# Lab 5 Report
# ROS Navigation Stack

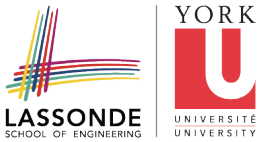
EECS 4421, Intro. to Robotics

Course Instructor: Michael Jenkin

April 26, 2021

Group Composition


| | |
|---|---|
| Ramavath, Sai Varun | varun333@my.yorku.ca, 213506936 |
| Patel, Henilkumar | henil@my.yorku.ca, 215245707 |


**4th Year**
**Electrical Engineering**
**and Computer Science**

# Contents

# List of Figures



# Listings

# 1 Robot Localization

## 1.1 Environment

The localization exercise requires the creation of a complex environment containing several obstacles; this has been modelled as the 'Dust' map from popular video game *Counter Strike: Global Offensive*. The map is properly credited in the `model.config` file found on the course Github repository linked in the appendix.

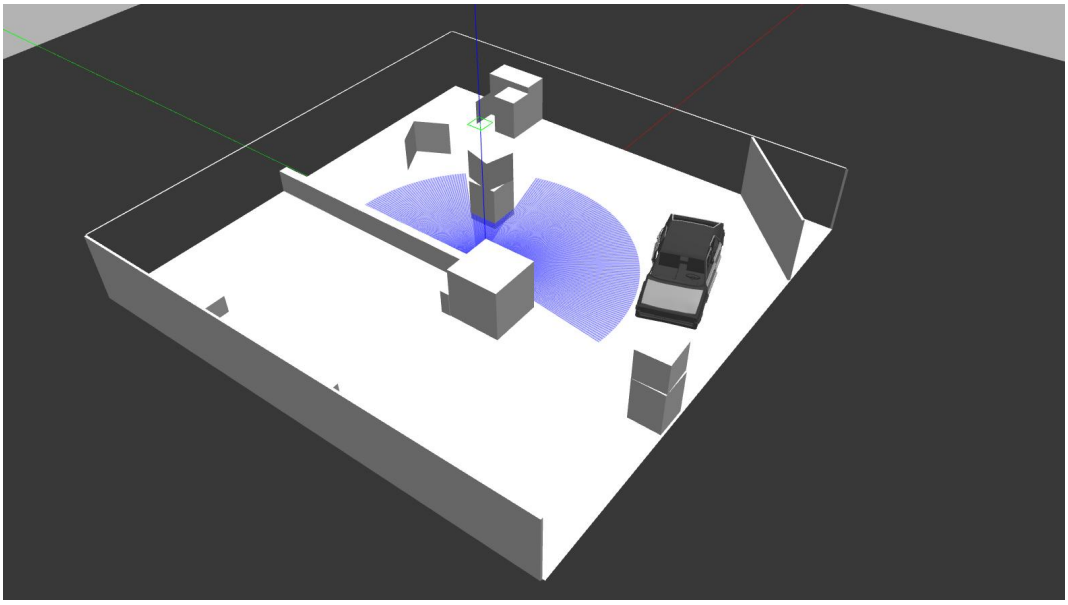


**Figure 1:** Complex environment created to run localization simulations for a differential drive robot. Note that the model features bounding walls on all exterior planes but one is not showing here due to Gazebo artifacts.

## 1.2 AMCL

Adaptive Monte Carlo Localization (AMCL) is a probabilistic technique used in inferring a (two-dimensional motion) robot's current pose with respect to a given (known) environment. The robot's current surroundings are determined through the use of range sensors that provide an 'image' of the robot's locale and its position relative to obstacles around it; in this simulation exercise this is achieved through a laser LIDAR sensor which has seen a fair amount of use in recent autonomous driving and robotics applications. AMCL takes the LIDAR readings and compares them with a two-dimensional occupancy grid map created using Simultaneous Localization and Mapping (SLAM) to infer the robot's current pose. The occupancy map is created by driving the robot around a desired environment and mapping LIDAR and odometry data to save obstacle and static object positions; this map is used later in the creation of a global cost map that determines the optimal path a robot must take in reaching a specified location and pose.

As one can expect, the localization process is not perfect and requires tuning several parameters to achieve a high level of positional accuracy and a dependable robot localization estimate. The so-called

'alpha' parameters control the expected error in robot odometry that results from rotation and translation, as well as one's effect on the other. The first parameter defines the expected rotational error when the robot is instructed to do so (rotate around `base_link`), the second parameter defines the expected rotational deviation when the robot is sent a translation command (move forward, backward), the third parameter defines the expected translational error from sending the robot a translation command, and the fourth parameter defines the expected translation error when the robot is sent a rotation command.

While these have a significant impact in real-life use cases, the (simulated) differential drive robot used in this exercise does not show a large error in any of the ways highlighted above which will be kept in mind when tuning these parameters for the exercise's use case. The default values of these parameters were given as:

$$\alpha_1 = 0.150, \quad \alpha_2 = 0.015, \quad \alpha_3 = 0.050, \quad \alpha_4 = 0.010$$

## 1.3 Implementation

The basic flow for setting up and optimizing AMCL in ROS is to (i) generate a two-dimensional occupancy grid map from laser and robot pose data using SLAM, (ii) use AMCL to estimate the robot's pose with the generated map fed in, (iii) tune relevant alpha parameters in an AMCL launch file used to get accurate localization estimates.

### 1.3.1 Environment mapping

To the first point, ROS contains a mapping utility called `gmapping` that is used to generate the necessary occupancy grid map. The command must be invoked while the desired environment is loaded into Gazebo along with the robot model used - in this case the differential drive robot. The `cpmr_ch7` package contains a launch file that spawns the differential drive robot with a laser scanner provided in the `cpmr_ch4` package, however it does so without launching a world file; the end result is that launching the file does nothing. An easy fix for this is to add a line at the top of the launch file that spawns the robot into an empty world, and then manually loading in the complex environment model but this gets frustrating quickly when Gazebo needs to be closed and reopened for whatever reason. The complete fix for this problem, then, is to create a world file based on the empty world, pre-load the environment at its origin and then load in the robot on top of it all; this is done by first creating a `worlds` directory in the `cmpr_ch7` main directory and adding a `DUST.world` file that contains the lines in Listing 1.

```
<?xml version="1.0" ?>
<sdf version="1.4">
 <world name="default">
  <include>
   <uri>model://ground_plane</uri>
  </include>
  <include>
   <uri>model://sun</uri>
  </include>
  <include>
   <uri>model://complex_env</uri>
   <name>dust</name>
   <pose>0.0 0.0 0.0 0.0 0.0 0.0</pose>
```

```
  </include>
 </world>
</sdf>
```

**Listing 1:** World file created to make launching the robot easier. Note that the environment has been added to `/.gazebo/models/` and is referenced here as `complex_env`.

```
<include file="$(find gazebo_ros)/launch/empty_world.launch">
 <arg name="world_name" value="$(find cpmr_ch7)/worlds/DUST.world"/>
</include>...</launch>
```

**Listing 2:** Environment and robot launch file that utilizes the previously created `DUST.world` to spawn the robot and the environment simultaneously, cutting down on time wasted in arranging the above properly.

The world file can be referenced in the `gazebo_laser_configuration.launch` file as seen in Listing 2.

With that out of the way, a `gmapping` node is created with the robot model loaded into Gazebo at the default (0, 0, 0) position in the environment using the following syntax:

```
rosrun gmapping slam_mapping
```

The robot is then driven around the map using a `teleop_twist_keyboard` node to capture the positions of all static objects and obstacles in the environment, yielding the occupancy grid map shown in Fig. 2b; notice the obstacles shown as darker regions that correspond to those seen in Fig. 1. The map is saved using a `map_server` node that is called in the `maps` directory of `cpmr_ch7` using:

```
rosrun map_server map_saver -f output
```

which saves `output.yaml, output.pgm` files that are used by AMCL to provide estimates.

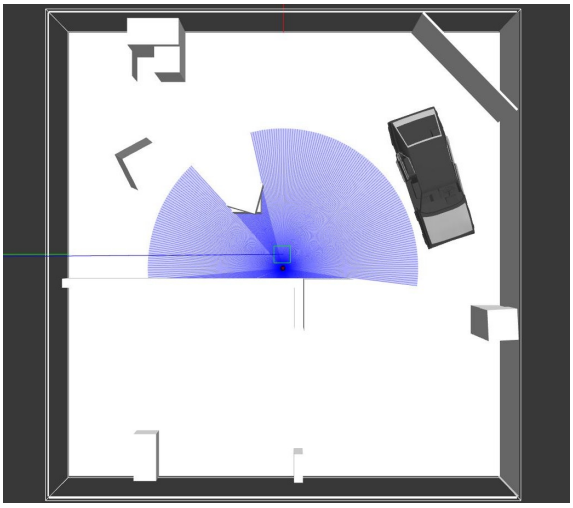
### 1.3.2 Running AMCL

```
<node name="map_server" pkg="map_server" type="map_server" args="$(find cpmr_ch7)/maps/
  output.yaml" />...<node...>...
```

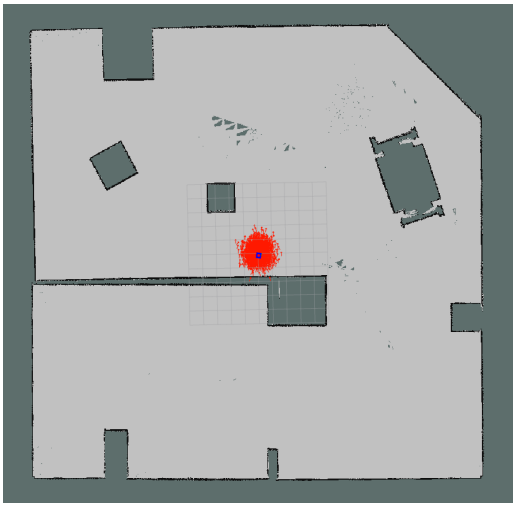
**Listing 3:** The required `map_server` line to load in the desired environment map, placed in the `amcl.launch` file.

The next step is to run the AMCL launch file that was supplied with the package, making sure that the `map_server` utilizes the newly generated `output.yaml` file, as shown in Listing 3. Then, we provide a good estimate of the robot's pose using the '2D Pose Estimate' feature in RVIZ and placing the arrow at the approximate location of the robot, pointing in the direction it is facing. The robot's initial pose as seen in Gazebo and RVIZ after performing this step is shown in Fig. 2. The red arrows in Fig. 2b are particle vectors that are used to represent all possible robot poses at this epoch (point in time), and are spread out quite far since the initial localization has zero to few data points to work with resulting in high pose uncertainty. The vectors are all pointing towards the top of the map, indicating that the robot is facing that direction. It is a little difficult to decipher the robot's pose from Fig. 2a, but the robot is indeed facing the top of the map which is evident when one observes that the characteristic gap in the LIDAR lines is not shown anywhere in the scan perimeter; also, the robot spawns at (0, 0, 0) facing the positive x-axis which is defined as pointing to the top of the map.

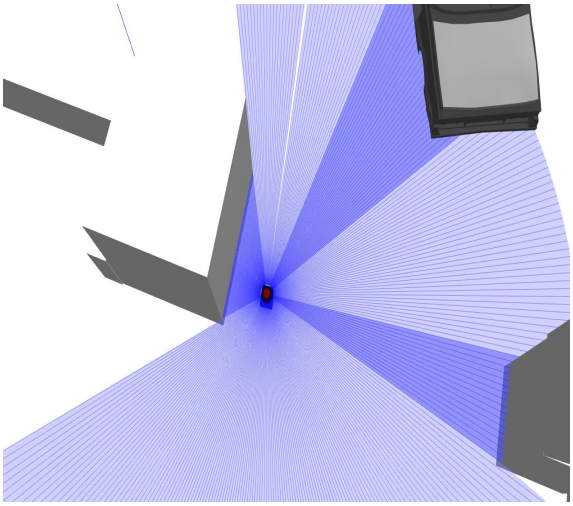
All subsequent pose estimates and localization attempts prove fruitful since more data points are collected as the robot is driven around the map. The robot is again operated using the tele-operation method highlighted previously. Although the angle of the Gazebo image in Fig. 2c is not the same as Fig. 2d, the location of the robot in both is fairly consistent at the default alpha parameter values shown earlier, which brings this discussion to one main point: the robot showed significant rotational drift when instructed to start at and stop from 0.5 m/s translational speed, but showed smaller amounts of drift when in motion. The robot was then controlled to start driving at a slower speed which was ramped up in an effort to minimize the start-up error and simulate real-life behaviour. The expected error is quite small, yet not quite negligible so the alpha parameters need to be tuned to reflect this.



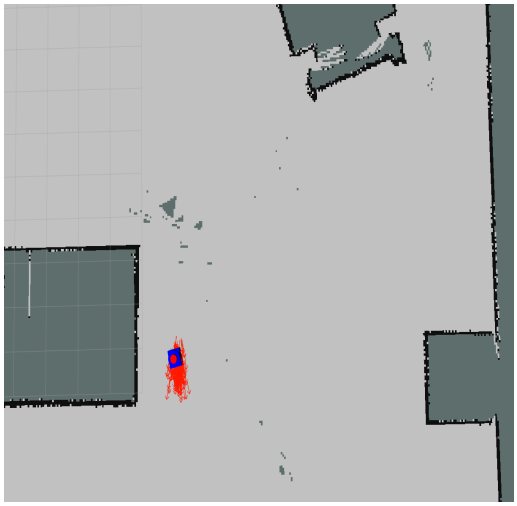
**(a)** Initial pose - Gazebo



**(b)** Initial pose - RVIZ



**(c)** Updated pose - Gazebo

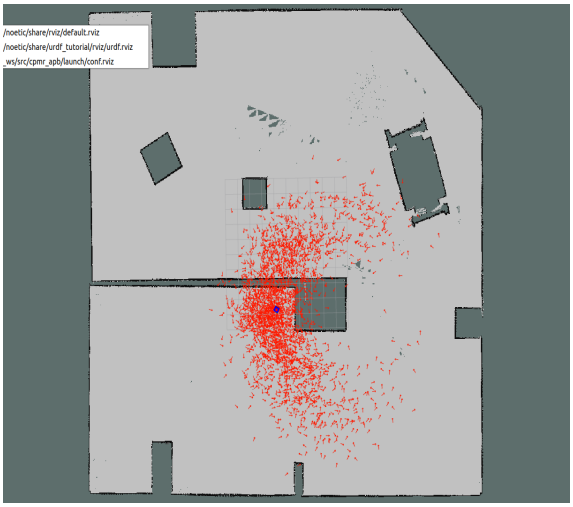


**(d)** Updated pose - RVIZ

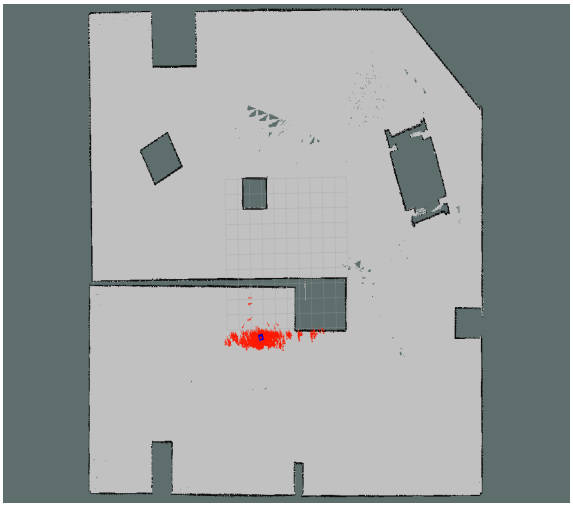
**Figure 2:** (a) - (b) Robot's initial pose as shown in the Gazebo model and the `gmapped` complex environment of Fig. 1. The point vector cloud showing the robot's possible poses has quite a significant spread to it which is indicative of AMCL's uncertainty in the robot's initial orientation. (c) - (d) Robot's updated pose and localization estimate after driving around the map.

### 1.3.3 Recovering from loss of localization

The loss of localization test is performed to ensure that the AMCL node is able to quickly and sufficiently recover from a sudden change in the robot's position - say if it were picked up and dropped somewhere else in the built-environment. Fig. 3a shows the particle vectors going haywire after this is done which indicates that the AMCL node does not have a good understanding of the robot's location, requiring more data points to be collected in order to get a better estimate (much like the initial pose estimation). Fig. 3b through 3d show the recovery process which involves driving the robot around a short distance.

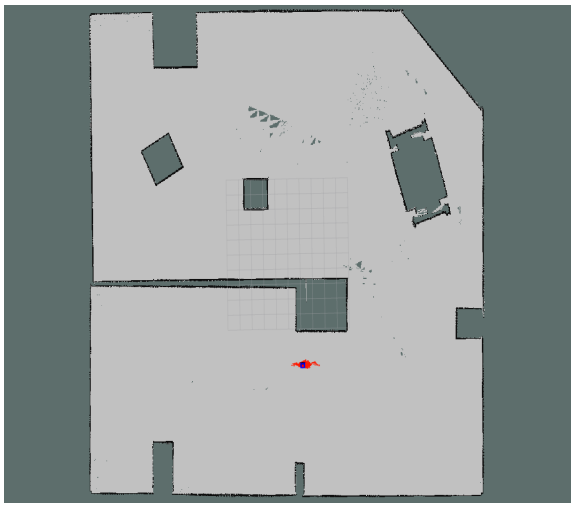



**(a)** Dropped robot, epoch zero



**(b)** Dropped recovery, epoch one

**(c)** Dropped recovery, epoch two

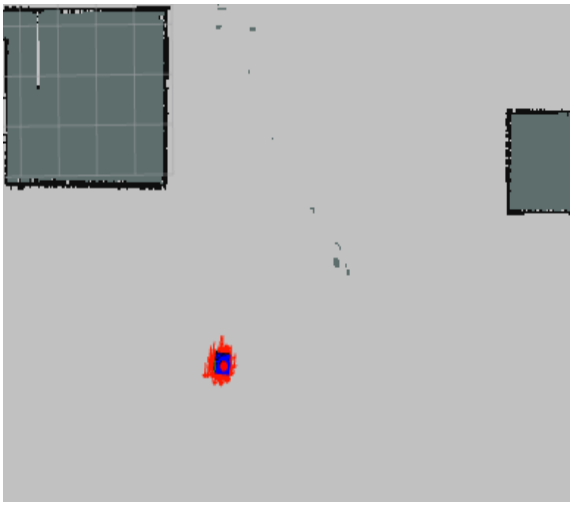


**(d)** Dropped recovery, epoch three

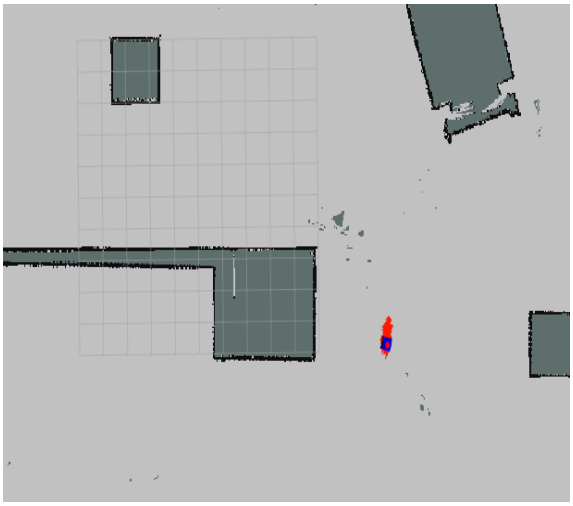
**Figure 3:** AMCL recovery from loss of localization. The robot was dropped at the location seen in (a), wreaking havoc on the localization estimate. (b) through (d) show how AMCL recovers from loss of localization.

### 1.3.4 Tuning alpha parameters

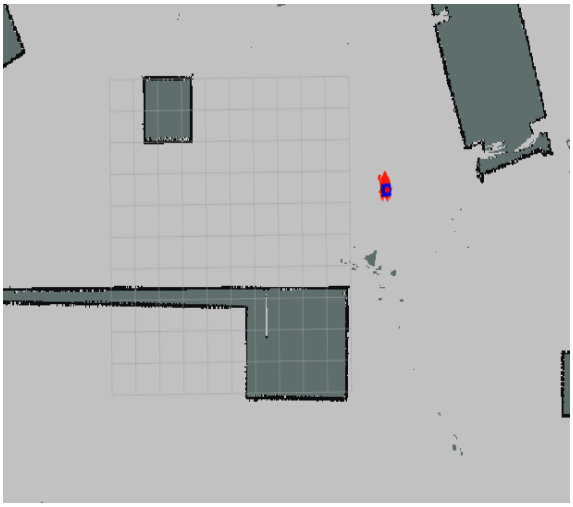
The first three alpha parameters were tuned in steps based on the driving style highlighted earlier while observing the impact of each on the localization estimate and particle cloud. As can be seen in Fig. 4a, there is a significant improvement in rotational uncertainty compared to Fig. 2d which is expected since the corresponding alpha parameter has been reduced from 0.15 to 0.05. Fig. 4b shows how the translation uncertainty has been reduced as indicated by the higher concentration of particles towards the center of the robot, and Fig. 4c shows how the spread has been minimized further, however, not as significantly. Referring to this ROS answer on AMCL parameter tuning, values that are too low hurt the localization process and therefore the following values were deemed to provide acceptable results: $\alpha_1 = 0.010, \quad \alpha_2 = 0.010, \quad \alpha_3 = 0.010, \quad \alpha_4 = 0.010.$



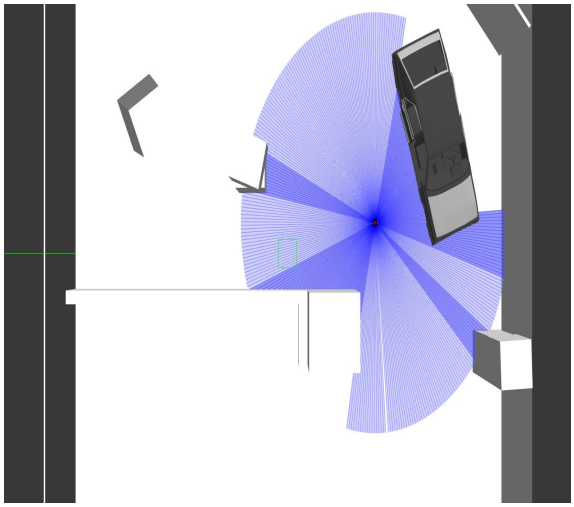
**(a)** $\alpha_1 = 0.05$



**(b)** $\alpha_1 = \alpha_2 = 0.05$



**(c)** $\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.01$



**(d)** Gazebo view with alpha parameters in (c)

**Figure 4:** Result of tuning alpha parameters for the AMCL node.

# 2 Robot Path Finding

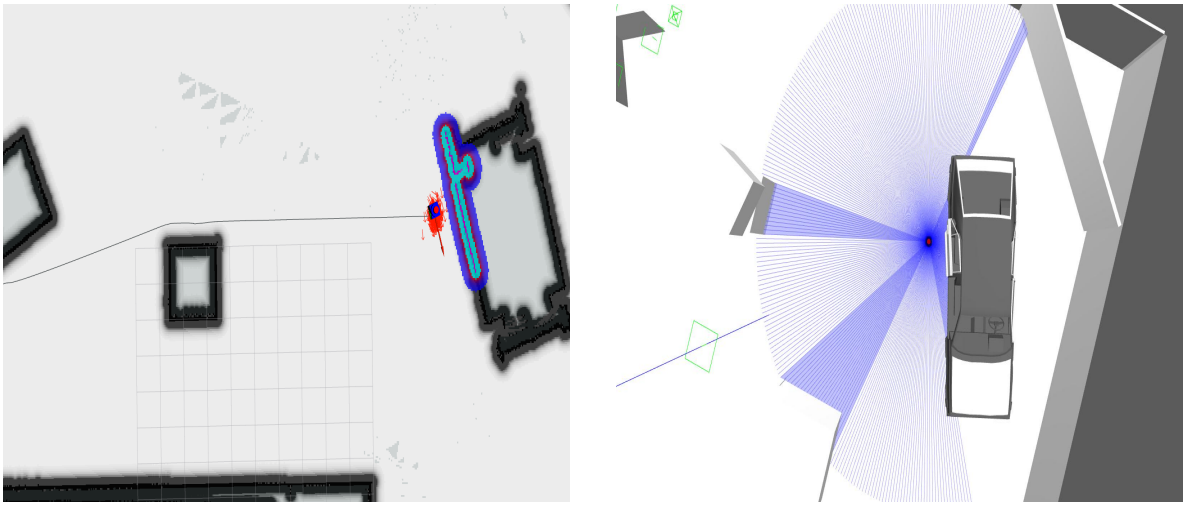
The path finding algorithm provided with the `cpmr_ch7` package utilizes the `move_base` ROS package to 'configure, run and interact' with the robot's navigation stack that we have built thus far. The package utilizes global and local costmaps to determine the optimal path to a given goal, taking into account all obstructions and obstacles in the environment. The `move_base.launch` file shows (Listing 4) that the package utilizes the occupancy grid created earlier to define a global cost map of the environment which tells the planning algorithm what areas to avoid. The package also creates a local cost map based on the robot's current localization estimate from AMCL to dynamically plan around obstacles as the robot encounters them.

```
1  <include file="$(find cpmr_ch7)/launch/amcl.launch" />...</launch>
```

**Listing 4:** `move_base` calls on the AMCL launch file which calls on the occupancy grid map discussed earlier to facilitate path planning.

Two changes were made to the `local_costmap_params.yaml, global_costmap_params.yaml` parameter files supplied with the `cpmr_ch7` package to adapt them to the Dust environment. The first change was to set the origin of the global map to the origin of the environment (0, 0, 0); the second was to change the `global_frame` of the local map to 'map' (from 'odom'), since the robot was experiencing obscure errors highlighting that messages to odom were requested at a past epoch. A quick 'ROS Answers' forum search yielded this fix which worked exactly as expected and led to a successful path traversal (Fig. 5). The changes are quite straightforward (and exactly as described), so they will not be shown in a listing here but instead can be found in the appendix (along with a screen dump of building the map).

## 2.1 Successful path



**(a)** Path planning and local costmap - RVIZ **(b)** Reached goal - Gazebo

**Figure 5:** Successful path planning attempt using the `move_base` package in ROS. Note the particle cloud arrows pointing in the direction of the desired robot pose at the goal, the black planned path line and the local cost map highlighting the edges of the obstacle.

Having made the changes above, the robot was able to successfully maneuver its way from one side of the map to the other as shown in Fig. 5a. The desired goal was set using the '2D Nav Goal' feature in RVIZ which presents an arrow that can be placed at a given location and pointed in the direction that the robot should be facing upon reaching this point. As the particle cloud shows in Fig. 5a, the robot faces the direction of the navigation goal arrow at the specified location. A successful path can be achieved by ensuring that the navigation goal is placed far enough away from an obstacle as will be discussed below.

## 2.2 Failed path

The failed attempt seen in Fig. 6 is a consequence of placing the navigation goal arrow within the collision zone of an obstacle. The path planner tried several different attack angles and orientations to get to the desired location but failed every time because the goal was placed in a zone that the planner was not allowed to reach. The robot did not stop struggling until a new navigation goal was placed to put it out of its misery. The faint green line in the same figure shows where the robot started from, indicating that it would have taken a simple right turn to reach the goal pose, whereas the robot is facing the exact opposite direction.

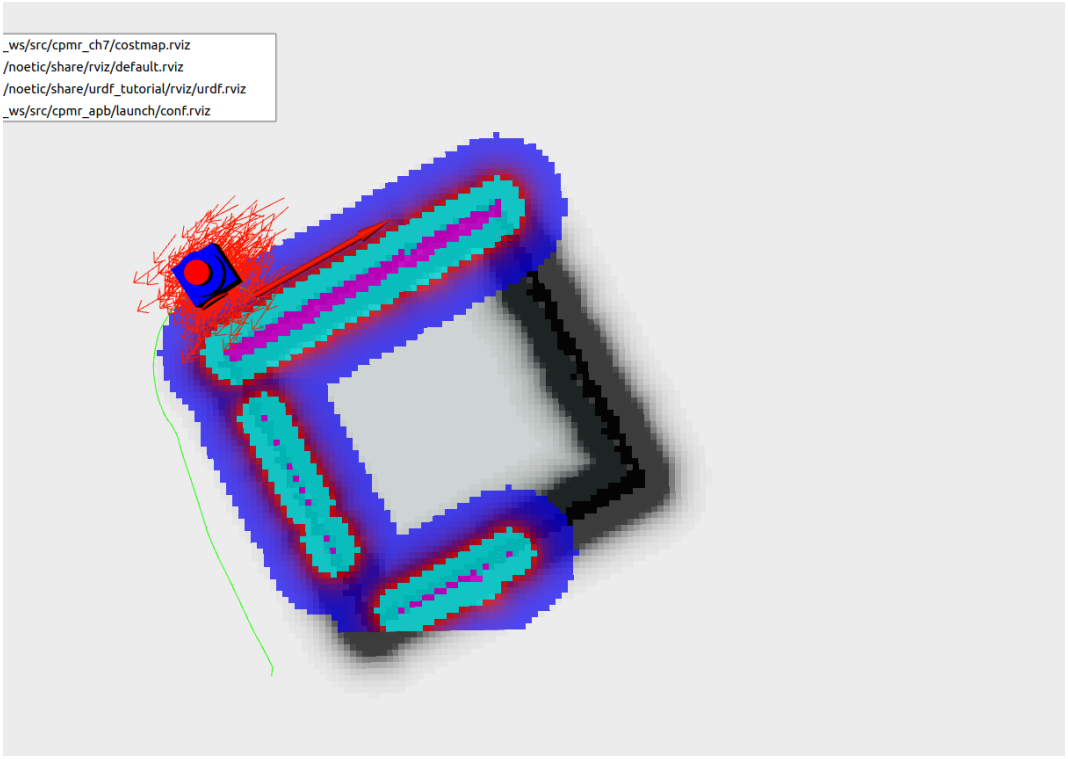



**Figure 6:** Robot cannot orient itself properly because the goal is set within the collision zone of the global and local costmaps. Note the planned path in green showing where the robot started from; it should not be facing opposite to the desired goal.

## 2.3 Costmaps

Fig. 7 shows both the global and local costmaps overlaid atop the occupancy grid map created earlier. The obstacle edge highlighted beside the robot is the local costmap, while the faint blue border on the rest of the obstacles represents the global costmap.

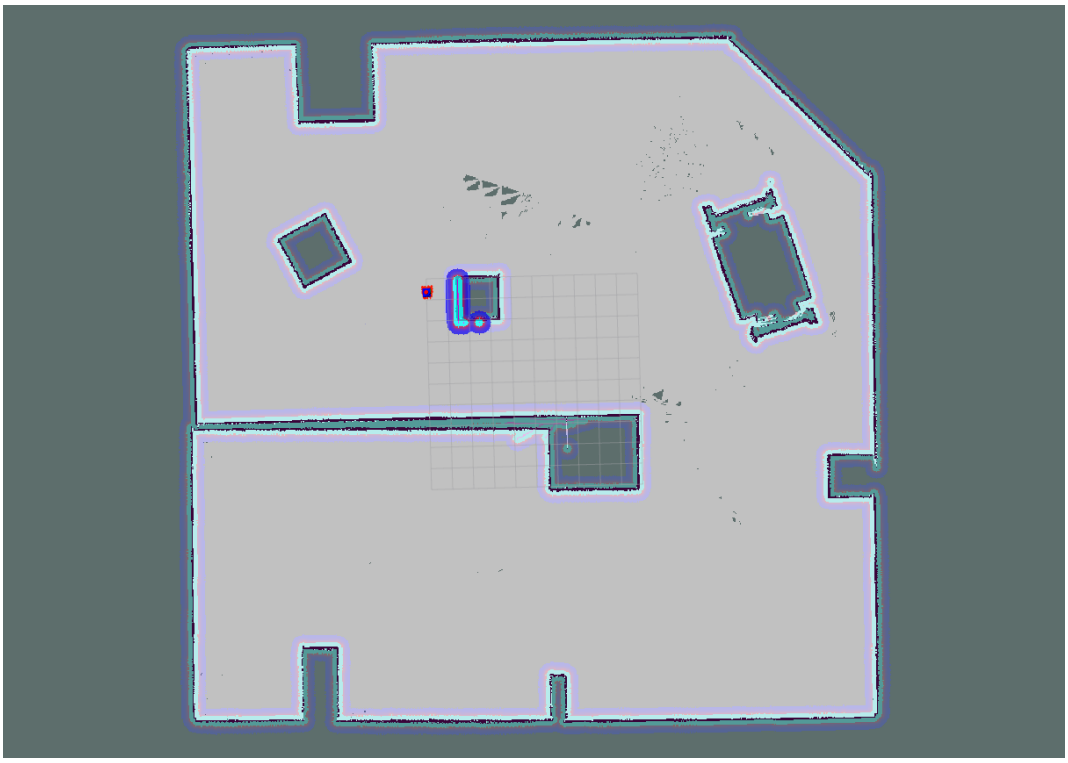
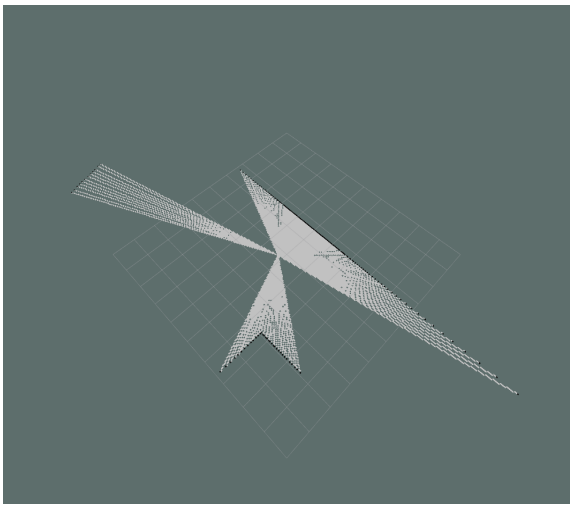


**Figure 7:** The Dust environment's global costmap and the robot's local costmap overlaid atop the occupancy grid map created earlier.

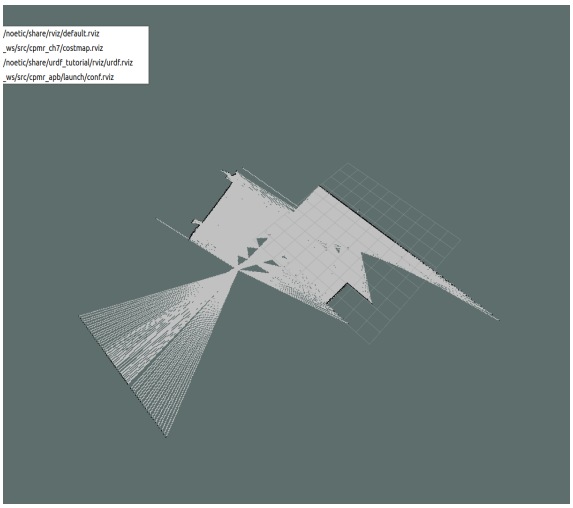
# Appendix

The code and environment model created for this exercise can be found at the *IntroRobotics_York* Github repository.

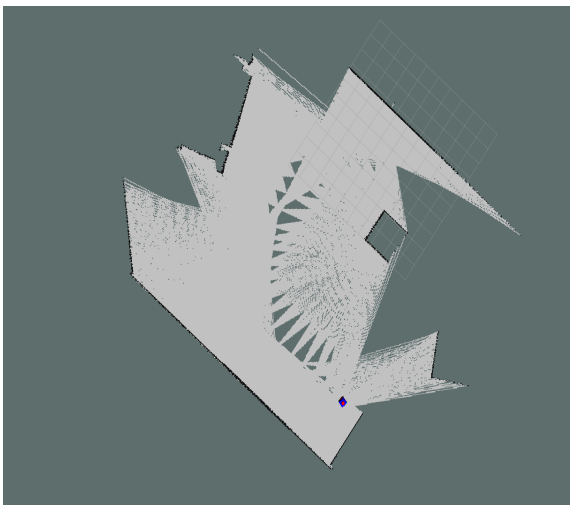
## Map building



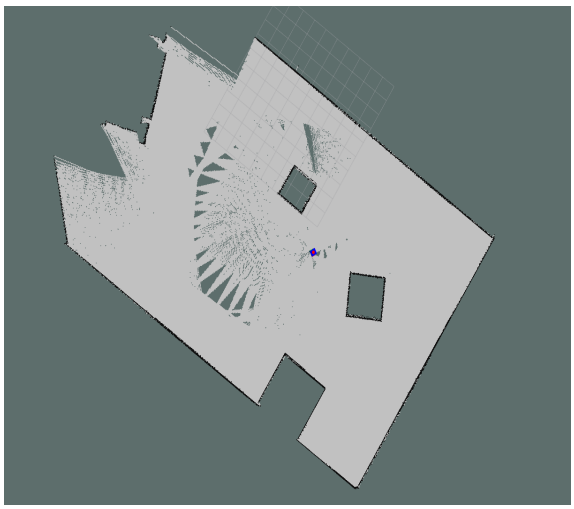
**(a)** Initial mapping, epoch zero




**(b)** Driving around, epoch one



**(c)** Driving around, epoch two



**(d)** Driving around, epoch three

**Figure 8:** Driving the robot around the map while running the mapping utility `gmapping` to capture an occupancy grid map of the Dust environment.