Overview and Simulation of Adaptive Monte Carlo Localization (AMCL)

Erick Ramirez

Abstract—This paper describes the implementation of a simulation of Adaptive Monte Carlo Localization (AMCL). The objective is to accurately localize two mobile robot inside a provided map using Robot Operating System, Gazebo and RViz simulation environment. I build a robot model with a RGBD camera and a laser sensor.

Index Terms—Kalman Filters, Monte Carlo Localization.

1 Introduction

The task of localization means to determine the pose or a good approximation of the current position of a robot. In this case a mapped environment is mandatory and with the use of sensor data and control information including the noise of it. For this project, a robot model has been developed and tested on a simulation. The robot is using an implementation of Adaptive Monte Carlo Localization (AMCL) to track the position problem. The source code of this project can be found in: https://github.com/mgangster/RoboND-Localization-Project.

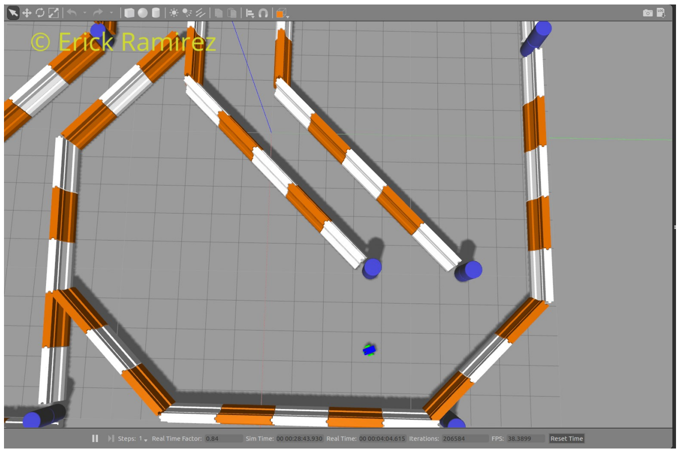


Fig. 1. udacity_bot in the jackal_race.world

2 Background: Localization Algorithms

It is a common task for a mobile robot to determine its current position. In a real implementation of a robotic system, it has to handle with noisy measurements and the actuators are imprecise. The probabilistic models will be useful to address this problems. The two most common approaches to work on this are the following:

2.1 Kalman Filters

The kalman filters algorithm uses linear functions, then it is assuming that the motion model and measurement model are linear. Also, it is assuming that the state space can be presented by a unimodal Gaussian distribution. However, in the real world this assumptions limit the applicability of the Kalman Filter, because most of the robotics systems are nonlinear. There is a nonlinear version of Kalman Filters, it is Extended Kalman Filters (EKF), this implementation linearizes the functions about an estimate of the current mean and covariance, the differentiable functions for the motion and observation model are approximated by their Jacobians. However, the implementation of EKF is more difficult than the linear Kalman Filters.

2.2 Particle Filters

Particle filters allow arbitrary distributions, it represent a belief distribution of a finite set of particles, for instance, an uniform distribution with equal weights(importance) when the initial pose of the robot is unknown. The Monte Carlo Filter is a particle filter implementation that uses Monte Carlo simulation on a distribution of particles to estimate the position and orientation of a robot. The belief, which is the robot's estimate of its current state, is a probability density function distributed over the state space. Basically it is composed by two main sections:

- Motion and sensor update
- Re-sampling process

and the basically actions to do are the following

- Previous belief: the first time the particles are drawn randomly and uniformly over the entire space
- Motion update: the hypothetical state is moved.
- Measurement Update.
- Re-sampling: the particles with hight probability survive and will be re-draw in the next iteration while the others die.
- New belief: output of the cycle, ready for a new iteration.

2.3 Comparison / Contrast

Kalman Filters has specific assumptions that will limit a real world implementation,even if they are easy to implement. it is based on linear functions and unimodal Gaussian

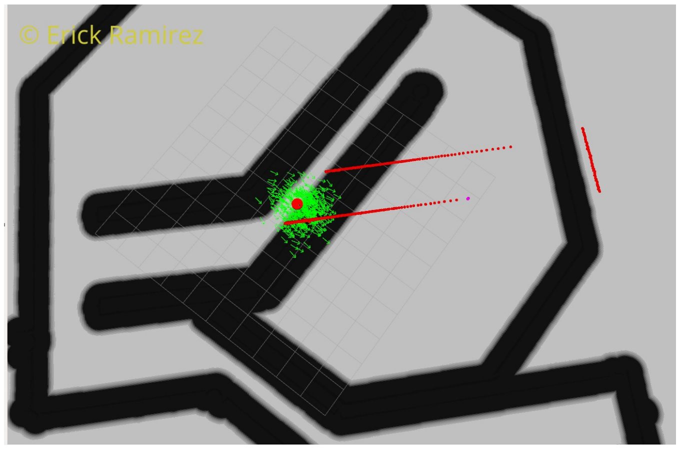


Fig. 2. Initial belief, in this case there is a lot of uncertainty

probability distribution. Extended Kalman filters is able to handle nonlinear functions, however it is converting those function to linear functions and it increase the mathematical complexity. Check the following table about AMCL vs EFK:

TABLE 1
MCL vs EKF

	MCL
Measurements	RAW Measurement
Measurements Noise	Any
Posterior	Particles
Efficiency(memory)	*
Efficiency(time)	*
Easy of implementation	**
Resolution	*
Robustness	**
Memory and Resolution Control	Yes
Global Localization	Yes
State Space	Multi model Discrete

TABLE 2
MCL vs EKF

	EKF
Measurements	Landmarks
Measurements Noise	Gaussian
Posterior	Gaussian
Efficiency(memory)	**
Efficiency(time)	**
Easy of implementation	*
Resolution	**
Robustness	No
Memory and Resolution Control	No
Global Localization	No
State Space	Unimodal Continuous

3 SIMULATION

The simulation has been done in an environment using Gazebo and RViz tools for visualization. It shows the performance of the robots. Check the overview of move_base node provides a ROS interface for configuring, running, and interacting with the navigation stack on a robot:

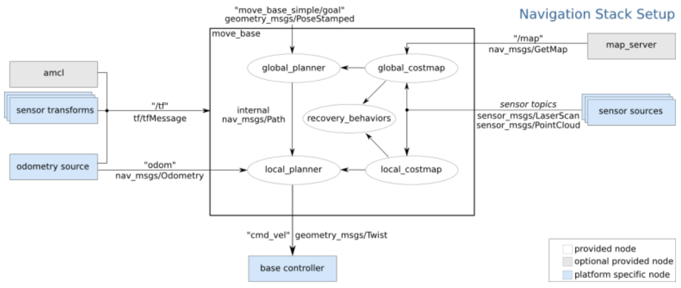


Fig. 3. Initial belief, in this case there is a lot of uncertainty

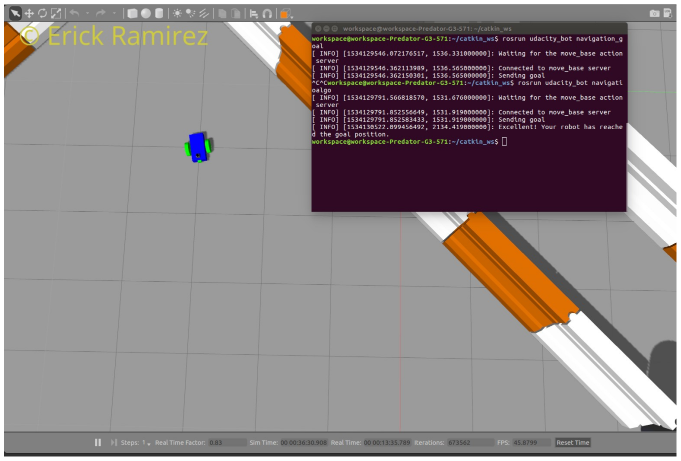
3.1 UdacityBot

At the beginning of the simulation the particles are broadly distributed, which means that there is a lot of uncertainty about the robot position.

Once it start to perform motion update, measurement update and resampling, the particles starts to converge to the real robot's position.

3.2 Achivement

The udacity robot has achieved the end goal. however, it performed an ineffective path to reach it and at the end it keeps in a circuitous route. This is the proof of success, the navigation service indicates that the goal has been reached.



```
workspace@workspace-Predator-G3-571: ~/catkin_ws
workspace@workspace-Predator-G3-571:~/catkin_ws$ rosrun udacity_bot navigation_g
oal
[ INFO] [1534129546.072176517, 1536.331000000]: Waiting for the move_base action
 server
[ INFO] [1534129546.362113989, 1536.565000000]: Connected to move_base server
[ INFO] [1534129546.362150301, 1536.565000000]: Sending goal
^C^Cworkspace@workspace-Predator-G3-571:~/catkin_ws$ rosrun udacity_bot navigati
oalgo
[ INFO] [1534129791.566818570, 1531.676000000]: Waiting for the move_base action
 server
[ INFO] [1534129791.852556649, 1531.919000000]: Connected to move_base server
[ INFO] [1534129791.852583433, 1531.919000000]: Sending goal
[ INFO] [1534130522.099456492, 2134.419000000]: Excellent! Your robot has reache
d the goal position.
workspace@workspace-Predator-G3-571:~/catkin_ws$
```

Fig. 4. udacity_bot achieving the goal

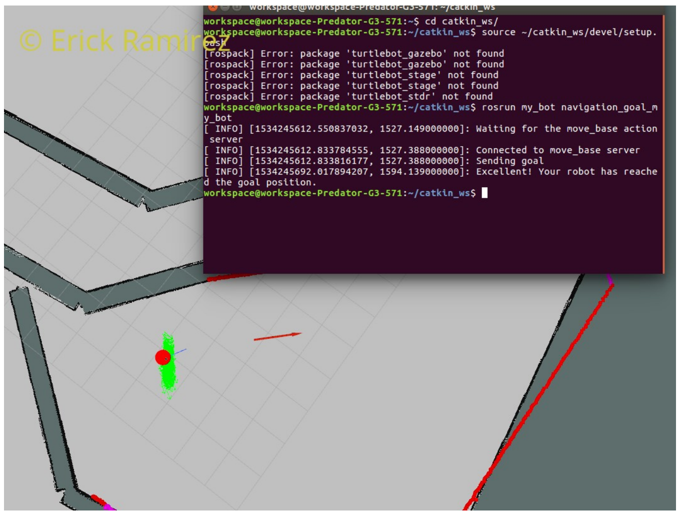
3.3 Benchmark Model

3.3.1 *Model design*

The Robot's design considerations should include: the size of the robot, the layout of sensors.

udacity_bot: The URDF files defines the robot model. udacity_bot.gazebo: it has the definitions of the differential drive controller, the camera, the Hokuyo laser scanner and their controllers for Gazebo. udacity_bot.xacro: it has the robot shape description in macro format.

my_bot: The URDF files defines the robot model. my_bot.gazebo: it has the definitions of the differential drive controller, the camera, the Hokuyo laser scanner and their controllers for Gazebo. my_bot.xacro: it has the robot shape description in macro format.



```
workspace@workspace-Predator-G3-571:~$ cd catkin_ws/
workspace@workspace-Predator-G3-571:~/catkin_ws$ source ~/catkin_ws/devel/setup.
bash
[rospack] Error: package 'turtlebot_gazebo' not found
[rospack] Error: package 'turtlebot_gazebo' not found
[rospack] Error: package 'turtlebot_stage' not found
[rospack] Error: package 'turtlebot_stage' not found
[rospack] Error: package 'turtlebot_stdr' not found
workspace@workspace-Predator-G3-571:~/catkin_ws$ rosrun my_bot navigation_goal_m
y_bot
[ INFO] [1534245612.550837032, 1527.149000000]: Waiting for the move_base action
 server
[ INFO] [1534245612.833784555, 1527.388000000]: Connected to move_base server
[ INFO] [1534245612.833816177, 1527.388000000]: Sending goal
[ INFO] [1534245692.017894207, 1594.139000000]: Excellent! Your robot has reache
d the goal position.
workspace@workspace-Predator-G3-571:~/catkin_ws$
```

Fig. 5. my_bot achieving the goal

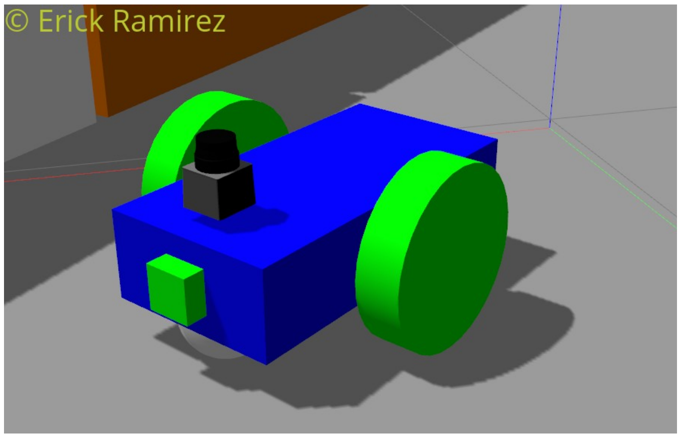
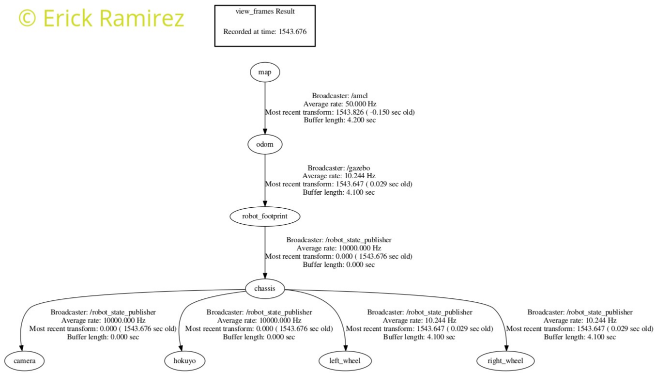


Fig. 6. Udacity_bot model



view_frames Result
Recorded at time: 1543.676
map
Broadcaster: /amcl
Average rate: 50.000 Hz
Most recent transform: 1543.826 (-0.150 sec old)
Buffer length: 4.200 sec
odom
Broadcaster: /gazebo
Average rate: 10.244 Hz
Most recent transform: 1543.647 (0.029 sec old)
Buffer length: 4.100 sec
robot_footprint
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1543.676 sec old)
Buffer length: 0.000 sec
chassis
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1543.676 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1543.676 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.244 Hz
Most recent transform: 1543.647 (0.029 sec old)
Buffer length: 4.100 sec
Broadcaster: /robot_state_publisher
Average rate: 10.244 Hz
Most recent transform: 1543.647 (0.029 sec old)
Buffer length: 4.100 sec
camera
hokuyo
left_wheel
right_wheel

Fig. 7. Transform tree of the udacity_bot model. Output from rosrun tf view_frames

TABLE 3
Model design

	Udacity_bot
chassis: length width x hight	0.4 0.2 x 0.1
chassis: shape	box
wheel radius width	0.1 0.05
wheel separation	0.4
camera: length width x hight	0.05 0.05 x 0.05
hokuyo: length width x hight	0.1 0.1 x 0.1
sensor joint	defined as fix

TABLE 4
Model design

	My_bot
chassis: radius width	0.2 0.1
chassis: shape	cylinder
wheel radius width	0.05 0.05
wheel separation	0.4
camera: length width x hight	0.05 0.05 x 0.05
hokuyo: length width x hight	0.1 0.1 x 0.1
sensor joint	defined as fix

also a third weel has been added and the mass of the elements is lower than the `Udacity_bot` which allows to move faster.

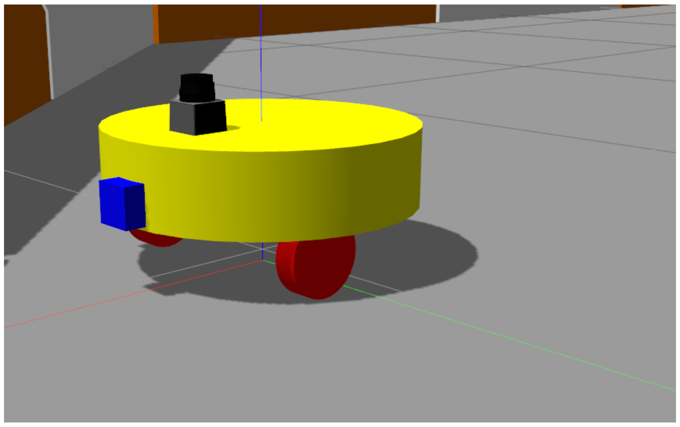


Fig. 8. My_bot model

Meshes: The Hokuyo scanner is defined in the file `hokuyo.dae`, it has the properties of the geometry.

3.3.2 *Map*

The map is defined by the files: `jackal race.yaml` and `jackal race.pgm`. They are necessary for the localization problem, because based on this maps the AMCL algorithm will work.

3.3.3 *Launch*

The launch files specify the parameters to set and nodes to launch. robot_description.launch: defines the robot model(urdf) and it defines the joint state publisher which sends fake joint values, and it send robot states to transform tree:

amcl.launch: launches the AMCL localization server, map server, odometry frame, move_base server and the

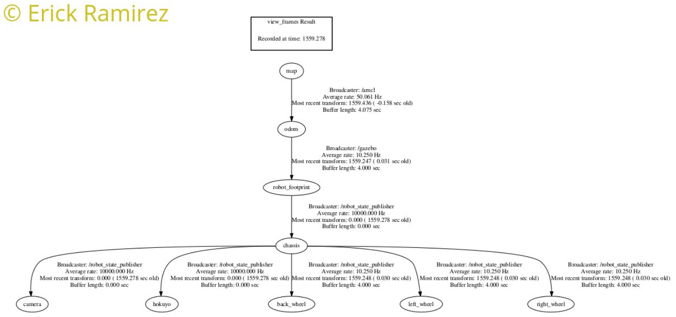

view_frames Result
Recorded at time: 1559.278
map
Broadcaster: /amcl
Average rate: 50.061 Hz
Most recent transform: 1559.436 (-0.158 sec old)
Buffer length: 4.075 sec
odom
Broadcaster: /gazebo
Average rate: 10.250 Hz
Most recent transform: 1559.247 (0.031 sec old)
Buffer length: 4.000 sec
robot_footprint
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1559.278 sec old)
Buffer length: 0.000 sec
chassis
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1559.278 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (1559.278 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.250 Hz
Most recent transform: 1559.248 (0.030 sec old)
Buffer length: 4.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.250 Hz
Most recent transform: 1559.248 (0.030 sec old)
Buffer length: 4.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10.250 Hz
Most recent transform: 1559.248 (0.030 sec old)
Buffer length: 4.000 sec
camera
hokuyo
back_wheel
left_wheel
right_wheel

Fig. 9. Transform tree of the my_bot model. Output from rosrun tf view_frames

trajectory planner server. udacity_world.launch: it launches the `robot_description.launch` file, the gazebo world using `jackal_race.world`, the RViz with the configuration file: `Rvizconfig.rviz` and also it spaws the robot in gazebo world.

3.3.4 *Worlds*

jackal_race.world: defines the maze.

3.3.5 *Packages Used*

The navigation goal service is a C++ program, this is the navigationgoal.cpp file. It is sending the goal and waiting for the result.

3.3.6 *Parameters Udacity_bot*

I used the documentation in: http://wiki.ros.org/amcl. and I sethed the following parameters:

```
#AMCL Parameters
min_particles: 50  # (default: 100)
max_particles: 400  # (default: 5000)
transform_tolerance: 0.2
# (default: 0.1 seconds)
initial_pose_x: 0.0  # (default: 0.0 meters)
initial_pose_y: 0.0  # (default: 0.0 meters)
initial_pose_a: -0.785
#(default: 0.0 radians) -pi/4
laser_model_type: likelihood_field_prob
#(default: likelihood_field)
```

If the number of particles is proportional to computing resources during the resampling, I selected the 50-400 particles because it was sending an confidence. The transform tolerance is the time with which to post-date the transform that is published, to indicate that this transform is valid into the future. It has to be a number greater than the update frequency that I set as 10Hz. The initial. The laser_model_type has been updated to likelihood_field_prob because it is the same as likelihood_field but incorporates the beamskip feature.

```
#Local_costmap_params.yaml
global_frame: odom
update_frequency: 10.0 # (default: 5.0)
publish_frequency: 10.0 # (default: 0.0)
width: 6.0 # (default: 10)
height: 6.0 # (default: 10)
resolution: 0.02 # (default: 0.05)
static_map: false
#(default: 0.0 radians) -pi/4
rolling_window: true #(default: false)
```

```
#global_costmap_params.yaml
global_frame: map
update_frequency: 10.0 # (default: 5.0)
publish_frequency: 10.0 # (default: 0.0)
width: 6.0 # (default: 10)
height: 6.0 # (default: 10)
resolution: 0.02 # (default: 0.05)
static_map: false  #(default: 0.0 radians) -pi/4
rolling_window: true #(default: false)
```

for the files `Local_costmap_params.yaml` and `global_costmap_params.yaml` most of the configuration is the same. The update frequency has been updated to 10Hz and it is related to the transform_tolerance, also, the publish frequency has been updated to 10Hz. The size of the map has been reduced, I got a better result in an smaller map. The resolution has been updated to 0.02 which is the same resolution in the map definition `jackal_race.yaml`.

```
#costmap_common_params.yaml
obstacle_range: 5.0
raytrace_range: 8.0
transform_tolerance: 0.3
inflation_radius: 0.5.5
```

for the file`costmap_common_params.yaml` The obstacle_range parameter determines the maximum range sensor reading that will result in an obstacle being put into the costmap, this number has been increased.Which means that the robot will only update its map with information about obstacles that are within 5 meters of the base. The raytrace_range parameter determines the range to which we will raytrace freespace given a sensor reading. This means that the robot will attempt to clear out space in front of it up to 8.0 meters away given a sensor reading. The inflation radius was set to 0.55 in order to avoid the robot collide with a barrier.

```
#base_local_planner_params.yaml
holonomic_robot: false
yaw_goal_tolerance: 0.05
xy_goal_tolerance: 0.05
controller_frequency: 10
```

The trajectory Planner base local planner params keeps the same goal tolerance, this will mens that the robot perform turn around movements once it is really near of the goal. The controller_frecuency has been updated with the same value as the others frequencies.

3.3.7 *Parameters My_bot*

The main change is the local_costmap, the size of the map is 3x3 meters instead of 6x6 meters, it is because a bigger local map doesn't work for My_bot.

for the `costmap_common_params.yaml` the inflation radios of the costmap has been increased, this in order to avoid a collision.

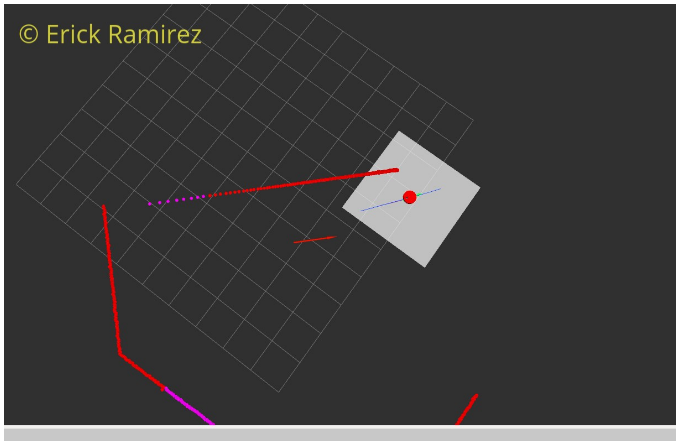
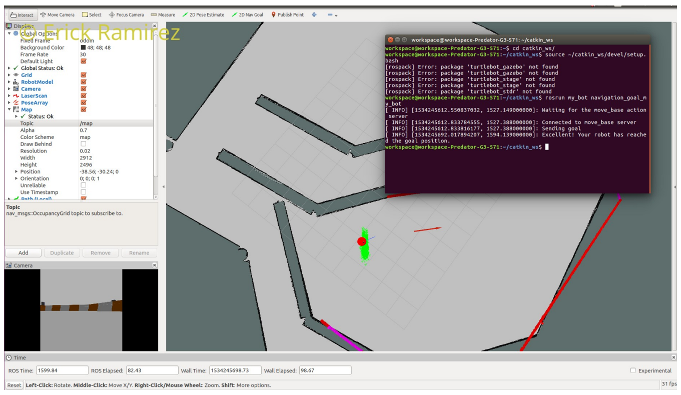


Fig. 10. local map size (global_costmap_params.yaml configuration)

4 Results

Both bots reached the goal position, but My_bot performed it in less time than the Udacity_bot.



```
Fixed Frame        odom
Background Color   48; 48; 48
Frame Rate         30
Topic              /map
Alpha              0.7
Color Scheme       map
Resolution         0.02
Width              2912
Height             2496
Position           -38.56; -30.24; 0
Orientation        0; 0; 0; 1
```

Topic
nav_msgs::OccupancyGrid topic to subscribe to.

```
workspace@workspace-Predator-G3-571: ~/catkin_ws
workspace@workspace-Predator-G3-571:~$ cd catkin_ws/
workspace@workspace-Predator-G3-571:~/catkin_ws$ source ~/catkin_ws/devel/setup.
bash
[rospack] Error: package 'turtlebot_gazebo' not found
[rospack] Error: package 'turtlebot_gazebo' not found
[rospack] Error: package 'turtlebot_stage' not found
[rospack] Error: package 'turtlebot_stage' not found
[rospack] Error: package 'turtlebot_stdr' not found
workspace@workspace-Predator-G3-571:~/catkin_ws$ rosrun my_bot navigation_goal_m
y_bot
[ INFO] [1534245612.550837032, 1527.149000000]: Waiting for the move_base action
 server
[ INFO] [1534245612.833784555, 1527.388000000]: Connected to move_base server
[ INFO] [1534245612.833816177, 1527.388000000]: Sending goal
[ INFO] [1534245692.017894207, 1594.139000000]: Excellent! Your robot has reache
d the goal position.
workspace@workspace-Predator-G3-571:~/catkin_ws$
```

```
ROS Time: 1599.84   ROS Elapsed: 82.43   Wall Time: 1534245698.73   Wall Elapsed: 98.67
```

Fig. 11. my_bot achieving the goal in 82 seconds

4.1 Localization Results

4.1.1 *Benchmark Udacity_bot*

The time taken to reach the goal was 10 minutes, and it never collided with any barrier.

4.1.2 *Benchmark My_bot*

The time taken to reach the goal was 1.5 minutes, and it never collided with any barrier. on each scenario the number of particles is the same.

4.2 Technical Comparison

My_bot is faster than Udacity_bot. My_bot is lighter than Udacity_bot. My_bot has a simplistic design than Udacity_bot. Basically the entire shape is a cylinder.

5 Discussion

Both robots reached the goal, My_bot reached it in less time and it is because that it is a lighter bot (it has less mass than the Udacity_bot). The design of the My_bot is only a cylinder and it doesn have to consider the model shape, which the radious inflation aproach is really good for this bot. The mayor impact is related to the local map size, then it means that more data to consider, doesn't mean a better result. A greater local map size configured on My_bot implied that the robot got stuck. The route taken for each robot was different, for Udacity_bot it takes a big loop and it was not optimal, for My_bot the route was almost optimal because at the end it performed some circuits to reach the goal, in this case the same fact that it has a small local data could affect it.

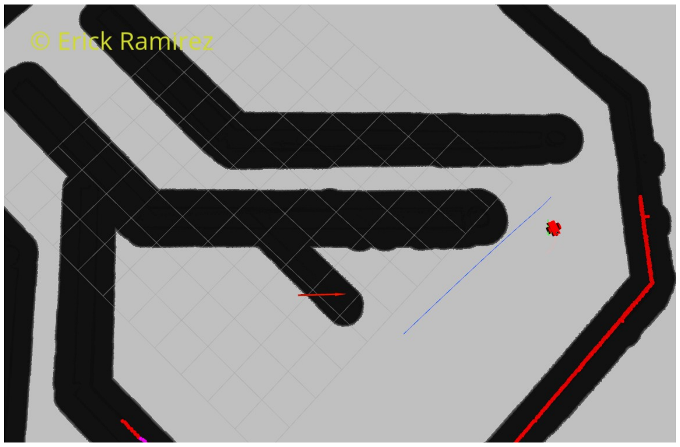


Fig. 12. Udacity_bot navigation and planning

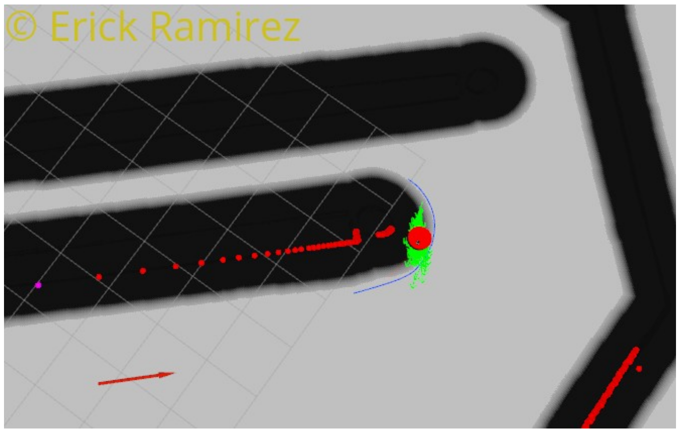


Fig. 13. my_bot navigation and planning

5.1 Topics

- Which robot performed better? My_robot
- Why it performed better? The robot design (shape and mass)
- How would you approach the 'Kidnapped Robot' problem? In this case the AMCL doesn't work well for this problem because of the uncertainty of the map. A better is using SLAM (Simultaneous localization and mapping) .
- What types of scenario could localization be performed? Localization itself would be applied in controlled scenarios when the uncertainty of the map is not a problem.

- Where would you use MCL/AMCL in an industry domain? localization can be applied in many industries like home service robots, on big warehouse to move products, self driving cars, among others.

6 Conclusion / Future work

Both robots were capable to reach the goal position without any collision. This project was focused on localization and some of the errors reached were related to the navigation, because the planning or expected path was right. This was an atypical scenario in real world, because it is completely static and we already had a map to start to work on. In real world applications even indoor environments the expected scenario is to interact with other objects and some of them are moving. Future work on this approach will be to test on other scenarios, and the validation of mapping, planning and navigation. The addition of sensors like Light Detection and Ranging o Laser Imaging Detection and Ranging (LIDAR) or an RGBD camera can help to improve the localization problem or even if we have a 3d map and not only a 2d map.