Overview and Simulation of Adaptive Monte Carlo Localization

Gan Ma

Abstract—This paper describes the creation and testing of two robot simulations in a ROS (Robot Operating System) / Gazebo / RViz simulation environment. Both robots use Adaptive Monte Carlo Localization techniques combined with a navigation plug-in to successfully navigate a maze to reach a predefined goal position. The two robot designs are compared for efficiency in reaching the goal.

Index Terms—Robot, Mobile Robotics, Kalman Filters, Particle Filters, Localization.

1 Introduction

Localization in robotics means determining a good approximation of the current position of a robot given uncertainties of noisy sensors such as a camera or LIDAR (Light Detection and Ranging) and uncertainties due to imperfect actuators moving the robot.

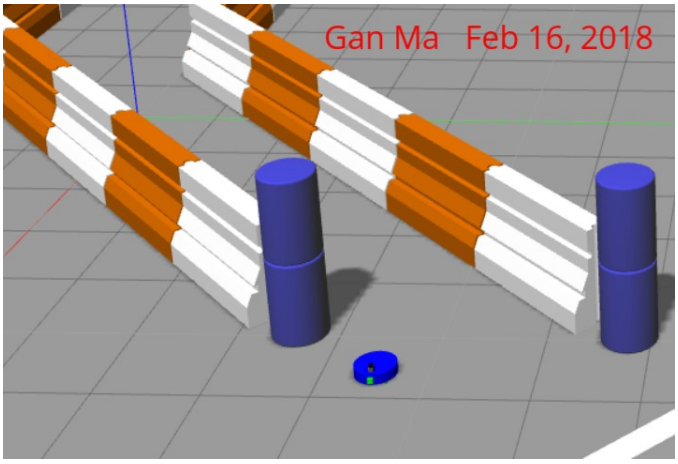

Figure 1. One of the simulation robots traversing the maze.

Two robots were developed and tested in a simulation environment. The robots successfully navigated the maze using the Adaptive Monte Carlo Localization (AMCL) algorithm. The benchmark definition of one of the robots was given as part of the project, while the second was created independently. The benchmark robot is called *'UdacityBot'* and the second robot designed for this project is called *'SweepingBot'* throughout this paper. The world definition uses a map created by *'jackal_race'* was developed by Clearpath Robotics [1]. The source software for this project can be found here: https://github.com/mgangster/RoboND-Localization-Project.

2 Background

Localization is a fundamental issue in developing mobile robots. In an imperfect world where sensors are inaccurate or noisy and actuators are imprecise, localization is central to determining a good estimate of the actual position of a robot.

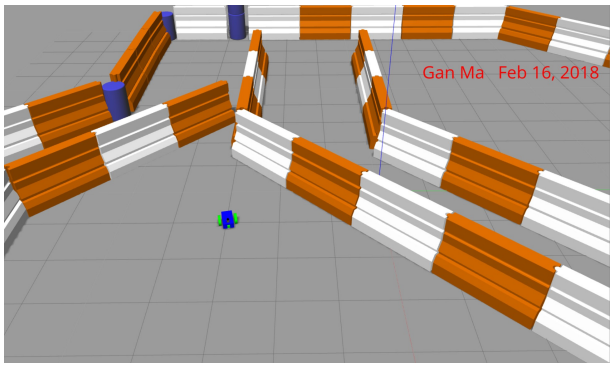

Figure 2. UdacityBot Shown at the Goal Position

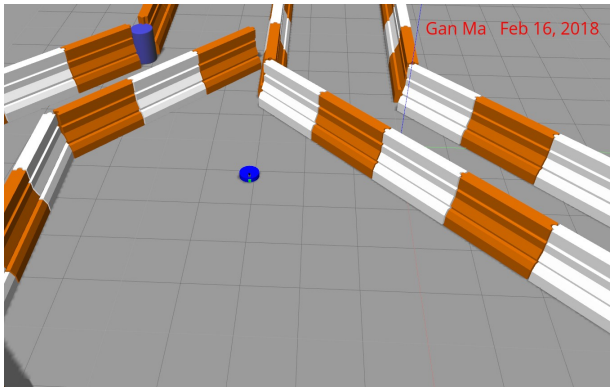

Figure 3. SweepingBot Shown at the Goal Position

The two most common approaches to Localization are Kalman Filters and Monte Carlo Simulation.

2.1 Kalman Filters

The Kalman Filter is very prevalent in control systems. It is very good at taking in noisy measurements in real time and providing accurate predictions of actual variables such

as position. The Kalman Filter algorithm is based on two assumptions:

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

These assumptions limit the applicability of the Kalman Filter as most real robot scenarios do not meet these assumptions.

The Extended Kalman filter (EKF) addresses these limiting assumptions by linearizing a nonlinear motion or measurement function with multiple dimensions using multidimensional Taylor series.

While the EKF algorithm addresses the limitations of the Kalman Filter, the mathematics is relatively complex and the computation uses considerable CPU resources.

2.2 Particle Filters

Particle Filters operate by uniformly distributing particles throughout a map and then removing those particles that least likely represent the current position of the robot.

The Monte Carlo Filter is a particle filter that uses Monte Carlo simulation on an even distribution of particles to determine the most likely position value. Computationally it is much more efficient the Kalman Filters. Monte Carlo localization is not subject to the limiting assumptions of Kalman Filters as outlined above.

The basic MCL algorithm steps are as follows:

1) Particles are drawn randomly and uniformly over the entire space.
2) Measurements are taken and an importance weight is assigned to each particle.
3) Motion is effected and a new particle set with uniform weights and the particles are resampled.
4) Measurement assigns non-uniform weights to the particle set.
5) Motion is effected and a new resampling step is about to start.

In this implementation we use *Adaptive Monte Carlo Localization*. AMCL dynamically adjusts the number of particles over a period of time, as the robot navigates a map.

2.3 Comparison / Contrast

Kalman Filters have limiting assumptions of a unimodal Gaussian probability distribution and linear models of measurement and actuation. 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. As such Monte Carlo Localization is used in this project.

The differences between the approaches can be summarized in a table:

AMCL EKF Comparison		
	MCL	EKF
Measurements	Raw	Landmarks
Measurement Noise	Any	Gaussian
Posterior Belief	Particles	Gaussian
Memory Efficiency	OK	Good
'' Time Efficiency	OK	Good
Ease of Implementation	Good	OK
Resolution	OK	Good
Robustness	Good	Poor
Global Localization	Yes	No
State Space	Multimodal Discrete	Unimodal Continuous

Table 1
AMCL EKF 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 [2] can be visualized as follows:

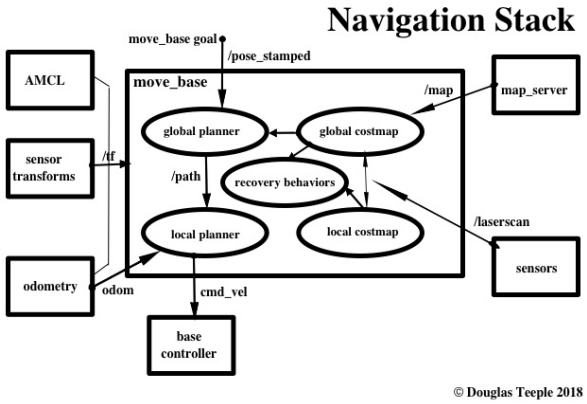

Figure 4. Navigation Stack

3.1 UdacityBot

At the start of the simulation the particles are broadly distributed, indicating great uncertainty in the robot position. At this point the sensors have not yet provided any information as to location. Figure 5 shows the broad distribution of particles (shown as green arrows). After

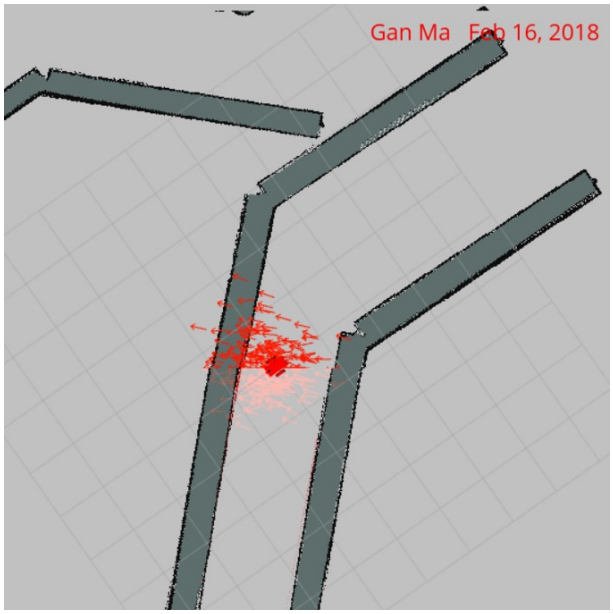

Figure 5. UdacityBot Shown at the Start Position

starting the simulation, sensor measurements are taken and

the localization of the robot improves. Figure 6 shows the narrowing distribution of particles as more sensor data is gathered.

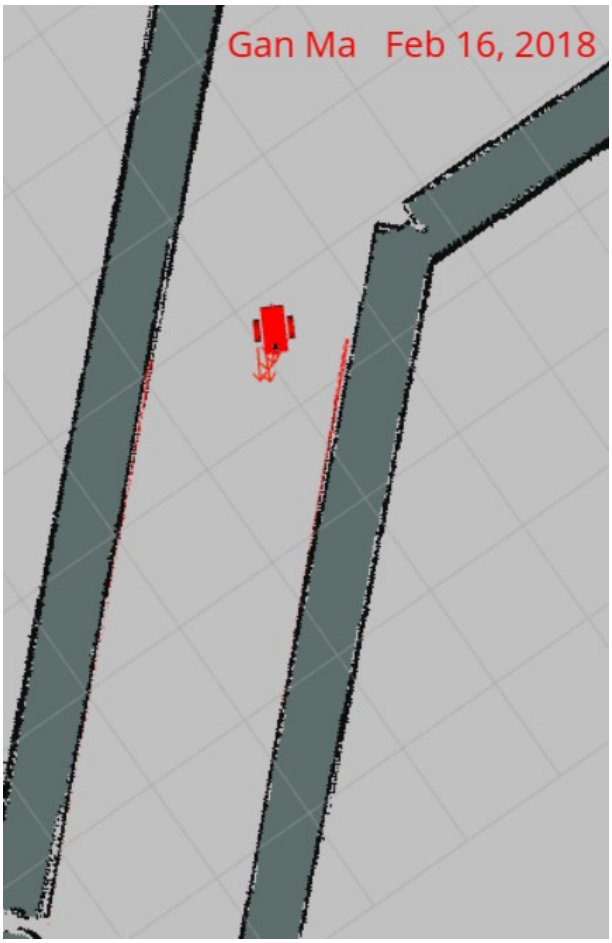

Figure 6. UdacityBot Shown moving along the navigation path

3.2 SweepingBot

At the start of the simulation the particles are broadly distributed, indicating great uncertainty in the robot position. At this point the sensors have not yet provided any information as to location. Figure 7 shows the broad distribution of particles (shown as green arrows). After starting the simulation, sensor measurements are taken and the localization of the robot improves. Figure 8 shows the narrowing distribution of particles as more sensor data is gathered.

3.3 Achievements

Both robots achieved the project requirement of reaching the end goal.

3.3.1 UdacityBot

The UdacityBot correctly reached the goal, though it did so by a somewhat circuitous route, first starting on a path that could not reach the goal (towards to North or topmost of the maze) then turning around and correctly navigating around obstacles to the goal. And the proof of success, the navigation service indicates that the goal has been reached (Figure 10).

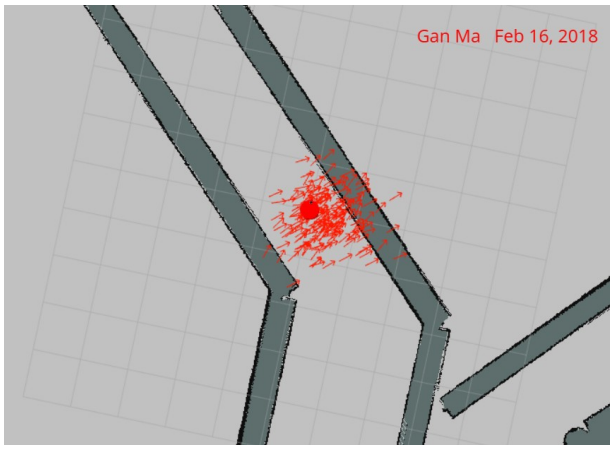

Figure 7. SweepingBot Shown at the Start Position

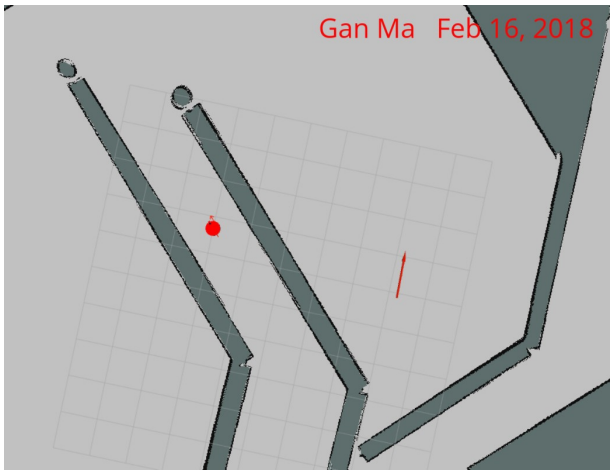

Figure 8. SweepingBot Shown moving along the navigation path

3.3.2 SweepingBot

The SweepingBot also correctly reached the goal, though it did so by the same circuitous route, first starting on a path that could not reach the goal (towards to North or topmost of the maze) then turning around and correctly navigating around obstacles to the goal.

Figure 11 and 12 shows the SweepingBot at the goal position:

And the proof of success: the navigation service indicates that the goal has been reached (Figure 13).

3.4 Benchmark Model - UdacityBot

3.4.1 Model design

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

Maps: The ClearPath [1] *jackal_race.yaml* and *jackal_race.pgm* packages were used to create the maps.

Meshes: The Laser Scanner simulated is a Hokuyo scanner [3]. The *hokuyo.dae* mesh was used to render it. **Launch**: Three launch scripts are used:

1) **robot_description.launch**: defines the *joint_state_publisher* which sends fake joint values, *robot_state_publisher* which sends

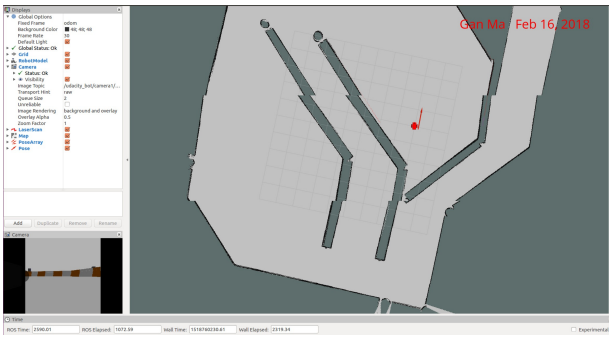

Figure 9. UdacityBot Shown in RViz at the goal

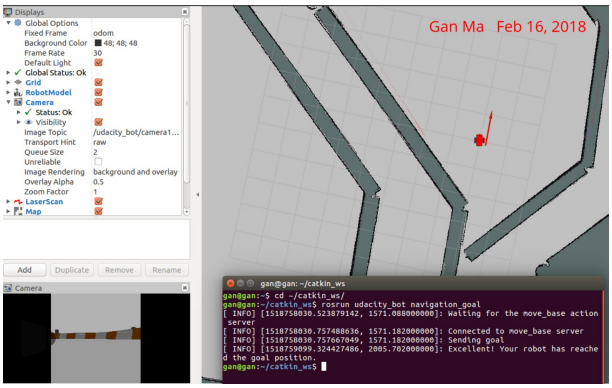

Figure 10. UdacityBot Navigation goal message

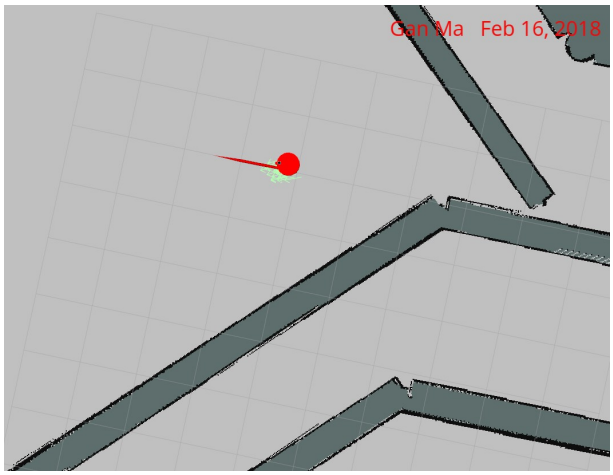

Figure 11. SweepingBot Shown in RViz at the goal

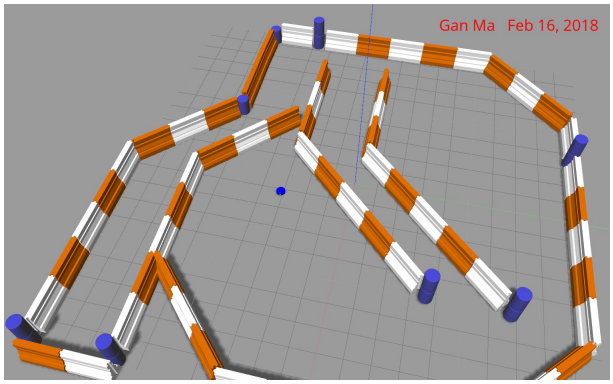

Figure 12. SweepingBot Shown in Gazebo at the Goal

robot states to tf, and *robot_description* which sends the URDF to the parameter server.

2) **amcl.launch**: launches the map server, the odometry frame, the AMCL localization server, the move_base server, and the trajectory planner server.
3) **udacity_world.launch**: includes the robot_description launch file, the gazebo jackal_race world, spawns the robot in gazebo world, and launches RViz.

Worlds: Two worlds are defined:

1) **jackal_race world**: defines the maze,
2) **udacity world**: defines the ground plane, light source and world camera.

URDF: The URDF files defines the shape of the robot. Two files define the Gazebo view and the basic robot description:

1) udacity_bot.gazebo: provides definitions of the differential drive controller, the camera and camera_controller, the Hokuyo laser scanner and controller for Gazebo.
2) udacity_bot.xacro: provides the robot shape description in macro format.

3.4.2 Packages Used

The navigation goal service is a C++ program:

Listing 1. navigationgoal.cpp

```
#include <ros/ros.h>
#include <move_base_msgs/MoveBaseAction.h>
#include <actionlib/client/simple_action_client.h>

typedef actionlib::SimpleActionClient<move_base_msgs::
MoveBaseAction> MoveBaseClient;

int main(int argc, char** argv){
    ros::init(argc, argv, "navigation_goal");
        // Spin a thread
        MoveBaseClient ac("move_base", true);

        // Wait for the action server to come up
        ROS_INFO("Waiting for the move_base action server");
        ac.waitForServer(ros::Duration(5));

        ROS_INFO("Connected to move_base server");

        move_base_msgs::MoveBaseGoal goal;

        // Send goal pose
    goal.target_pose.header.frame_id = "map";
        goal.target_pose.header.stamp = ros::Time::now();

        // Do NOT modify the following for final submission.
        goal.target_pose.pose.position.x = 0.995;
        goal.target_pose.pose.position.y = -2.99;

        goal.target_pose.pose.orientation.x = 0.0;
    goal.target_pose.pose.orientation.y = 0.0;
    goal.target_pose.pose.orientation.z = 0.0;
    goal.target_pose.pose.orientation.w = 1.0;

        ROS_INFO("Sending goal");
        ac.sendGoal(goal);

        ac.waitForResult();

        if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED) {}
        ROS_INFO("Excellent! Your robot has reached the goal position.");
        } else {
        ROS_INFO("The robot failed to reach the goal position");
            return -1;
    }
    return 0;
}
```

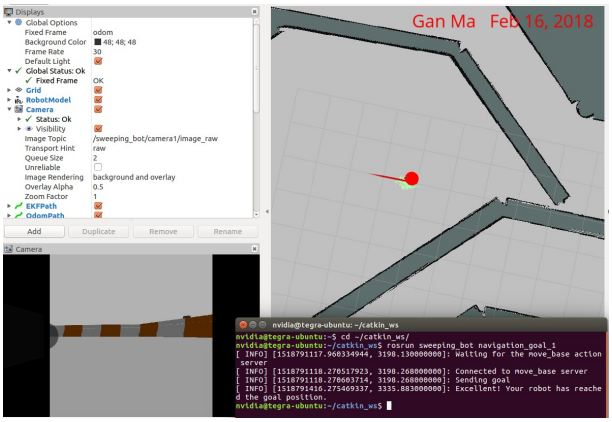

Figure 13. SweepingBot Navigation Goal Message

Table 2
Key udacity_bot xacro Definitions

Key XACRO Definitions		
Name	**Type**	**Value**
footprint joint	Joint	xyz= ”0 0 0” rpy= ”0 0 0”
chassis	Pose	0 0 0.1 0 0 0
chassis	collision / visual	box: .4 .2 .1
back caster	collision / visual	xyz= ”-0.15 0 -0.05” sphere radius= ”0.05”
front caster	collision / visual	xyz= ”0.15 0 -0.05” sphere radius=”0.05”
left wheel	collision / visual	cylinder radius= ”0.1” length= ”0.05”
left wheel hinge	continuous	xyz= ”0 0.15 0” axis xyz= ”0 1 0”
right wheel	collision / visual	cylinder radius= ”0.1” length= ”0.05”
right wheel hinge	continuous	xyz= ”0 -0.15 0” axis xyz= ”0 1 0”
Camera	collision / visual	box size= ”0.05 0.05 0.05”
Camera	joint	origin xyz= ”0.2 0.0 0.0”
Scanner	collision / visual	box size= ”0.1 0.1 0.1”
Scanner	joint	origin xyz= ”0.15 0.0 0.1”

3.4.3 Parameters

Heavy use of the ROS Basic Navigation Tuning Guide [4] and Kaiyu Zheng's ROS Navigation Tuning Guide [5] in setting parameters. Localization parameters in the AMCL node:

```
############################################
#
#  AMCL Parameters
#
############################################
############################################
# Overall filter parameters
############################################
max_particles: 200          # default 5000
min_particles: 20           # default 100
transform_tolerance: 0.2    # (default 0.1s)
initial_pose_x: 0.0         # default
initial_pose_y: 0.0         # default
initial_pose_a: -0.785      # -pi/4
```

The min and max particles parameters were set to a value of 5000, a number too large for the simulation environment on the simulation platform. It was modified down to a minimum of 20 and maximum of 200 particles.

The transform_tolerance is the time with which to postdate the transform that is published, to indicate that this transform is valid into the future. This was a key parameter affecting stability of the model that had to be modified to be larger than the update frequency which was set to 10Hz.

The odom_model_type parameter was set to ”diff-corrected” to correctly model the differential drive of the robot. This parameter indicates which model to use, either ”diff”, ”omni”, ”diff-corrected” or ”omni-corrected”.

The move_base parameters in the configuration file. are:

```
############################################
#
# Cost map common parameters
#
############################################
map_type: costmap
obstacle_range: 5.0 # was 0.0, default 2.5
raytrace_range: 8.0 # was 0.0, default 3.0
############################################
# laser scanner section
############################################
observation_sources: laser_scan_sensor
laser_scan_sensor:   {
   sensor_frame: hokuyo,
   data_type:LaserScan,
   topic: /udacity_bot/laser/scan,
   marking: true,
   clearing: true}
############################################
```

The obstacle_range was modified to have a greater range. The obstacle_range is the default maximum distance from the robot at which an obstacle will be inserted into the cost map in meters. This can be over-ridden on a per sensor basis.

The raytrace_range was modified to have a greater range. The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be over-ridden on a per sensor basis.

The cost maps used are:

```
############################################
# Global cost map
############################################
global_costmap:
global_frame: map
robot_base_frame: robot_footprint
update_frequency: 10.0 # was 50.0
publish_frequency: 10.0 # was 50.0
width: 40.0 # The width of the map in meters.
height: 40.0 # The height of the map in meters.
resolution: 0.02 # The resolution of the map in meters.
static_map: true
rolling_window: false
############################################
# Trajectory Planner local cost map parameters
############################################
local_costmap:
global_frame: odom
robot_base_frame: robot_footprint
update_frequency: 10.0 # was 50.0
publish_frequency: 10.0 # was 50.0
width: 20.0 # The width of the map in meters.
height: 20.0 # The height of the map in meters.
resolution: 0.05 # default 0.05,the resolution in meters.
static_map: false
rolling_window: true
############################################
```

Two parameters in the global and local cost maps were changed: 1) The update_frequency is the frequency in Hz for the map to be updated, this was changed to 10Hz. 2) The publish_frequency is the frequency in Hz for the map to be published on the display, it was set also to 10Hz from the default of 50Hz.

```
############################################
# Trajectory Planner base local planner params
############################################
TrajectoryPlannerROS:
holonomic_robot: false
yaw_goal_tolerance: 0.05
xy_goal_tolerance : 0.05
sim_time: 1.0
meter_scoring: false
pdist_scale: 0.6
controller_frequency: 10.0
############################################
```

The holonomic_robot [1] [6] parameter was left at the default of *false*. Differential wheeled robots are non-holonomic.

The yaw_goal_tolerance is the tolerance in radians for the controller in yaw/rotation when achieving its goal,

1. In classical mechanics a system may be defined as holonomic if all constraints of the system are holonomic. For a constraint to be holonomic it must be expressible as a function: $f(x_1, x_2, x_3, \ldots, x_N, t) = 0$ i.e. a holonomic constraint depends only on the coordinates xjx_j and time t. It does not depend on the velocities or any higher order derivative with respect to t.

the default was left at 0.05 radians. The xy_goal_tolerance is the tolerance in meters for the controller in the x, y distance when achieving a goal. The default of 0.1 meters was tightened to 0.05 meters to achieve the end goal.

The controller_frequency was set 10.0Hz from the default 20.0Hz to match the capabilities of the hardware platform.

3.5 Personal Model - SweepingBot

3.5.1 Model design

SweepingBot has a cylinder on the base with a camera ahead. The laser scanner at the top head of the base.

Maps: The ClearPath *jackal_race.yaml* and *jackal_race.pgm* packages were used to create the maps, identical to UdacityBot. [1]

Meshes: The Laser Scanner simulated is a Hokuyo scanner. The *hokuyo.dae* mesh was used to render it.

Closeup Gazebo view of SweepingBot:

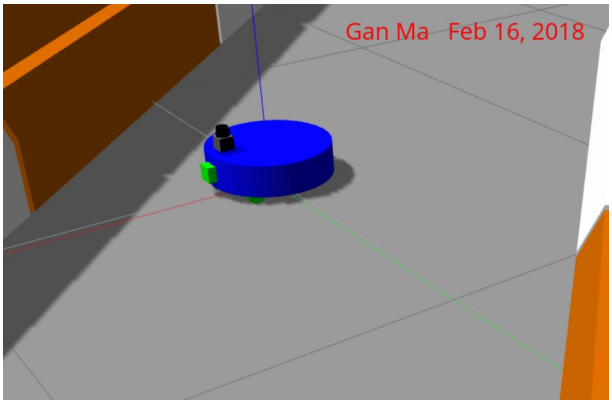

Figure 14. SweepingBot Closeup

Launch: The same three launch scripts are used as UdacityBot.

Worlds: SweepingBot uses the same worlds as UdacityBot.

URDF: The URDF files defines the shape of the robot. Two files define the Gazebo view and the basic robot description:

1) sweeping_bot.gazebo: provides definitions of the differential drive controller, the camera and camera_controller, the Hokuyo laser scanner and controller for Gazebo.
2) sweeping_bot.xacro: provides the robot shape description in macro format. Considerable use of xacro properties was made, to take advantage of multiple uses of a common definition (such as collision and visual definitions) and to take advantage of expressions.

For detailed definitions of the entire xacro definition please see the GIT repository [7].

3.5.2 Packages Used

The same packages are used as UdacityBot.

Table 3
Key sweeping_bot xacro Definitions

Key XACRO Definitions		
Name	**Type**	**Value**
footprint joint	Joint	xyz="0 0 0" rpy="0 0 0"
chassis	collision / visual	cylinder radius="0.2" length="0.1"
left wheel	collision / visual	cylinder length="0.02" radius="0.03"
left wheel hinge	continuous	xyz= "0.12 0.12 -0.05" axis xyz= "0 1 0"
right wheel	collision / visual	cylinder length="0.02" radius="0.03"
right wheel hinge	continuous	xyz= "0.12 -0.12 -0.05" axis xyz= "0 1 0"
back wheel	collision / visual	cylinder length="0.02" radius="0.03"
back wheel hinge	continuous	xyz= "-0.15 0 -0.05" axis xyz= "0 1 0"
Camera	collision / visual	box size= "0.05 0.05 0.05"
Camera	joint	origin xyz= "0.2 0 0"
Scanner	collision / visual	box size= "0.1 0.1 0.1"
Scanner	joint	origin xyz= "0.15 0 0.1"

3.5.3 Parameters

While most of the parameters are the same as UdacityBot, some of them have minor changes. Localization parameters in the AMCL node:

```
############################################
#
#  AMCL Parameters
#
############################################
############################################
# Overall filter parameters
############################################
max_particles: 400         # default 5000
min_particles: 30          # default 100
transform_tolerance: 0.2   # (default 0.1s)
initial_pose_x: 0.0        # default
initial_pose_y: 0.0        # default
initial_pose_a: -0.785     # -pi/4
odom_alpha1: 0.005         # default 0.2
odom_alpha2: 0.005         # default 0.2
odom_alpha3: 0.010         # default 0.2
odom_alpha4: 0.005         # default 0.2
odom_alpha5: 0.003         # default 0.2
```

The min and max particles are increased in order to obtain a precise localization; meanwhile, the values of odom_alpha are reduced.

The move_base parameters in the configuration file are:

```
############################################
#
# Cost map common parameters
#
############################################
map_type: costmap
obstacle_range: 5.0 # was 0.0, default 2.5
raytrace_range: 8.0 # was 0.0, default 3.0
inflation_radius: 0.45 # default 3.0
############################################
# laser scanner section
############################################
observation_sources: laser_scan_sensor
laser_scan_sensor:   {
   sensor_frame: hokuyo,
   data_type:LaserScan,
   topic: /udacity_bot/laser/scan,
   marking: true,
   clearing: true}
############################################
```

Here, the inflation_radius was reduced to avoid the robot collide with a barrier.

4 Results

SweepingBot navigated the same path as UdacityBot. The difference in mass had no effect on time to reach the goal.

4.1 Localization Results

4.1.1 Benchmark - UdacityBot

The time taken to reach the goal was 14 minutes. At no time did the robot collide with a barrier. The maximum number of particles was reduced from 5000 to 200 to meet

computational constraints of the deployment environment (NVidia Jetson TX2).

4.1.2 *Individual - SweepingBot*

The time taken to reach the goal was 25 minutes. At no time did the robot collide with a barrier. The maximum number of particles was reduced from 5000 to 200 to meet computational constraints of the deployment environment (NVidia Jetson TX2).

4.2 Technical Comparison

SweepingBot is considerably lighter than the benchmark UdacityBot. The laser Scanner is placed lower than UdacityBot.

5 Discussion

Both robots performed equally well. SweepingBot is considerably lighter than the benchmark UdacityBot, though the extra mass did not significantly change the time to reach the goal. Perhaps this is due to the fact that friction of the wheels is set to maximum so there is no slippage. This assumption does not reflect real world conditions, where slippage of one wheel may well be quite common, and the difference in mass would then become important.

The Laser Scanner placement modification also had no impact on performance in the given environment as the scanner was still lower than the height of the barriers. Had the barriers been lower or of uneven height SweepingBot may well have collided with a barrier.

The route taken was circuitous and suboptimal. The route is as shown:

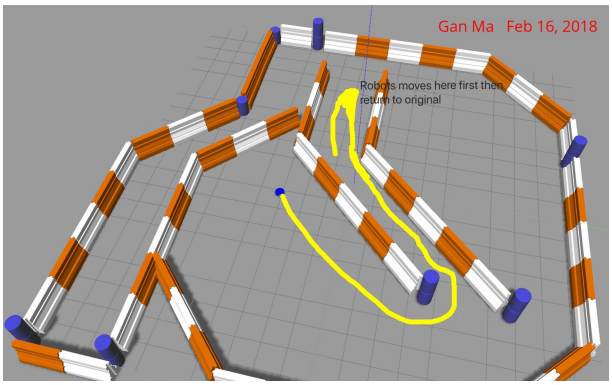

Figure 15. Navigation Route

The turquoise line section shows the initial route taken. Apparently the navigation module was trying to take the shorter route to the goal, but eventually decided that the gap between walls was too narrow to navigate, so turned around and took the longer route to the goal.

5.1 Topics

- Both robots performed performed well.
- AMCL does not work well for the kidnapped robot problem.
- The scenario that needs to be accounted for in the kidnapped robot problem is one of abruptly disappearing from one location and showing up in another.
- MCL/AMCL would work well in any industry domain where clear barriers guide the path of the robot. The ground also needs to be flat and clear of obstacles.

6 Conclusion / Future work

Both robots were equally capable of avoiding collisions with the barriers, indicating that localization was working well. The major contributing factor to the extended time to reach the goal was an error in navigation. This project focused on localization, not navigation.

So, while both robots reached the goal, the circuitous route would mean neither robot model could be applied to commercial products.

It is possible that changing navigation parameters would permit the global planner to "see" further and avoid the backtrack. There was insufficient time to pursue this notion.

Future work to make the robots commercially viable would be in working on improving or better tuning the navigation planner.

The model developed contains only one robot. In the future it would be advantageous to handle multiple robots.

6.1 Hardware Deployment

1) The two project models are deployed on a Jetson TX2 board running ROS and Ubuntu 16.04 Linux.
2) Experience shows that this hardware configuration has adequate processing power both in CPU power and memory to host the model.
3) The models were simulated in Gazebo and RViz only in this project, and no drivers were implemented/integrated to actuate drive motors or read sensors. The TX2 prototype board has a camera which could be connected into the model with suitable drivers. Laser scanner hardware and drivers would have to be integrated in order for a hardware version to operate. It would also need GIO connections to drive wheels and be implemented on a suitable platform modeling the simulation robot.

References

[1] ClearPathRobotics, "Clearpath robotics home page." https://www.clearpathrobotics.com, 2018.
[2] WillowGarage, "Willow garage nav stack image." http://wiki.ros.org/navigation/Tutorials/RobotSetup, 2018.
[3] Hokuyo, "Hokuyo laser scanner home page." https://www.hokuyo-aut.jp, 2018.
[4] ROS.ORG, "Ros navigation tuning guide." http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide, 2018.
[5] K. Zheng, "Ros navigation tuning guide." http://kaiyuzheng.me/documents/navguide.pdf, 2018.
[6] Wikipedia, "Holonomic constraint." https://en.wikipedia.org/wiki/Holonomic_constraints, 2018.
[7] G. Ma, "Github robond localization project." https://github.com/mgangster/RoboND-Localization-Project, 2018.