Project: Where Am I?

Thomas Hatting

Abstract—This paper describes the creation of two robot simulations in the ROS / Gazebo / RViZ simulation environment. Both robots use Adaptive Monte Carlo Localization combined with a navigation plugin to successfully navigate a maze to reach a goal position.

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

1 Introduction

LOCALIZATION in the field of robotics means determining a good estimate of the current position given uncertainties due to noisy sensors, such as a laser rangefinder, and uncertainties due to imperfections in the actuators that move the robot.

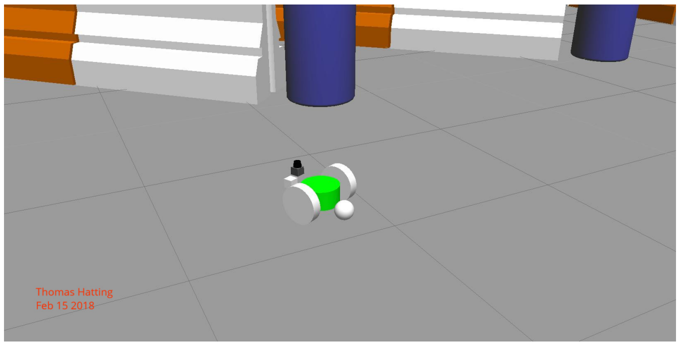


Fig. 1: One of the robots in the simulated environment

Two robots were developed and tested in a simulated environment with the aim of successfully navigating a maze using the Adaptive Monte Carlo Localization (AMCL) algorithm. The definition of the first robot was given as part of the project. The second robot was created independently. In the paper, the benchmark robot is called *UdacityBot* and the second robot is called *T-Bot*. The robots were developed in ROS (Kinetic Distribution), and the simulated environment was created by the Gazebo tool as a physics engine along with RViz as a 3D visualization tool to display sensor data and state information. Figures 1-3 show the two robots in the simulated environment of Gazebo.

2 Background

Localization is a key issue when developing mobile robots. In the real world, sensors are inaccurate and noisy and actuators are imprecise. Therefore, good localization algorithms are crucial in order to determine the position of a robot. In the field of robotics, research has been carried out to create robust, high-accuracy algorithms that are computationally efficient [1], [2].

The two most common approaches to localization are Kalman filters and the Monte Carlo Simulation.

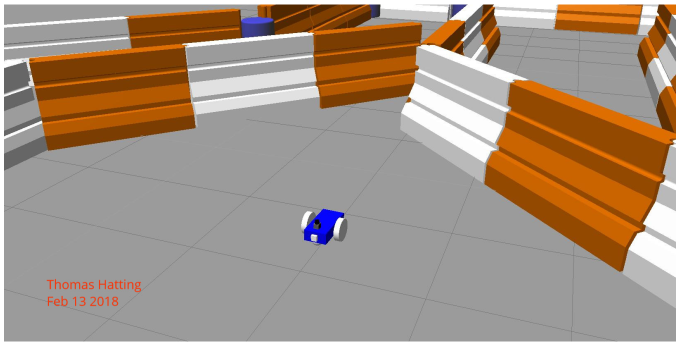


Fig. 2: UdacityBot at the goal position

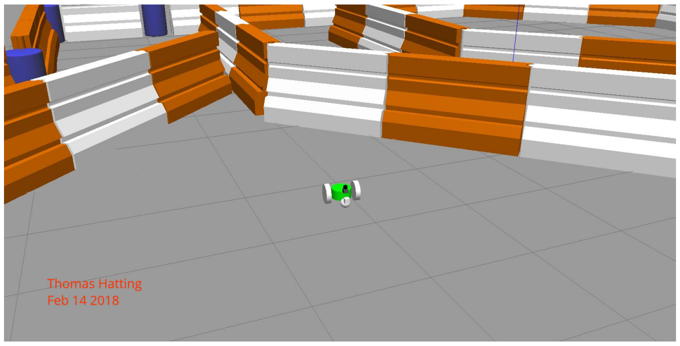


Fig. 3: T-Bot at the goal position

2.1 Kalman Filters

Kalman filtering is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement. The Kalman filter is commonplace in control systems, and the algorithm is good at capturing noisy measurements in real time and providing accurate predictions of variables such as position [3].

The Kalman filter algorithm is based on two assumptions:

- Motion and measurement models are linear
- State space is represented by a Gaussian distribution

In real-life situations, these assumptions limit the application of Kalman filters. Instead, the Extended Kalman filter (EKF) addresses these limitations by ”linearizing” nonlinear motion through the use of multi-dimensional Taylor series [4]. While the EKF algorithm addresses limitations of the Kalman filter, the mathematics is relatively complex and consumes considerable CPU resources [1].

2.2 Monte Carlo Localization

Particle filters operate by uniformly distributing particles throughout a map and then removing particles least likely to represent the current position of the robot. The Monte Carlo filter is a particle filter using the Monte Carlo Simulation on an even distribution of particles in order to determine the most likely position of a robot. It is much more computationally efficient than the Kalman filter. Also, Monte Carlo localization is not subject to the limiting assumptions of the Kalman filter mentioned previously.

The steps in the MCL algorithm are as follows [1]: 1. Move the robot and measure its distance to a set of designated landmarks 2. Simulate noise and add to the measurements 3. Randomly and uniformly spread particles throughout the map 4. Assign a non-uniform weight to each particle then re-sample the particles based on these weights 5. Generate an error value to check quality of the solution and determine how close the particles are to the robot.

2.3 Comparison

The Kalman filter has the limiting assumption of a Gaussian probability distribution and linear model of measurement. The Extended Kalman filter relaxes these assumptions but at the cost of increased mathematical complexity and greater CPU resource requirements. The most significant advantages of MCL over EKF can be summarized as follows [1]:

- You cannot always model real-world problems with linear, Gaussian distributions. MCL is not restricted by Gaussian assumptions and can therefore model a greater variety of environments.
- The MCL algorithm is easier to implement with less mathematical complexity compared to EKF.
- In the MCL algorithm, you can control memory and resolution by changing the number of particles. MCL is therefore more computational efficient than EKF.

In this project, an improved version of the algorithm called Adaptive Monte Carlo Localization (AMCL) is used. This algorithm dynamically adjusts the number of particles, as the robot moves around the map. This adaptive process provides a significant computational advantage over MCL.

3 Simulations

The simulations below describe the performance of the system.

3.0.1 *UdacityBot*

At the start of the simulation, the particles were scattered indicating strong uncertainty in the robot's position. The sensors had not yet been provided with any information to determine location. Figure 4 shows the distribution of particles for UdacityBot at this position. As the simulation progressed, sensor measurements were taken and the localization began to improve. In figure 5, the narrowing distribution of particles is shown, as the robot moves along the navigation path. The particles (or PoseArray) appear more narrowly distributed and centered around the robot with a significant portion of particles pointing in the right direction.

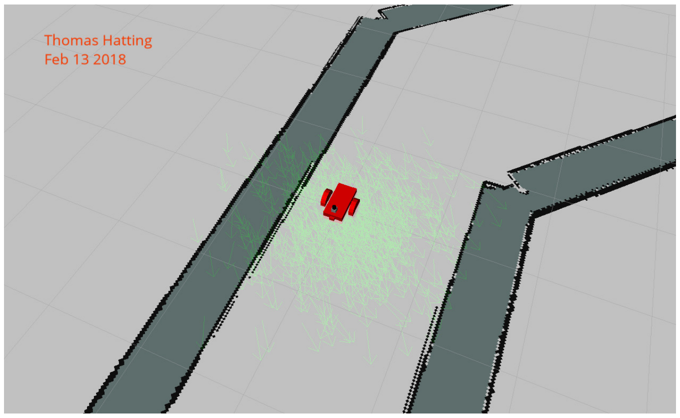


Fig. 4: UdacityBot at start position

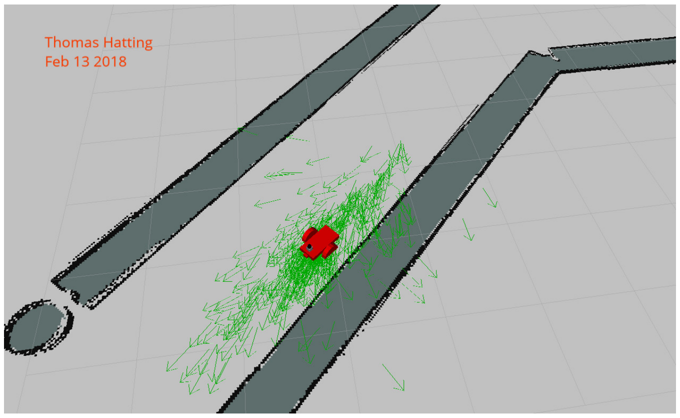


Fig. 5: UdacityBot moving along navigation path

3.0.2 *T-Bot*

In a similar way, at the start of the simulation, the particles were broadly distributed indicating strong uncertainty in the robot's position. At this point the sensors had not yet been provided with any information. The robot in this position is shown in figure 6. As the simulation progressed, the localization improved. Figure 7 shows the narrowing distribution of particles of the T-bot, as it moves towards the end of the corridor.

3.1 Achievements

Both robots achieved the project requirement of reaching the goal position. The goal was achieved by both robots in 20-30 minutes, depending on the simulation.

3.1.1 *UdacityBot*

The UdacityBot correctly reached the goal. The robot spent a significant amount of time in the corridor navigating in a somewhat circular way. Eventually, the robot found its way out of the corridor and moved into the open area on

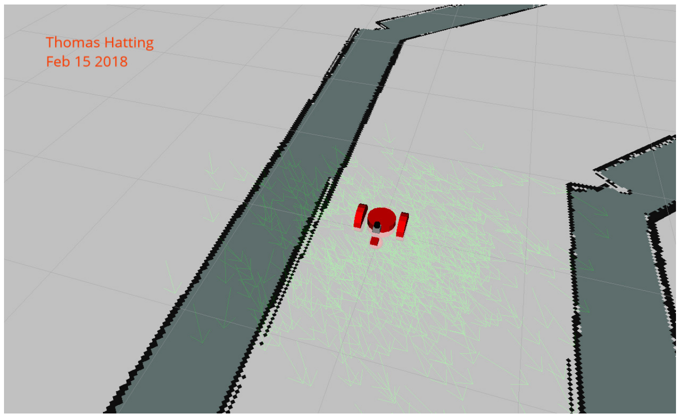


Fig. 6: T-Bot at start position

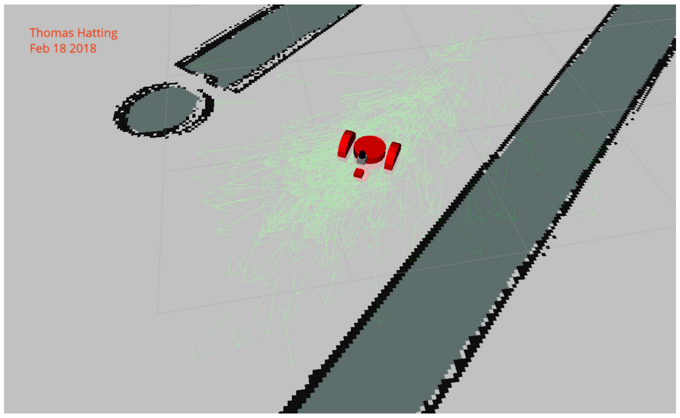
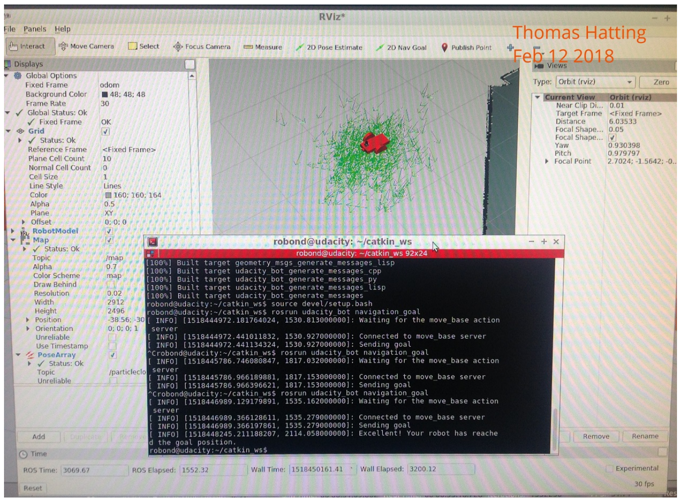


Fig. 7: T-Bot along navigation path



```
robond@udacity: ~/catkin_ws 92x24
[100%] Built target geometry_msgs_generate_messages_lisp
[100%] Built target udacity_bot_generate_messages_cpp
[100%] Built target udacity_bot_generate_messages_py
[100%] Built target udacity_bot_generate_messages_lisp
[100%] Built target udacity_bot_generate_messages
robond@udacity:~/catkin_ws$ source devel/setup.bash
robond@udacity:~/catkin_ws$ rosrun udacity_bot navigation_goal
[ INFO] [1518444972.181764024, 1530.813000000]: Waiting for the move_base action
 server
[ INFO] [1518444972.441011832, 1530.927000000]: Connected to move_base server
[ INFO] [1518444972.441134324, 1530.927000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun udacity_bot navigation_goal
[ INFO] [1518445786.746080847, 1817.032000000]: Waiting for the move_base action
 server
[ INFO] [1518445786.966189881, 1817.153000000]: Connected to move_base server
[ INFO] [1518445786.966396621, 1817.153000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun udacity_bot navigation_goal
[ INFO] [1518446989.129179891, 1535.162000000]: Waiting for the move_base action
 server
[ INFO] [1518446989.366128611, 1535.279000000]: Connected to move_base server
[ INFO] [1518446989.366197861, 1535.279000000]: Sending goal
[ INFO] [1518448245.211188207, 2114.058000000]: Excellent! Your robot has reache
d the goal position.
robond@udacity:~/catkin_ws$
```

Fig. 9: UdacityBot with navigation goal message

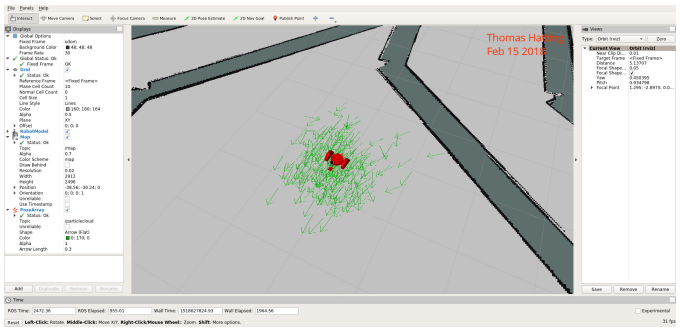


Fig. 10: T-Bot at the goal in RViz

the left-hand side of the map. It spent some time moving around this area until the goal was finally found. Figure 8 shows the UdacityBot at goal position in RViz. Also, proof

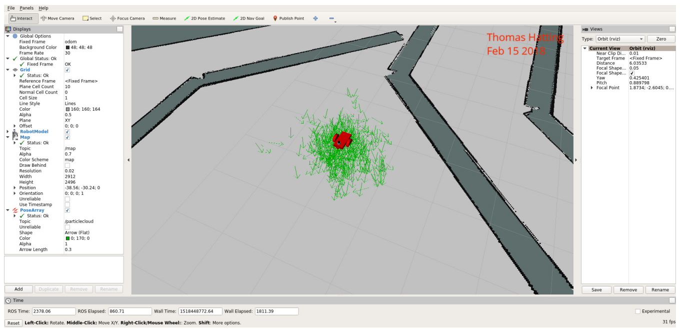


Fig. 8: UdacityBot at the goal in RViz

of success is shown in figure 9, where the terminal output indicates that the goal has been reached.

3.1.2 T-Bot

The T-bot behaved in a similar way, spending time moving around the corridor then onto the open area until the robot eventually found the goal. In figure 10, the T-bot is shown at goal position in RViz. The terminal output is show in figure 11.

3.2 Benchmark Model - UdacityBot

3.2.1 Model design

The table below summarizes the robot's design contained in the URDF file:

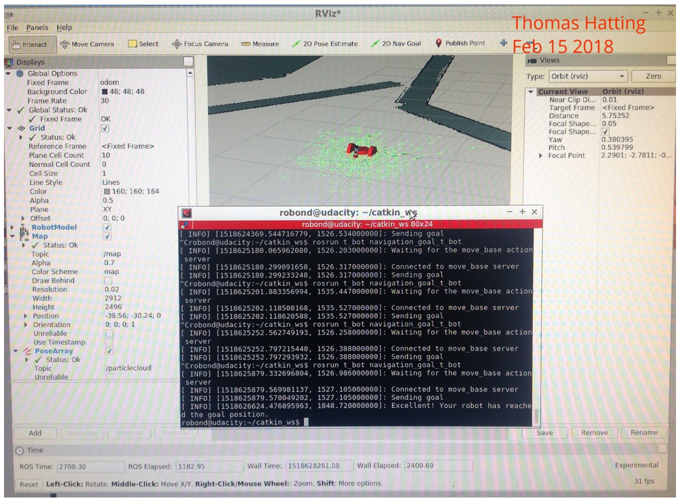
TABLE 1: Design of the UdacityBot (sizes measured in meters and weight in kilograms)

Link or Joint	Origin	Comment
Chassis (link)	(0, 0, 0)	box size 0.4x0.2x0.1, mass 15.0
Back caster (link)	(-0.15, 0, -0.05)	sphere radius 0.05
Front caster (link)	(0.15, 0, -0.05)	sphere radius 0.05
Left wheel (link)	(0, 0, 0)	cylinder radius 0.1, length 0.05, mass 5.0
Right wheel (link)	(0, 0, 0)	cylinder radius 0.1, length 0.05, mass 5.0
Left wheel hinge (joint)	(0, 0.15, 0)	parent link: chassis; child link: left wheel
Right wheel hinge (joint)	(0, -0.15, 0)	parent link: chassis; child link: right wheel
Camera (link)	(0, 0, 0)	box size 0.05x0.05x0.05, mass 0.1
Camera (joint)	(0.2, 0, 0)	parent link: chassis; child link: camera
Hokuyo (link)	(0, 0, 0)	box size 0.1x0.1x0.1, mass 0.1
Hokuyo (joint)	(0.15, 0, 0.1)	parent link: chassis; child link: hokuyo

The robot consists of a rectangular chassis with a pair of wheels and stabilized by front and back casters. Two sensors are mounted: a camera at the front and a Hokuyo laser rangefinder elevated slightly above the chassis.

3.2.2 Packages Used

The main ROS packages were AMCL and move_base. The laser rangefinder published its information to the topic



```
Global Options
  Fixed Frame          odom
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
  Status: Ok
  Reference Frame      <Fixed Frame>
  Plane Cell Count     10
  Normal Cell Count    0
  Cell Size            1
  Line Style           Lines
  Color                160; 160; 164
  Alpha                0.5
  Plane                XY
  Offset               0; 0; 0
RobotModel             ✓
Map                    ✓
  Status: Ok
  Topic                /map
  Alpha                0.7
  Color Scheme         map
  Draw Behind
  Resolution           0.02
  Width                2912
  Height               2496
  Position             -38.56; -30.24; 0
  Orientation          0; 0; 0; 1
  Unreliable
  Use Timestamp
PoseArray              ✓
  Status: Ok
  Topic                /particlecloud
  Unreliable
```

```
Type: Orbit (rviz)            Zero
Current View        Orbit (rviz)
  Near Clip Di...   0.01
  Target Frame      <Fixed Frame>
  Distance          5.75352
  Focal Shape...    0.05
  Focal Shape...    ✓
  Yaw               0.380395
  Pitch             0.539799
  Focal Point       2.2901; -2.7811; -0...
```

```
robond@udacity: ~/catkin_ws
robond@udacity: ~/catkin_ws 80x24
[ INFO] [1518624369.544716779, 1526.534000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun t_bot navigation_goal_t_bot
[ INFO] [1518625180.065962080, 1526.203000000]: Waiting for the move_base action
 server
[ INFO] [1518625180.299091658, 1526.317000000]: Connected to move_base server
[ INFO] [1518625180.299233248, 1526.317000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun t_bot navigation_goal_t_bot
[ INFO] [1518625201.883356994, 1535.447000000]: Waiting for the move_base action
 server
[ INFO] [1518625202.118508168, 1535.527000000]: Connected to move_base server
[ INFO] [1518625202.118620588, 1535.527000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun t_bot navigation_goal_t_bot
[ INFO] [1518625252.562749193, 1526.258000000]: Waiting for the move_base action
 server
[ INFO] [1518625252.797215440, 1526.388000000]: Connected to move_base server
[ INFO] [1518625252.797293932, 1526.388000000]: Sending goal
^Crobond@udacity:~/catkin_ws$ rosrun t_bot navigation_goal_t_bot
[ INFO] [1518625879.332696804, 1526.986000000]: Waiting for the move_base action
 server
[ INFO] [1518625879.569981137, 1527.105000000]: Connected to move_base server
[ INFO] [1518625879.570049202, 1527.105000000]: Sending goal
[ INFO] [1518626624.476895963, 1848.720000000]: Excellent! Your robot has reache
d the goal position.
robond@udacity:~/catkin_ws$
```

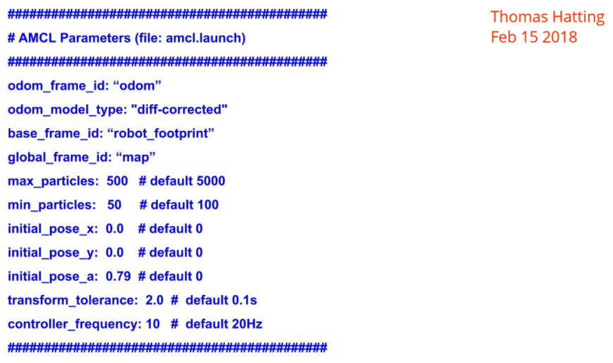
```
ROS Time: 2700.30   ROS Elapsed: 1182.95   Wall Time: 1518628261.08   Wall Elapsed: 2400.69
Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click/Mouse Wheel:: Zoom. Shift: More options.
```

Fig. 11: T-Bot with navigation goal message

/udacity_bot/laser/scan. The camera published its information to the topic *image_raw*. Significant ROS nodes in the system were */joint_state_publisher* and */robot_state_publisher*. The first node published the joint state message such as angles of non-fixed joints. The second node published 3D poses of all joint and links via the robot_state_publisher package.

3.2.3 Parameters

The localization parameters for AMCL are show in the figure below.

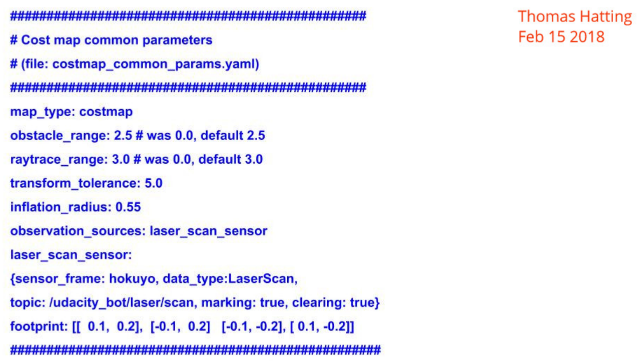


```
###############################################
# AMCL Parameters (file: amcl.launch)
###############################################
odom_frame_id: "odom"
odom_model_type: "diff-corrected"
base_frame_id: "robot_footprint"
global_frame_id: "map"
max_particles:  500   # default 5000
min_particles:   50    # default 100
initial_pose_x:  0.0    # default 0
initial_pose_y:  0.0    # default 0
initial_pose_a:  0.79  # default 0
transform_tolerance:  2.0  #  default 0.1s
controller_frequency: 10   #  default 20Hz
###############################################
```

Fig. 12: AMCL parameters for UdacityBot

The parameter odom_model_type was set to "diff-corrected", as the robot used differential drive. The default values for max_particles and min_particles (particles in the AMCL algorithm) were 5000 and 100 respectively. These values were too large for the simulation platform (a Dell laptop). The parameters were reduced to 500 and 50 respectively to match the hardware platform.

The parameters inital_pose_x, initial_pose_y and initial_pose_a were set to 0.0 , 0.0 and 0.79 respectively. These values placed the robot in the middle of the corridor on the map and turned at an angle of 45 degrees. The transform_tolerance is the time given to post-date the published transform, indicating that this transform is valid into the future. This was a key parameter which affected the stability of the model. It was increased to 2.0 to improve stability. The controller_frequency was reduced to 10 Hz (from a default of 20 Hz) to match the capabilities of the hardware platform and to reduce the likelihood of the simulation getting stuck (PS: at times, the robot got stuck with warning messages appearing in the terminal) [5]. The controller_frequency was modified by setting the parameter in the move_base node and is therefore included in the launch file containing the AMCL parameters.

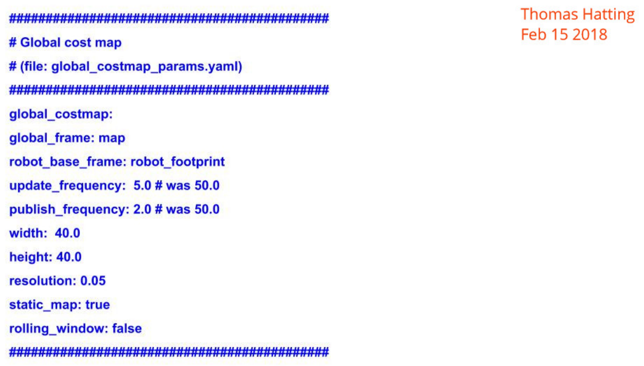


```
###############################################
# Cost map common parameters
# (file: costmap_common_params.yaml)
###############################################
map_type: costmap
obstacle_range: 2.5 # was 0.0, default 2.5
raytrace_range: 3.0 # was 0.0, default 3.0
transform_tolerance: 5.0
inflation_radius: 0.55
observation_sources: laser_scan_sensor
laser_scan_sensor:
{sensor_frame: hokuyo, data_type:LaserScan,
topic: /udacity_bot/laser/scan, marking: true, clearing: true}
footprint: [[  0.1,  0.2],  [-0.1,  0.2]   [-0.1, -0.2], [ 0.1, -0.2]]
###############################################
```

Fig. 13: Common costmap parameters for UdacityBot

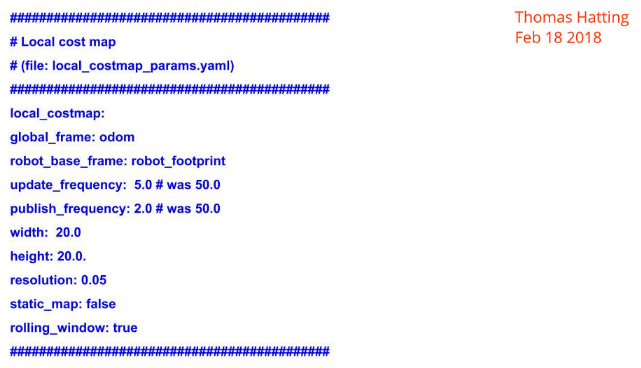
The common costmap parameters are shown in figure 13. The obstacle_range was increased to 2.5 . The obstacle_range is the maximum distance from the robot at which an obstacle will be inserted into the cost map (measured in meters). The raytrace_range was increased to 3.0. The ray_trace parameter determines the range at which the robot will ray-trace free space for a given sensor reading. Setting the parameter to 3.0 meters meant the robot would attempt to clear out space up to 3.0 meters in front [6].

To ensure that the model was stable, the transform_tolerance was increased to 5.0 (relatively high value). The inflation_radius was set to 0.55. The inflation_radius is the maximum distance from obstacles at which a cost should be incurred. Setting the inflation_radius to 0.55 meters meant that the robot would treat all paths 0.55 meters or further away as having an equal obstacle cost [6]. The parameter laser_scan_sensor was configured with characteristics of the Hokuyo laser rangefinder. Finally, the footprint was configured corresponding to a rectangle of length 0.4m and width 0.2m, matching the way the robot's chassis had been configured in the URDF file.



```
###############################################
# Global cost map
# (file: global_costmap_params.yaml)
###############################################
global_costmap:
global_frame: map
robot_base_frame: robot_footprint
update_frequency:  5.0 # was 50.0
publish_frequency: 2.0 # was 50.0
width:  40.0
height: 40.0
resolution: 0.05
static_map: true
rolling_window: false
###############################################
```

Fig. 14: Global cost parameters for UdacityBot



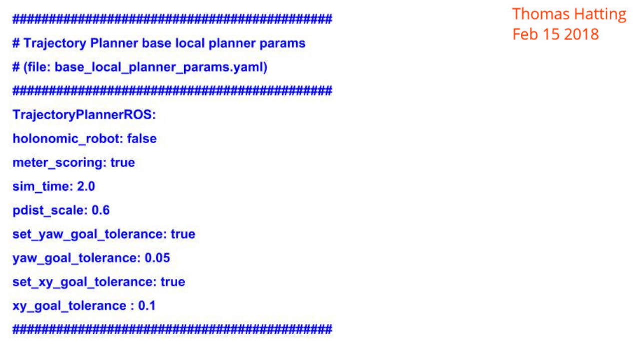
```
###############################################
# Local cost map
# (file: local_costmap_params.yaml)
###############################################
local_costmap:
global_frame: odom
robot_base_frame: robot_footprint
update_frequency:  5.0 # was 50.0
publish_frequency: 2.0 # was 50.0
width:  20.0
height: 20.0.
resolution: 0.05
static_map: false
rolling_window: true
###############################################
```

Thomas Hatting
Feb 18 2018

Fig. 15: Local cost parameters for UdacityBot

The global and local costmaps are shown in figures 14 and 15. The robot_base_frame parameter defines the coordinate framework which the costmap should reference. This parameter was set to ”robot_footprint”. The update_frequency was reduced to a value of 5.0 for both the global and local costmaps. The update_frequency determines the frequency, measured in Hz, at which the costmap runs the update loop. The publish_frequency was reduced to 2.0 for both the global and local costmaps. The publish_frequency determines the rate, measured in Hz, at which the costmap publishes visualization information [6].

The width, height and resolution parameters determines the map dimensions and were set accordingly for global and local costmaps. The static_map parameter determines whether the costmap should be initialized based on a map or map server. The parameter was set to ”true” for the global costmap and ”false” for the local costmap. The rolling_window parameter determines whether the costmap should remain centered around the robot as it navigates around. It was set to ”false” for the global costmap and ”true” for the local costmap.



```
###############################################
# Trajectory Planner base local planner params
# (file: base_local_planner_params.yaml)
###############################################
TrajectoryPlannerROS:
holonomic_robot: false
meter_scoring: true
sim_time: 2.0
pdist_scale: 0.6
set_yaw_goal_tolerance: true
yaw_goal_tolerance: 0.05
set_xy_goal_tolerance: true
xy_goal_tolerance : 0.1
###############################################
```

Thomas Hatting
Feb 15 2018

Fig. 16: Base local planner parameters for UdacityBot

The base local planner parameters are shown in figure 16. The parameter holonomic_robot parameter was set to ”false” (default value). For holonomic robots, strafing velocity commands may be issued to the base. For non-holonomic robots, no strafing velocity commands are issued. The parameter meter_scoring was set to ”true”. It was set to this value to make the settings more robust to changes in costmap resolution (PS: this appeared as a warning in the terminal when the parameter was set to ”false”).

The sim_time was increased to 2.0. The parameter indicates the amount of time to forward-simulate trajectories, measured in seconds. To set it to a higher value can result in slightly smoother trajectories [7]. The parameter pdist_scale was kept at a default value of 0.6. This parameter indicates the weight assigned to how much the controller should stay close to a given path. The yaw_goal_tolerance is the tolerance, measured in radians, for the controller in yaw rotation when the robot is about to reach the goal. It was kept at a default value of 0.05 radians. The xy_goal_tolerance is the tolerance, measured in meters, for the controller in the x- and y-directions when the robot is about to reach the goal. It was kept at a default value of 0.1.

3.3 Personal Model - T-Bot

3.3.1 Model design

The table below summarizes the design of the T-Bot contained in the URDF file:

TABLE 2: Design of the T-Bot

Link or Joint	Origin	Comment
Chassis (link)	(0, 0, 0)	cylinder length 0.1, radius 0.1, mass 15.0
Back caster (link)	(-0.15, 0, -0.05)	sphere radius 0.05
Front caster (link)	(0.15, 0, -0.05)	sphere radius 0.05
Left wheel (link)	(0, 0, 0)	cylinder radius 0.1, length 0.05, mass 5.0
Right wheel (link)	(0, 0, 0)	cylinder radius 0.1, length 0.05, mass 5.0
Left wheel hinge (joint)	(0, 0.15, 0)	parent link: chassis; child link: left wheel
Right wheel hinge (joint)	(0, -0.15, 0)	parent link: chassis; child link: right wheel
Camera (link)	(0, 0, 0)	box size 0.05x0.05x0.05, mass 0.1
Camera (joint)	(0.2, 0, 0)	parent link: chassis; child link: camera
Hokuyo (link)	(0, 0, 0)	box size 0.1x0.1x0.1, mass 0.1
Hokuyo (joint)	(0.15, 0, 0.1)	parent link: chassis; child link: hokuyo

Various designs were considered. Initially, a spherical chassis was chosen. However, during the simulation, the spherical robot proved unstable and tipped over at times. The spherical chassis was replaced by a cylindrical chassis, as indicated in the table. Two sensors were mounted: a camera and laser rangefinder. The rangefinder was initially placed at the center of the cylinder (coordinates: 0, 0, 0.1), however it was moved slightly to the front of the robot (coordinates: 0.15, 0, 0.1) in an attempt to improve the results.

3.3.2 Packages Used

Just like the UdacityBot, the main ROS packages of the T-Bot were AMCL and move_base. The Hokuyo laser rangefinder published its information on topic */t_bot/laser/scan*, and the camera published its information on topic *image_raw*. The most significant ROS nodes were */joint_state_publisher* and */robot_state_publisher*. The first node published the joint state message such as angles of the non-fixed joints. The second published 3D poses of all joint and links via the robot_state_publisher package.

3.3.3 Parameters

In order to achieve the desired results, it was decided to use the same parameter values for T-Bot as UdacityBot (see figures 12 to 16 for these values). Initially, an attempt was made to improve the T-Bot vis-a-vis UdacityBot by adjusting several key parameters. However, it was not possible to find the right combination of values to further improve the performance and stability of the T-Bot. In the end, it was decided to go back to the values used for UdacityBot.

4 Results

4.1 Localization Results

4.1.1 UdacityBot

For UdacityBot, the localization results looked reasonable. The robot managed to reach the goal, as described in the previous section. The particles converged within a relatively short time and appeared to be distributed narrowly and centered around the robot. A significant portion of particles pointed in the right direction. It took this robot 20 to 30 minutes to reach the goal depending on the simulation. The path was somewhat roundabout rather than smooth.

It is important to note that there were several problems during the simulation. At times, the robot got stuck next to a wall and remained immobile, and the only way to alleviate the problem was to restart the simulation.

Other times, the robot spent a large amount of time (45 minutes or more) moving around in the circles at the top of corridor and would never get further down. When the simulation took an excessive amount of time in this particular way, and the robot did not show any signs of being able to exit the corridor, the simulation had to be restarted. Obviously, this was not an ideal situation. However, after a couple of attempts the robot did manage to move down the corridor and turn into the open area to reach the goal. The image of UdacityBot at goal position with PoseArray displayed has been shown previously in figure 8.

4.1.2 T-Bot

An attempt was made to improve the design of the T-Bot to alleviate some of the problems with UdacityBot. By reducing the parameter max_particle, an attempt was made to make the particles even more narrowly distributed and centered compared to UdacityBot. However, reducing this parameter did not have the intended effect and led to instability of the model.

To alleviate the problem of the robot occasionally getting stuck, the controller_frequency was reduced. However, this led to model instability with an increase in warning messages in the terminal. The transform_tolerance was also lowered but this did not seem to improve the simulation and led to instability.

To alleviate the problem of the robot spending an excessive amount of the time in the upper parts of the corridor parameters such as obstacle_range, raytrace_range, inflation_radius and sim_tim were adjusted [7], [8]. However, it was not possible to find the right combination of values to alleviate this problem .

Finally, it was decided to re-use the parameters of UdacityBot in order to reach the goal successfully, as the simulation of T-Bot had become too unstable. The image of T-Bot at goal position with PoseArray displayed has been shown previously in figure 10.

So, in the end, the two robots were both able to reach the goal and seemed to perform equally well. However, it could only be achieved if T-Bot reused the parameter values of UdacityBot.

5 Discussion

The two robots did not behave in an optimal way. Occasionally, they got stuck or spent a large amount of time moving around the corridor without exiting. Although further parameter tuning was carried out, it was not possible to alleviate these problems. So although the AMCL algorithm appeared to be a reasonable choice, it was hoped that the system would have performed better. It is possible that this was due to limitation of the hardware platform (Dell laptop). Also, it is possible that the optimal combination of parameter values had not been found or that additional parameter types had to included in the tuning process.

The kidnapped robot problem is when a robot abruptly disappears from one location and reappears in another, and it can be viewed as a benchmark for a worst-case scenario. It could take place when someone picks up the robot and places it randomly somewhere else, or, if the system malfunctions temporarily and all sensor data are reset. In this project, the proposed models are unlikely to work well for the kidnapped robot problem. The robot models showed weaknesses in terms of stability and accuracy. Also, in general, it is a challenging task to use Monte Carlo Localization for mobile robotics [9]. Therefore, it is likely that the system would struggle carrying out localization for the kidnapped robot problem.

The AMCL algorithm would work well in any industry domain where the path of the robot are guided by clear barriers. For example, the AMCL algorithm could be used to achieve high-accuracy vehicle localization in autonomous warehousing [2]. In the past, industrial autonomous warehouses have mainly relied on external infrastructure to obtain precise location data. This approach has increased warehouse installation costs and decreased system reliability, as the localization system has been sensitive to errors in external measurements and external infrastructure getting dirty or damaged. Instead, the AMCL algorithm in combination with other techniques could be used to create autonomous warehouse vehicles that take care of localization themselves [2].

6 Conclusion and Future work

Two robots were developed and tested in a simulated environment. The definition of the first robot was given as part of the project, and the second robot was created independently. The aim was to successfully navigate through a maze using the Adaptive Monte Carlo Localization algorithm to reach a goal position. Both robots achieved the project requirement of reaching the goal and appeared to perform equally well. They reached the goal position in 20 to 30 minutes, depending on the simulation. The localization algorithm appeared to work in the right way. The particles converged within a relatively short time and were distributed narrowly and centered around the robot. A significant portion of particles (not all) pointed in the right direction. Despite both robots being able to reach the destination, the simulation did not run in an optimal way. At times, there were problems with the robots getting stuck or spending an unusual amount of time inside the corridor. This could have been due to limitations of the hardware platform. Also, it is possible that the optimal combination of design specifications and parameter values had not been found or that additional parameter types had to included in the tuning process.

In terms of future work, the model could be implemented on a faster hardware platform or alternatively in the cloud. Secondly, there was a particular focus on following parameters to improve the localization results of T-Bot compared to UdacityBot: controller_frequency, transform_tolerance, obstacle_range, raytrace_range, inflation_radius and sim_tim [6]–[8]. However, there are other types of parameters that could be included and tuned.

Also, choice and location of sensors can be investigated. In the model for T-Bot, the laser rangefinder was located at position (0.15, 0, 0.1). This sensor could be moved to a more elevated position, for example, to position (0.15, 0, 0.3). Note, the last coordinate represents the z-axis and thus elevation of the sensor above the robot. The Gazebo plugin for the laser rangefinder is defined in the URDF directory. This plugin corresponds to a certain model of the rangefinders provided by the company Hokuyo. A more up-market laser rangefinder from this company's range of products (or alternative supplier) could be considered. The corresponding Gazebo plugin could then be adjusted accordingly.

The weight of the chassis of both robots was set to 15 kilograms. Experimentation could be done using an increased weight to make the system more stable. Also, the chassis of UdacityBot was a rectangle of dimensions 0.4 m x 0.2m x 0.1m. This footprint could be doubled or trebled to investigate whether this could improve stability and localization results. Furthermore, the chassis of T-Bot was a cylinder of length 0.1 m and radius 0.1 m, which was quite small compared to the size of the wheels. Again, experimentation could be done with a larger cylinder of radius 0.2 m or 0.3 m.

The T-Bot model could be deployed on a Jetson TX2 board running ROS and Ubuntu 16.04 Linux. This hardware configuration would have adequate processing power both in terms of CPU capability and memory to implement the model. The models were simulated in Gazebo and RViz only, and no drivers were implemented to actuate drive motors or sensors. This would have to be implemented on the TX2 board. The TX2 board has a camera which could be connected to the model. Also, the Jetson TX2 board could be mounted on a TurtleBot in order to implement the differential drive. The TurtleBot is a low-cost robot development kit and has been designed for easy assembly. It has a cylindrical shape like the T-Bot. Furthermore, later versions of the TurtleBot include a laser rangerfinder, which could be connected to the system [10], [11].

REFERENCES

[1] Udacity, "Classroom material, term 2, sections 9-13, robotics software engineer nanodegree program," 2018.
[2] G. Vasiljevic, D. Miklic, I. Draganjac, and Z. Kovacic, "High-accuracy for vehicle localization for autonomous warehousing," *Robotics and Computer-Integrated Manufacturing*, 2016.
[3] Wikipedia.org, "Kalman filter," 2018.
[4] Wikipedia.org, "Extended kalman filter," 2018.
[5] ROS.org, "Navigation stack never reach goal," 2018.
[6] ROS.org, "Setup and configuration of the navigation stack on a robot," 2018.
[7] ROS.org, "Basic navigation tuning guide," 2018.
[8] K. Zheng, "Ros navigation tuning guide," 2016.
[9] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial Intelligence*, vol. Summer 2001, 2001.
[10] M. Quigley, B. Gerkey, and W. D. Smart, *Programming Robots with ROS - A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015.
[11] ROS.org, "Turtlebot," 2018.