3.2 publish_frequency:
Number of times per second to publish the map. This is limited by the hardware constraint. A good value for the machine used was found to be 10Hz.

3.4.3.3 Robot radius, inflation and padding:
The distinction between robot radius and inflation is not clear. However both of them expand the obstacles in the costmap so the robot would have enough clearance if it gets close to an obstacle.

3.4.3.4 changing the inflation factor:
This value is related to how fast the inflated area cost decay with distance. "The cost function is computed as follows for all cells in the costmap further than the inscribed radius distance and closer than the inflation radius distance away from an actual obstacle" [2]. Increasing the factor will decrease the resulting cost values due to the multiplication by negative.

$$e^{(-1.0*f*(d-r))} * (i-1)$$

Where:
f = cost scaling factor
d = distance from obstacle
r = inscribed radius
i = costmap_2d::INSCRIBED_INFLATED_OBSTACLE

3.4.4 local costmap

Those parameters that are explicitly defined are defined in the "local_costmap_params.yaml" and "costmap_common_params.yaml". Those that aren't are assigned to their default value.

- footprint: '[]'
- footprint_padding: 0.0
- height: 3
- origin_x: 0.0
- origin_y: 0.0
- publish_frequency: 10.0
- resolution: 0.03
- robot_radius: 0.3
- transform_tolerance: 0.3
- update_frequency: 10.0
- width: 3
- inflation_layer plugin
  - cost_scaling_factor: 0.0
  - enabled: true
  - inflate_unknown: false
  - inflation_radius: 0.4
- obstacle_layer
  - combination_method: 1
  - enabled: true
  - footprint_clearing_enabled: true
  - max_obstacle_height: 2.0

3.4.5 changing the inflation radius

Setting this value too small should be avoided. It should be set to the smallest possible such that the robot doesn't hit the obstacle, and such that gaps in the obstacles would filled as this gets inflated. There are two fields, robot_radius and footprint_padding that seem redundant. They appear to be doing the same thing as inflation radius. Interestingly though, the inflation radius puts a limit on the other 2. $robot\_radius + footprint\_padding < inflation\_radius$.

3.4.6 Config files

The parameters that are explicitly defined are defined in the 4 files in the config folder of the package.

Here is how the configbase_local_planner_params.yaml looks like:

```
TrajectoryPlannerROS:
  max_vel_x: 0.45
  min_vel_x: 0.1
  max_vel_theta: 1.0
  min_in_place_vel_theta: 0.4
  acc_lim_theta: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5
  holonomic_robot: false
  meter_scoring: true
```

Here is how configcostmap_common_params.yaml looks like:

```
map_type: costmap
obstacle_range: 3
raytrace_range: 3.0
transform_tolerance: 0.3
robot_radius: 0.3  #0.3 works
inflation_radius: 0.55 ##0.55 works
observation_sources: laser_scan_sensor
laser_scan_sensor: {sensor_frame: hokuyo,
data_type: LaserScan, topic: /udacity_bot/las
er/scan, marking: true, clearing: true}
```

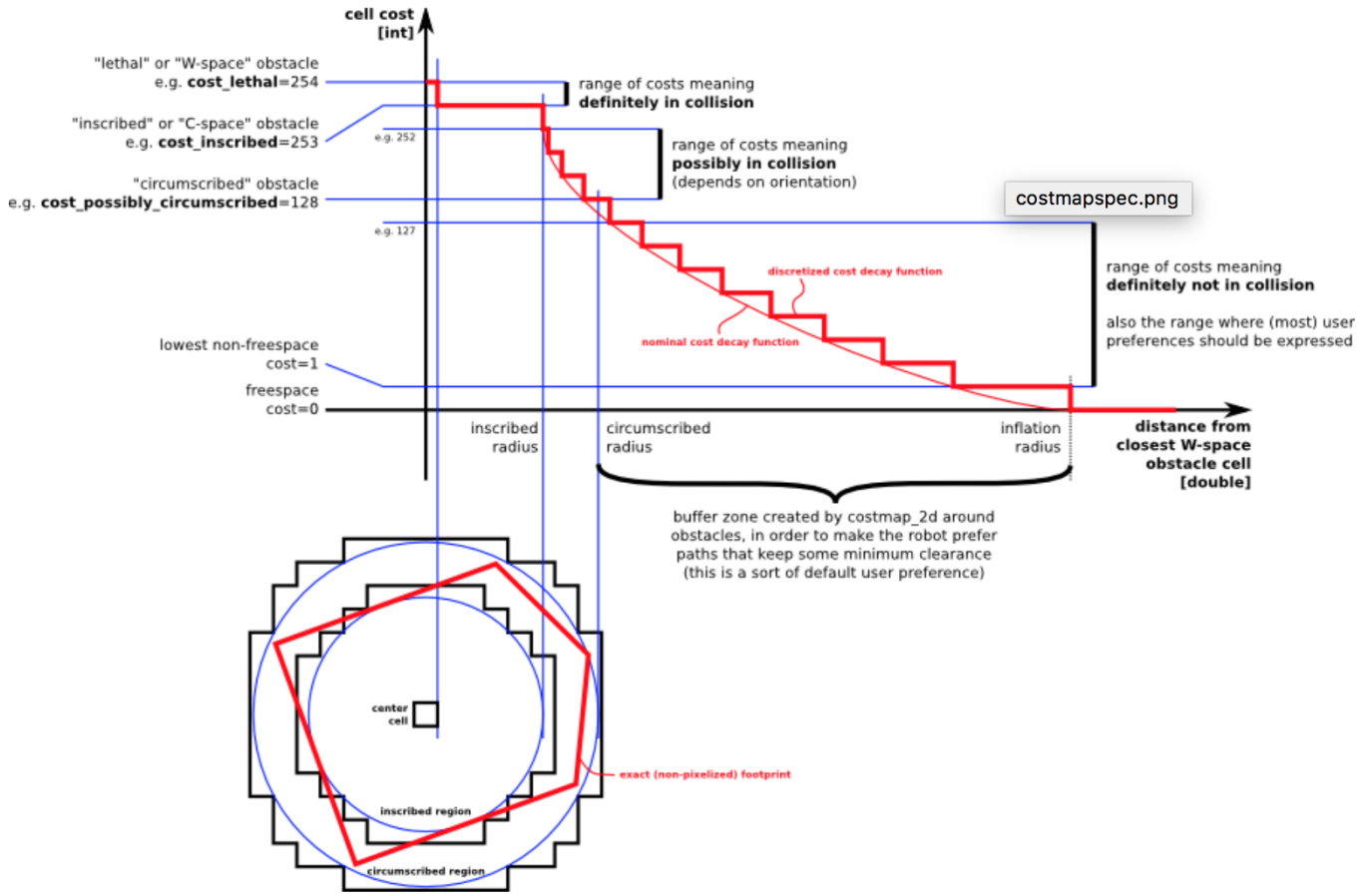

Fig. 8. costmap cell value vs distance [2]

Here is how the global_costmap_params.yaml looks like.

```
global_costmap:
   global_frame: map
   robot_base_frame: robot_footprint
   update_frequency: 10.0
   publish_frequency: 30.0
   width: 40.0
   height: 40.0
   resolution: 0.05
   static_map: true
   rolling_window: false
```

Here is how the local_costmap_params.yaml looks like:

```
local_costmap:
   global_frame: odom
   robot_base_frame: robot_footprint
   update_frequency: 10
   publish_frequency: 10
   width: 3.0
   height: 3.0
   resolution: 0.03
   static_map: false
   rolling_window: true
```

4 Nodes and Topics

4.1 rqt graphs

Figure 9 below shows the graph after running "roslaunch udacity_bot udacity_world.launch".

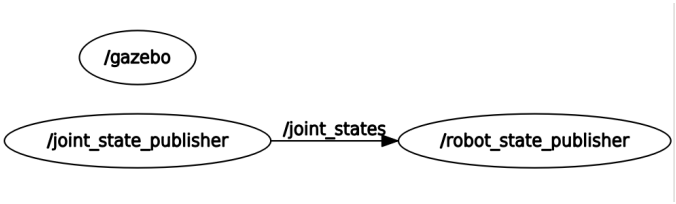

Fig. 9. rqt_graph before the AMCL node was launched

After AMCL node was launched using "roslaunch udacity_bot amcl.launch", the graph got substantially bigger as shown in figure 10.

Figure 11 shows the nodes and topics that were running after launching the navigation goal node using "rosrun udacity_bot navigation_goal"

Figure 12 shows the list of nodes that were running during goal navigation.

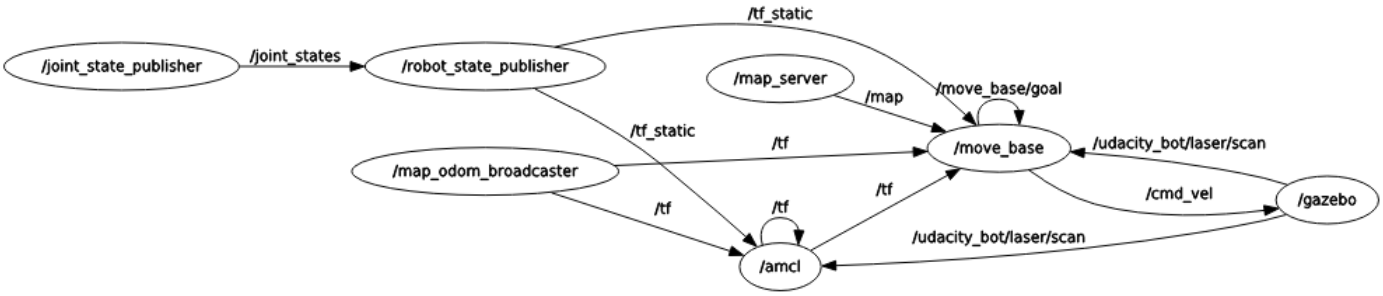

Fig. 10. after launching _AMCL node

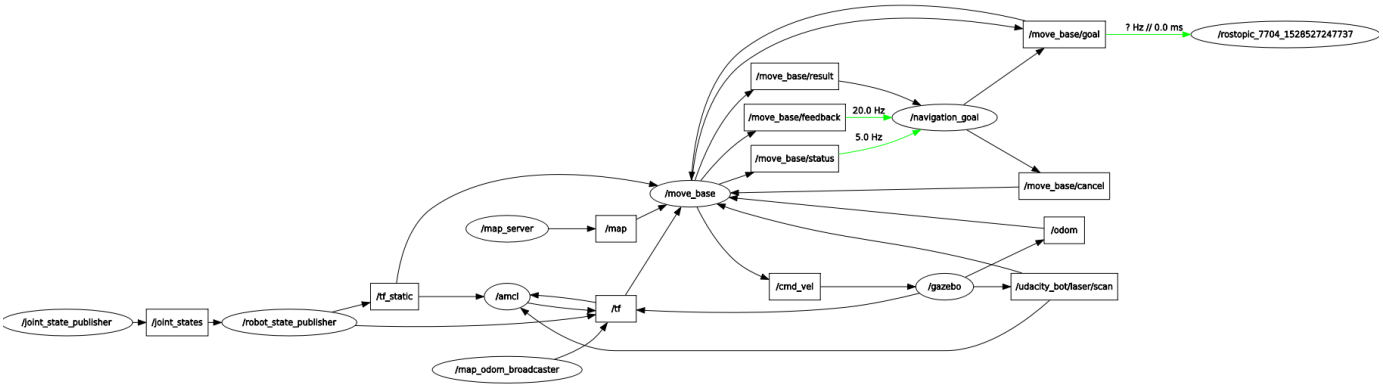

Fig. 11. End position for the modified robot - Gazebo

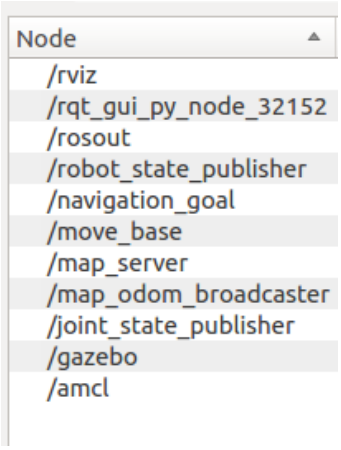

Fig. 12. node list during goal navigation

4.2 Topics

This section highlights some of the important topics used in this project.

1) odom
2) particlecloud
3) udacity_bot \laser \scan
4) cmd_vel
5) move_base: action topics
    - goal
    - plan
    - status
    - result
    - cancel

4.2.1 Odom

The odom topic is the topic in charge of tracking the location and the speed of the robot. It has 4 main fields:

- child_fram_id: contains the frame name that is being tracked
- header: some ovearhead information
- pose: the position and orientation of the robot in reference to it's initial position
- twist: the linear and angular velocity of the robot

Figure 13 shows an example of how the odom topic looks like:

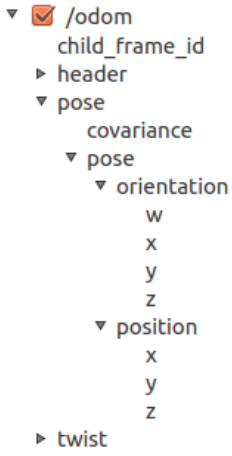

Fig. 13. odom topic structure

4.2.2 Particle Cloud

The particle_cloud topic is the topic in charge of tracking the poses of all the particles estimating the position of the robot. It has 2 main fields:

- header: contains meta information such as the frame id of the map.
- poses: an array of poses of length according to number of particles.

When AMCL node starts, the number of particles is the max possible, as the robot moves around, the number of particles needed to localize the robot decreases.

Figure 14 shows an example of how the particle_cloud topic looks like:

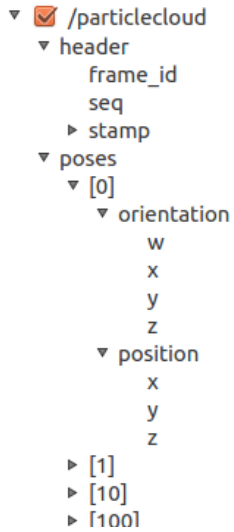

Fig. 14. particle_cloud topic structure

4.2.3 Command Velocity

The command velocity topic is where the orders are sent to the wheels so they would actuate. It has 2 main fields. This is the topic that the move_base node sends commands to.

- linear: contains the linear speed
- angular: contains the angular speed

Figure 15 shows an example of how the cmd_vel topic looks like:

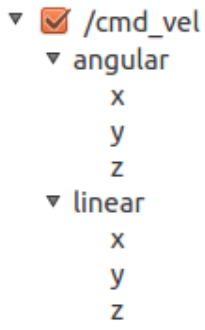

Fig. 15. cmd_vel topic structure

4.2.4 Move Base

move_base is a node that provides an implementation of an action. The navigation_goal node is the action client. As any action server client pair communicate with each other using the following 5 topics.

- goal: the topic through which the client sends the sever the desired goal pose of the robot.
- feedback: the topic through which the server tells the client about the progress of the goal.
- result: sent only once from the server to the client after goal completion.
- cancel: sent from client to server if client wants to cancel.
- status

5 RESULTS

The environemnt was laucnhed followed by the amcl node. After that a navigation goal was sent move_base action server. The action client is navigation_plan, it was run using the rosrun command.
The picture below shows the trajectory followed by the robot. Both robot models (shrunk and actual size) had the same trajectory.

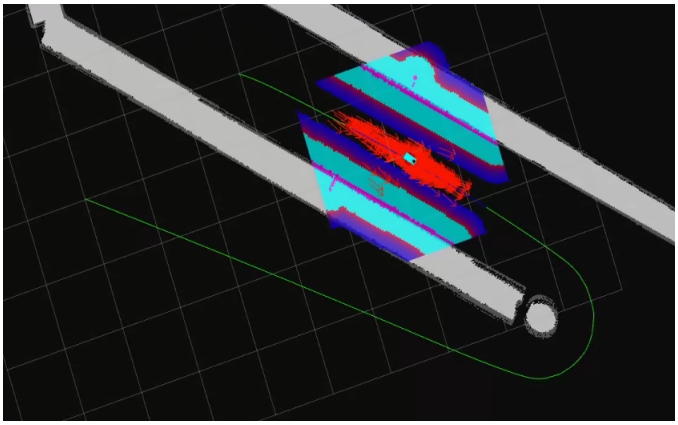

Fig. 16. Full plan as published by move_base/navfnROS/plan topic

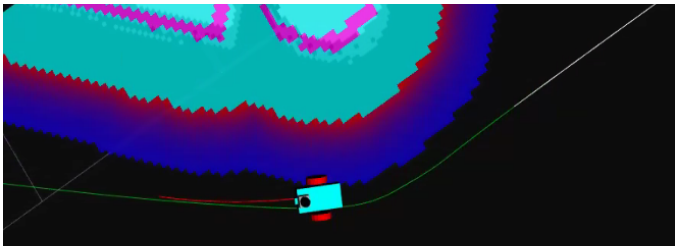

Fig. 17. Full plan (white), global plan (green) and local plan(red)

In some runs using the original robot, the robot got stuck in a turn on it's way to the goal. That occurred very frequently. This is what motivated the design of the smaller robot. It was hypothesized that if the size was reduced, perhaps the robot would not get stuck as it would be more flexible around the turns. However, that wasn't the case. The small robot still got stuck.
Later in the project it was found out that the this was due to setting the parameter of holonomic as true. This was discussed in the parameters section.

When running navigation_goal for both robots, the result was exactly the same. Figure 19 and 20 show the end position of both robots. The arrows are localized in close proximity of each other and are pointing towards the same general direction and are centralized to the robot. That is a good result showing that the AMCL parameters are adequately tuned.

It takes both robots 55 seconds to reach the goal from the spawning/ initial position at point (0,0). Both robots follow the trajectory as expected if the local costmap is 5x5 and not more. If local costmap is big, the robot local planner give unexpected results. It stops following the global planner and starts going up and loop around itself.

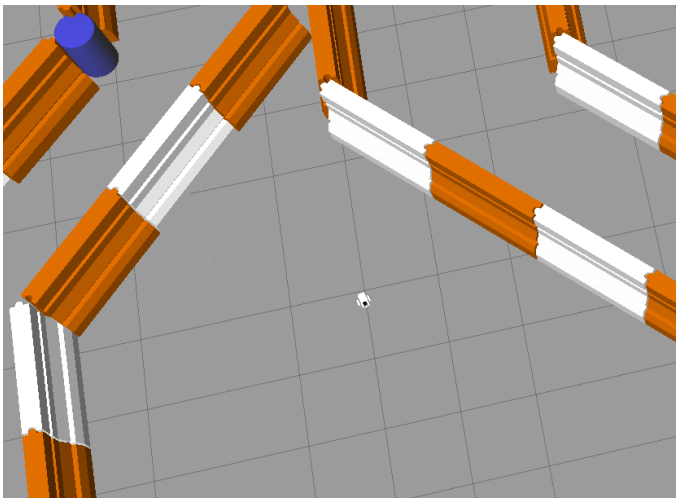

Fig. 18. End position for the modified robot - Gazebo

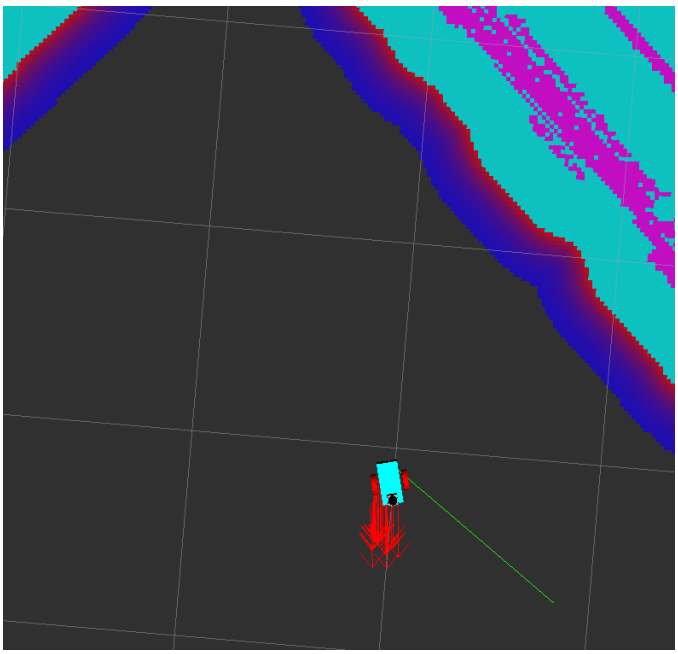

Fig. 19. End position for the modified robot - RVIS

Even thought the holonomic parameter was set to false, the large robot still gets stuck sometime. There were no command velocities being sent to the robot when it got stuck. There is no incident were the small robot got stuck around the turn. The notorious turn in question is shown in figure 21.
Both robot display unexpected behavior around the turn, they sometimes pause and start spinning in place once or twice.
More often than now, the robot follows the path smoothly to the goal pose.

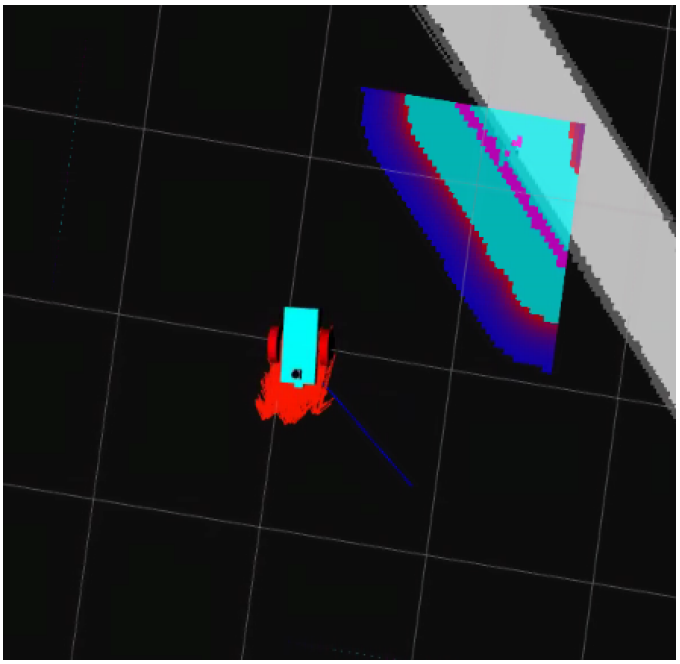

Fig. 20. Big robot after goal is reached

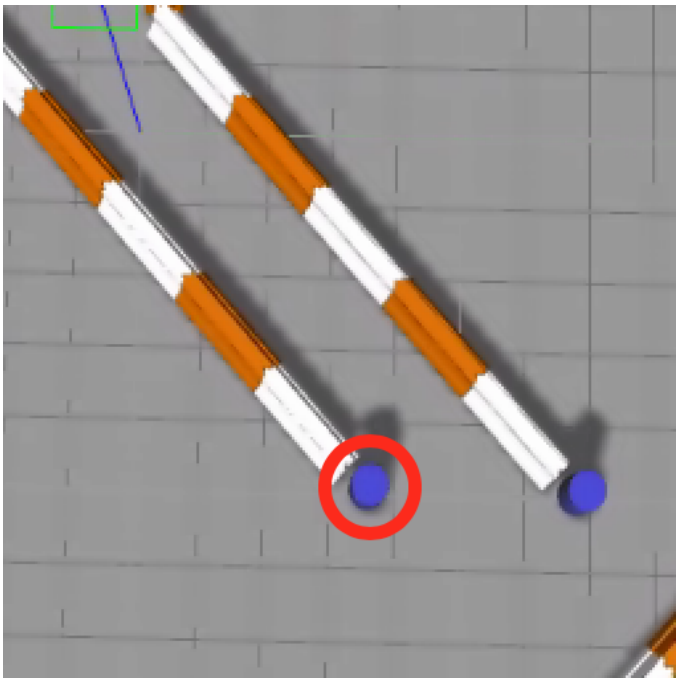

Fig. 21. Turn at which the robot sometimes get stuck

6 Discussion

The goal is sent to the move_base using the following command.
rosrun udacity_bot navigation_goal
Within few motion updates the particle filter manages to converge to a good estimation of where the robot is. The arrows become sufficiently dense and generally pointing towards one direction indicating that the filter is able to guess with a high degree of accuracy where the robot is and where it is pointing.

The size of the robot did not seem to make any difference, both of the robots took about 55 seconds to compete the plan and both had good point cloud estimations at the end of the navigation process. For both robots, the particle covariance rate was about 20 seconds (the speed at which the point cloud converges).

Here is a list of the problems encountered and how they were solved:

- robot gets stuck at turn:
  set holonomic to false. That didnt solve the problem but made it much less frequent. At the very least, move_base stopped sending y velocities to the robot expecting to get unstack.
- robot pauses at the turn and rotates before continuing to end location. This problem was not solved.
- robot takes an unexpected path and moves up into the map rather than down where the goal is located: Solved by making local costmap smaller. Seems like when the local costmap is big enough, it starts disobeying the global plan.

7 Conclusion / Future work

In the future there are few things that can be done to improve the project and make it more interesting. The first thing that needs to be done is to understand why does the robot get stuck even though it is not physically hitting an obstacle. Why does the move_base sometimes stop sending values to the robot. The robot should be able to navigate sharp turns effectively. The local cost map parameters should be explored more to figure out a solution to the robot getting stuck.
Additionally, in the future, the problem regarding the robot pausing and spinning in place while taking a sharp turn should be fixed. This will also be solved by further exploration of parameters.

In the future, a good test case to test the robustness of the navigation is to divide the grid into sections and have a destination for every section and see if the robot is able to reach from each section to the next. It is a very long test, however, if successful, it shows to a very high degree the robustness of the algorithm.

This should also be tried on different maps. Another map that would be hard for this algorithm to deal with is a map with less walls. The less the walls the more uncertain the point cloud is.

It doesn't seem that computation time was an issue, even with low update frequency the robot managed to reach the destination successfully.

Also, in the future, this algorithm should be tried on a physical robot to see it's effectiveness on them.

8 APPENDIX

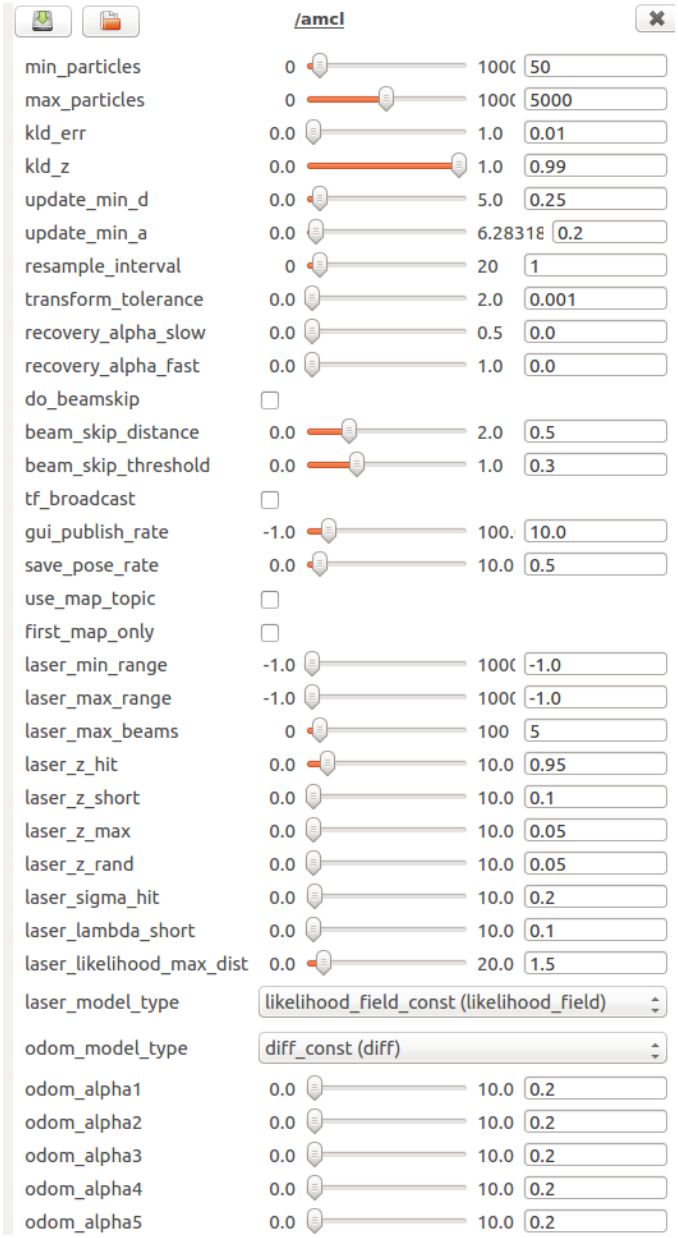

Fig. 22. AMCL parameters in dynamic reconfiguration of params in rqt

REFERENCES

[1] *Udacity* @ url= udacity.com, urldate = 2018
[2] *costmap_2d documentation* url= http://wiki.ros.org/costmap_2d/hydro/inflation, urldate = 2013-04-25, author = David Lu, title = Inflation Costmap Plugin