☐ Abhaycl / **RoboND-Where-Am-I-2P2**

Branch: master ▾   **RoboND-Where-Am-I-2P2** / README.md   Find file   Copy path

🖼 **Abhaycl** Add files via upload   826c177 a minute ago

1 contributor

---

303 lines (188 sloc)   17.9 KB

Where Am I Project Starter Code

The objective of this project is to learn how to utilize ROS packages to accurately localize a mobile robot inside a provided map in the Gazebo and RViz simulation environments.

How to run the program with your own code

For the execution of your own code, we head to the Project Workspace

Go to Desktop

You can launch it by running the following commands first

```
cd /home/workspace/catkin_ws
catkin_make
source devel/setup.bas
```

And then run the following in separate terminals

```
roslaunch udacity_bot udacity_bot
roslaunch udacity_bot amcl
rosrun udacity_bot navigation goal
```

The summary of the files and folders int repo is provided in the table below:

File/Folder	Definition
config/*	Folder that contains all the parameters and some values defined for you to help you get started.
launch/*	Folder that contains all the launch files in ROS that allow us to execute more than one node simultaneously.
maps/*	Folder that contains all the new environment using a map created by Clearpath Robotics.
meshes/*	Folder that contains all the parameterization of the sensors.
src/*	Folder that contains all the project repo we have provided you with a C++ node that will navigate the robot to the goal position for you.
urdf/*	Folder that contains all the robot's URDF description.
worlds/*	Folder that contains all the Gazebo worlds.
misc_images/*	Folder containing the images of the project.

File/Folder	Definition
README.md	Contains the project documentation.
README.pdf	Contains the project documentation in PDF format.

Steps to complete the project:

1. Follow the steps outlined in this Project Lesson, to create your own ROS package, develop a mobile robot model for Gazebo, and integrate the AMCL and Navigation ROS packages for localizing the robot in the provided map.
2. Using the Parameter Tuning section as the basis, add and tune parameters for your ROS packages to improve your localization results in the map provided. Feel free to explore new parameters using the resources provided earlier in the same section. Please include the image of RViz with the robot at goal position and the PoseArray displayed. Each run may take a long time so don't forget to screenshot your robot at the goal position.
3. After achieving the desired results for the robot model introduced in the lesson, implement your own robot model with significant changes to the robot's base and possibly sensor locations.
4. Check your previous parameter settings with your new robot model, and improve upon your localization results as required for this new robot.
5. Document your work. You could use this Writeup Template.

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Abstract

Where Am I? The goal of this project is to configure a number of ROS packages that can be used in conjunction to accurately localize and navigate a mobile robot inside a provided map in the Gazebo and RViz simulation environments.

Introduction

This project consists of creating a complete ROS package that includes a Gazebo world, a robot model in URDF and using the ROS navigation stack to localize the robot on the map as well as move it to a desired destination and pose avoiding any obstacles on the way.

Background / Formulation

Robots often operates in unpredictable environments where the agent is uncertain about its state. Using calculus and probability theory it is possible to manage the this uncertainty and represent the robots beliefs of its state in mathematical form that can later be used for decision making.

For a robot to find its position on a map (localization problem) it needs to filter noisy sensor data using probabilistic methods. There are 3 types of localization problems:

1. Position tracking

This is the first and simplest form of the localization problem: in this scenario the robots initial position is known and the algorithm keeps track of the robot's position as it moves.

2. Global localization

For cases where the initial position is unknown and needs to be determined at as the robot moves. This problem combines the uncertainties from measurements and actions in a cycles to achieve a precise estimate of the location.
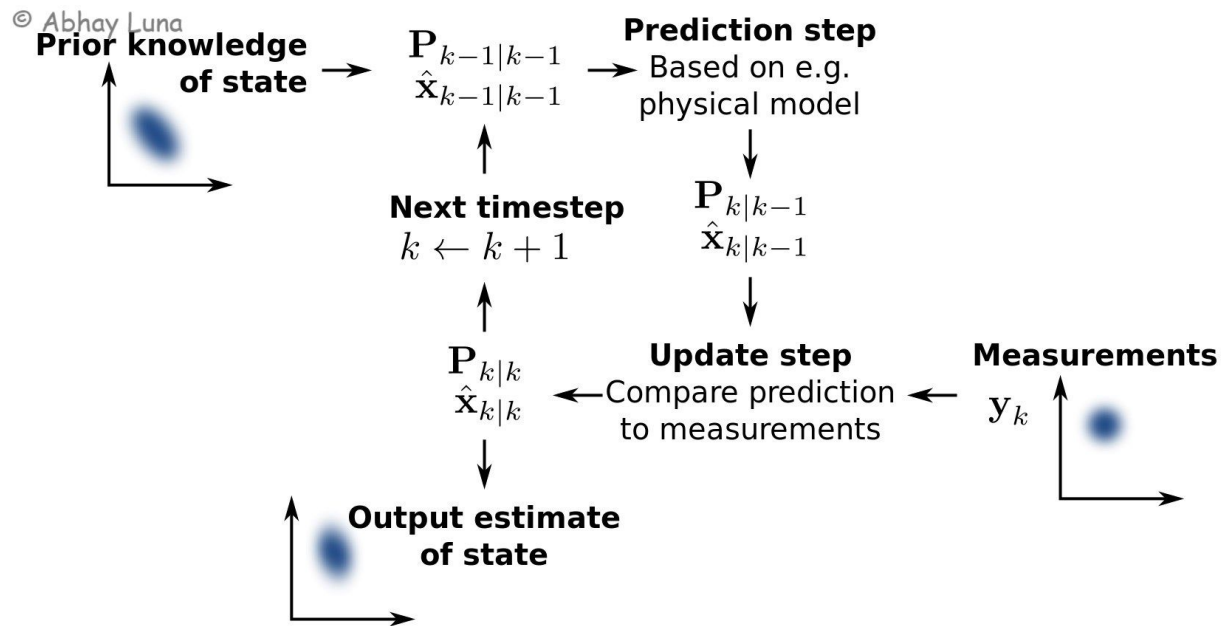
3. Kidnaped robot

The most difficult localization problem is to recover from abrupt changes in position like moving a robot from a position on the map to another. This is particularly difficult for bayesian algorithms to recover from since they preserve an internal belief that interfere with the new robot state.En el programa del curso se nos presentaron dos métodos de localización: Filtros Kalman y filtros de partículas. Ambos métodos probabilísticos se pueden aplicar con éxito al seguimiento de posición y a la localización global y más tarde ayudarán a localizar mi robot en Gazebo y RViz.

Kalman filters

The Kalman filter estimates the value of a variable by updating its estimate as measurement data is collected filtering out the noise. Kalman Filters models the state uncertainty using Gaussians and it is capable of making accurate estimates with just a few data points.

KF starts with an initial state estimate then performs the following cycle: measurement update produced by sensor measurements followed by a state prediction from control actions.

© Abhay Luna

Prior knowledge of state $\rightarrow$ $\mathbf{P}_{k-1|k-1}$ $\hat{\mathbf{x}}_{k-1|k-1}$ $\rightarrow$ **Prediction step** Based on e.g. physical model

Next timestep $k \leftarrow k + 1$

$\mathbf{P}_{k|k-1}$ $\hat{\mathbf{x}}_{k|k-1}$

$\mathbf{P}_{k|k}$ $\hat{\mathbf{x}}_{k|k}$ $\leftarrow$ **Update step** Compare prediction to measurements $\leftarrow$ **Measurements** $\mathbf{y}_k$

Output estimate of state

Multidimensional KF

Most of real world robots operates on multiple dimensions, i.e. a drone's position is defined by x, y and z coordinates and orientation in roll, pitch and yaw angles. This motivates generalizing the KF to multiple dimensions with equations in matrix format.

The MKF algorithm consists of calculating the Kalman Gain K that determines how much weight should be placed on the state prediction, and how much on the measurement update. It is an averaging factor that changes depending on the uncertainty of the measurement and state updates.

© Abhay Luna

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

$$x = x' + Ky$$

$$P = (I - KH)P'$$

The Kalman gain K averages the measurement update P and motion update x

Extended Kalman Filter

Kalman Filter assumes that motion and measurement models are linear and that the state space can be represented by a unimodal Gaussian distribution. Most mobile robots will execute non linear motion like following a curve. Non linear actions will result in non-Gaussian posterior distributions that cannot be properly modeled by a closed form equation.
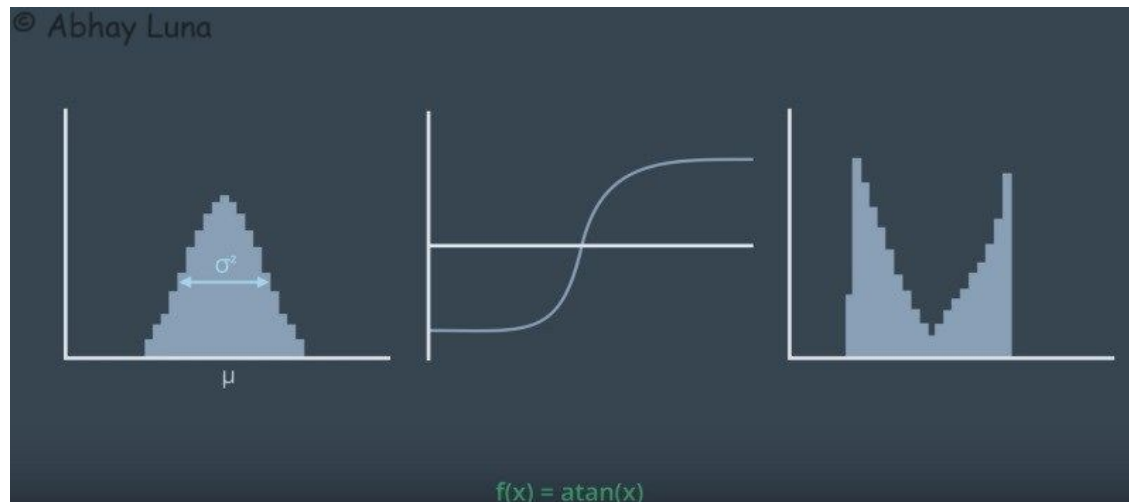
Gaussian prior (left) subject to a non-linear action (atan(x) — center) results in a non-Gaussian posterior (right)

EKF approximates motion and measurements to linear functions locally (i.e. by using the first two terms of a Taylor series) to compute a best Gaussian approximation of the posterior covariance. This trick allows EKF to be applied efficiently to a non-linear problems.

y = z − Hx′

is replaced with the nonlinear h(x'):

y = z — h(x′)

This is where the multivariate Taylor series comes into play. The function function h(x) can be approximated by a Taylor series centered about the mean μ, as defined below.

$$h(x) \simeq h(\mu) + (x - \mu)^T Df(\mu)$$

The h(x) approximation above is now linear around μ and will produce a Gaussian posterior.


$$f(\mu) + \frac{\delta f(\mu)}{\delta x}(x - \mu)$$
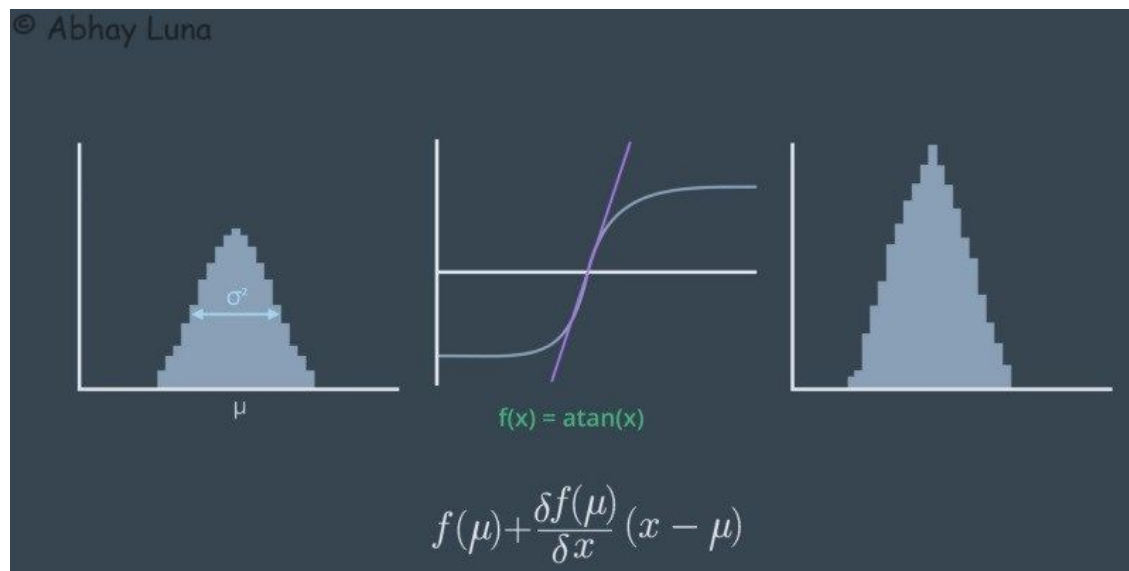
Linear approximation (in purple) of the atan(x) function around x = 0

Particle filters

Monte Carlo localization algorithm similar to Kalman Filters estimates the posterior distribution of a robot's position and orientation based on sensory information but instead of using Gaussians it uses particles to model state.

The algorithm is similar to KF where motion and sensor updates are calculated in a loop but MCL adds one additional step: a particle resampling process where particles with large importance weights (computed during sensor updates) survive while particles with low weights are ignored.

In the MCL example below all particles are uniformly distributed initially. In the following update steps the particles that better match the predicted state survive resulting in a concentration of particles around the robot estimated location.
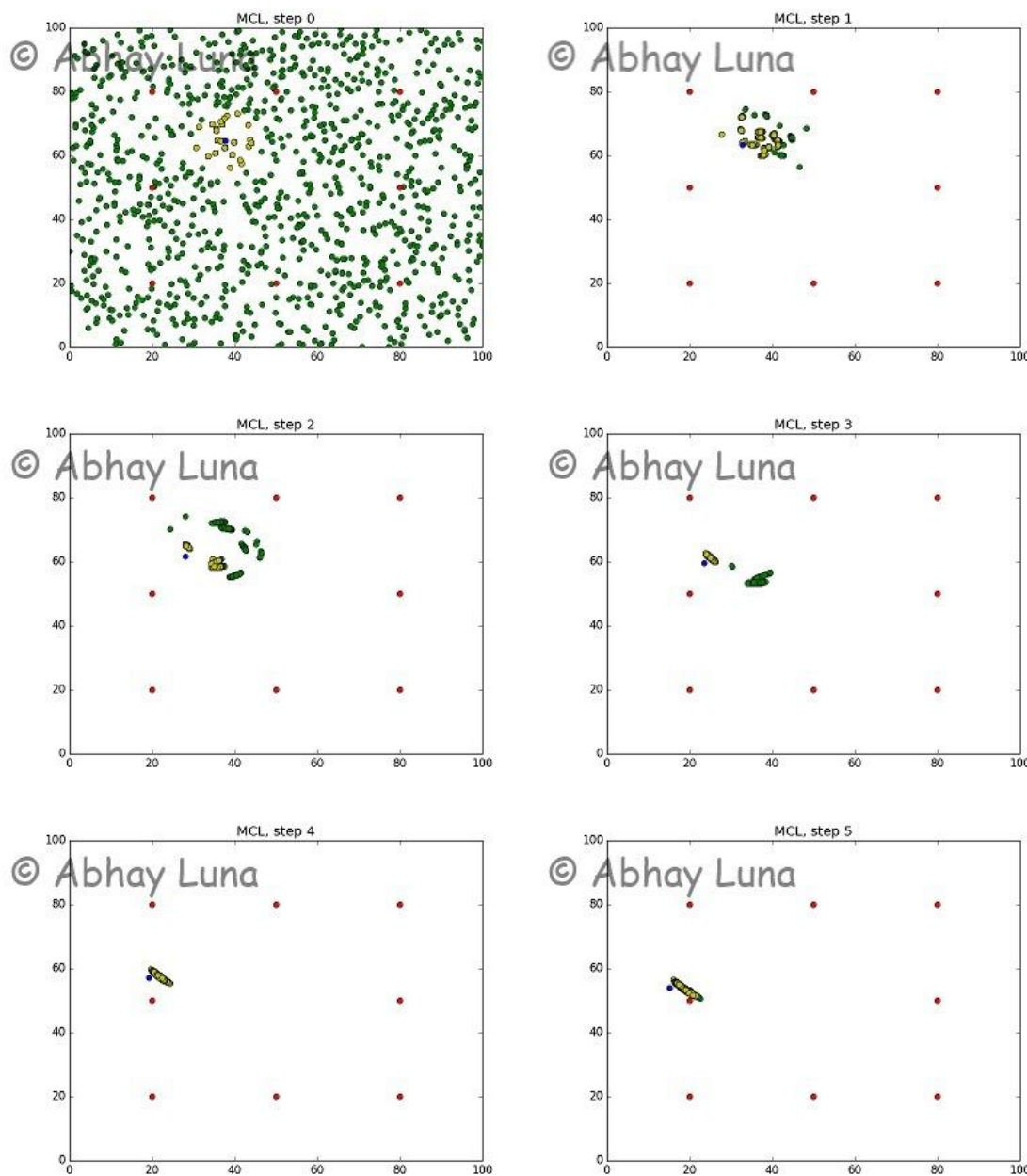
MCL steps visualization — Particles (green/yellow) start uniformly distributed (step 0) and then concentrates around the ground truth (blue) (steps 1–5). Robot's sensors read distances to landmarks (red).

MCL solves the local and global localization problems but similar to KF is not suitable for addressing the kidnaped robot problem.

Model Configuration

First we started configuring the world and launch files by following the instructions in the classroom, then we proceed to model the 2 robots below. The first robot based off the classroom (left) and the second is a custom robot (left).
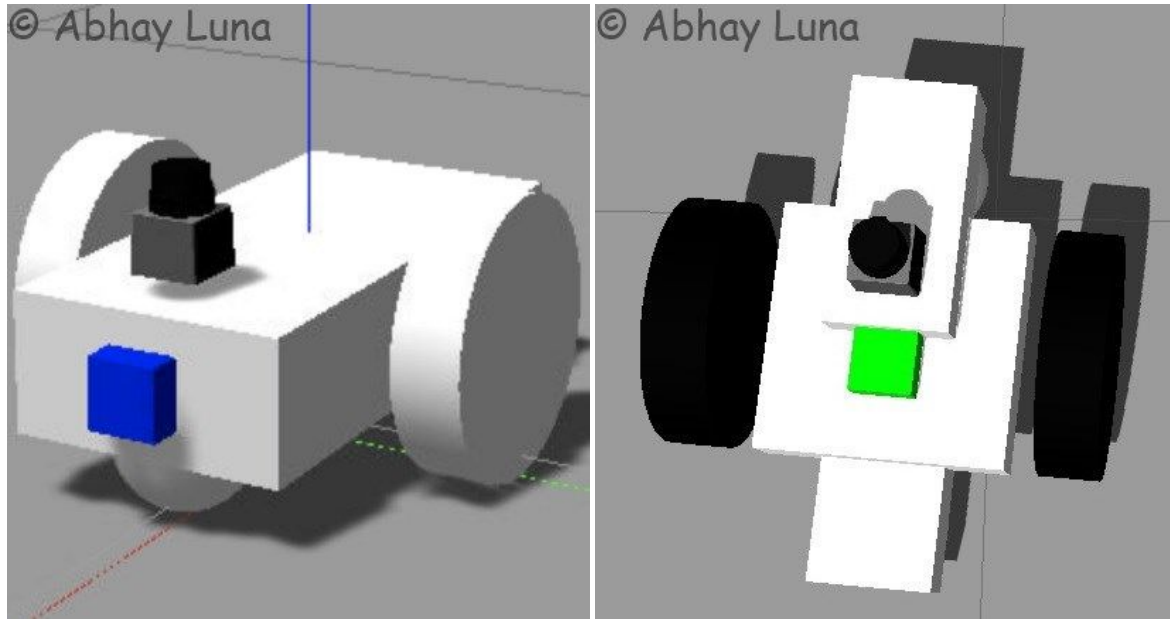

Classroom robot (left) and custom robot (right).

The custom robot has a differential actuation, just like the classroom robot, instead of 4 wheels of a regular car. so only 2 wheels are maintained to comply with the ROS navigation package hardware requirement.

Tuning Parameters

Now it will be described the choice the parameters for each of the packages from the ROS navigation stack. The next tutorial is a starting point as it provides an excellent overview of the parameters and packages discussed below.
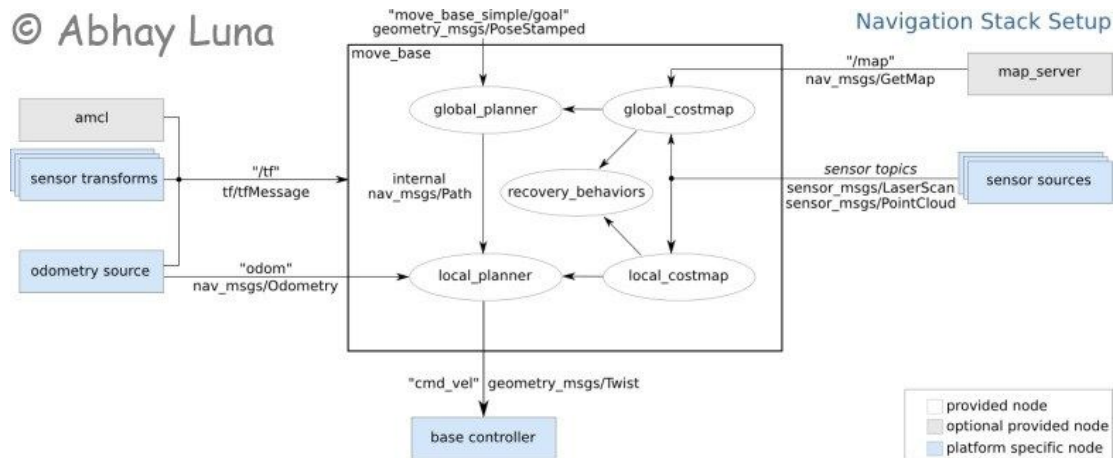
amcl.launch

The only parameters that were needed for tweek in the AMCL launch file was the translational and rotational movement required before performing a filter update. The default values were too large for the slow moving robot. After reducing them by an order of magnitude, status updates were obtained far more frequently resulting in a reduction of the location uncertainty to a minimum.

update_min_d = 0.01 update_min_a = 0.005

With the experiments it was discovered that very good results can be obtained with just 5–20 particles! The more particles the more accurate location will be but at the cost of additional compute resources. To keep the state updates as frequent as posible it's chosen to keep the particle count down to a minimum.

min_particles = 5 max_particles = 20

Noisy readings from the laser sensor are also discovered. It would at times detect obstacles at short distances when there was nothing there. To prevent those readings to interfere with the localization, a minimum range of lasers is defined.

laser_min_range = 0.4

Finally, the estimate of the initial pose is set to zero to coincide with the location of the robot at start-up.

initial_pose = (0, 0, 0)

TrajectoryPlannerROS — base_local_planner_params.yaml

The trajectory planner is responsible for computing velocity commands to send to the mobile base of the robot given a high-level plan. It seems useful to first understand what was going on under the planner's covers. Enabling the publish_cost_grid_pc parameter allows you to visualize the cost_cloud topic in RViz.

publish_cost_grid_pc: true
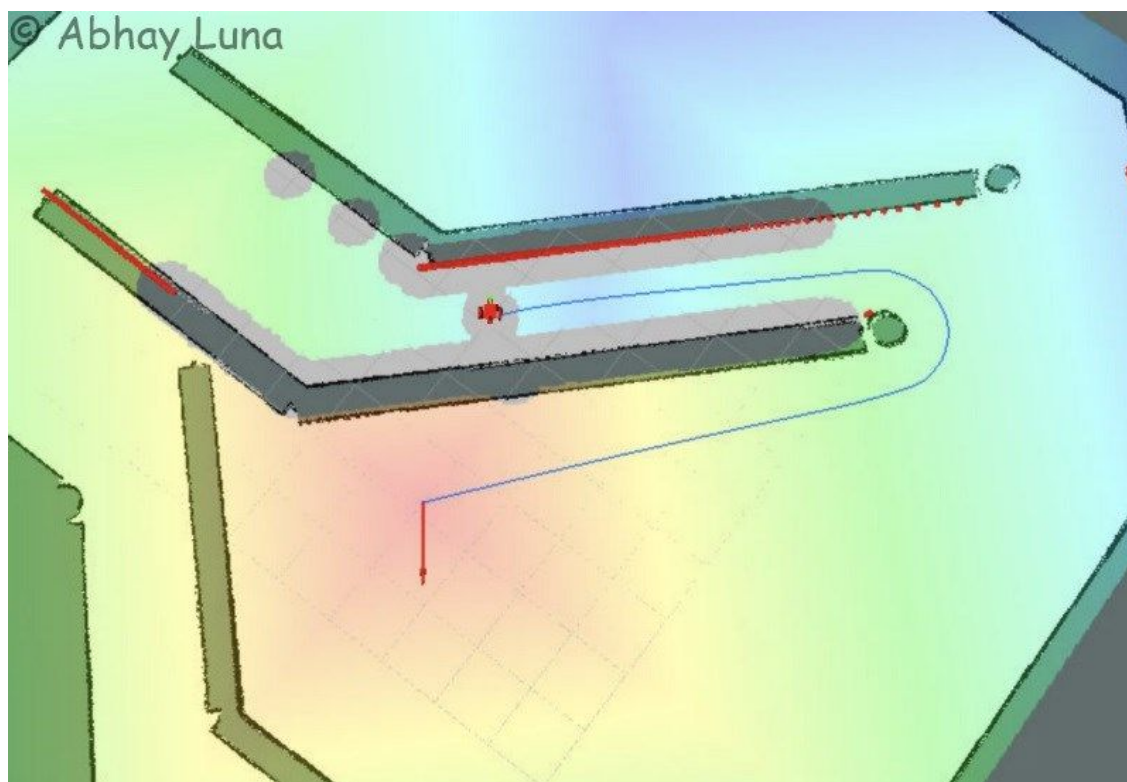
A large costmap is heavily influenced by the navigation goal.

It's detected that the robot was deviating too much from the global path as if attempting to head straight to the goal. Therefore, the following parameters are changed to reduce the goal influence (gdist_scale) and increasing the global path (pdist_scale) compliance.

pdist_scale: 1.0 gdist_scale: 0.4

local_costmap_params.yaml

The local costmap size was by far the most important parameters to get right to successfully navigate around the corner at the end of the corridor. This is because the goal creates a huge influence over the local costmap. This resulted in the robot getting pulled away from the calculated global path. Reducing the size of the local costmap to approximate to the corridor width solved this problem.

width: 5.0 height: 5.0
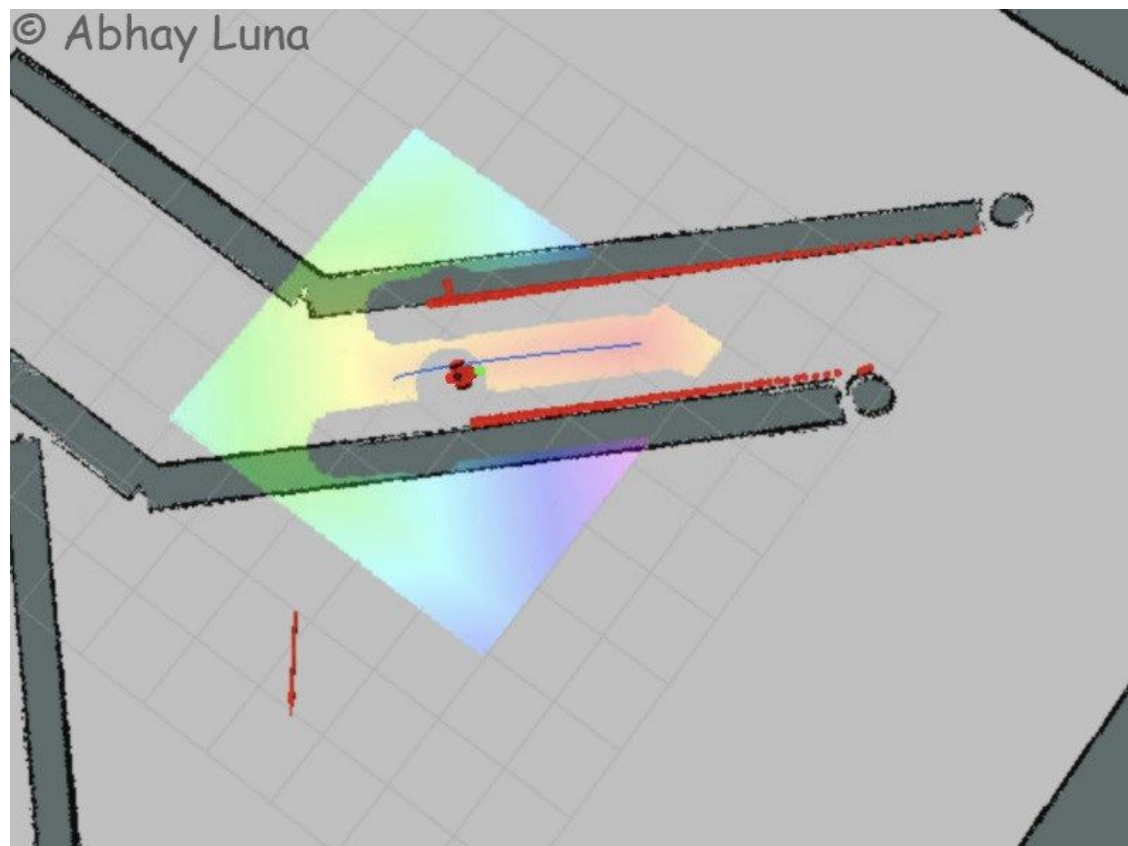
A smaller costmap will produce a gradient along the global path that is not directly affected by the end goal.

costmap_common_params.yaml

The distance range is adjusted because the costmap should be updated based off the laser readings to either add or remove obstacles.

obstacle_range: 4.0 raytrace_range: 4.0

To avoid having the robot bumping on the walls, a radius is defined that conformably fits the robot size. This parameters defines a padding that is added to obstacles. The navigation planer then takes the padding into account when calculating the global path.

inflation_radius: 0.6
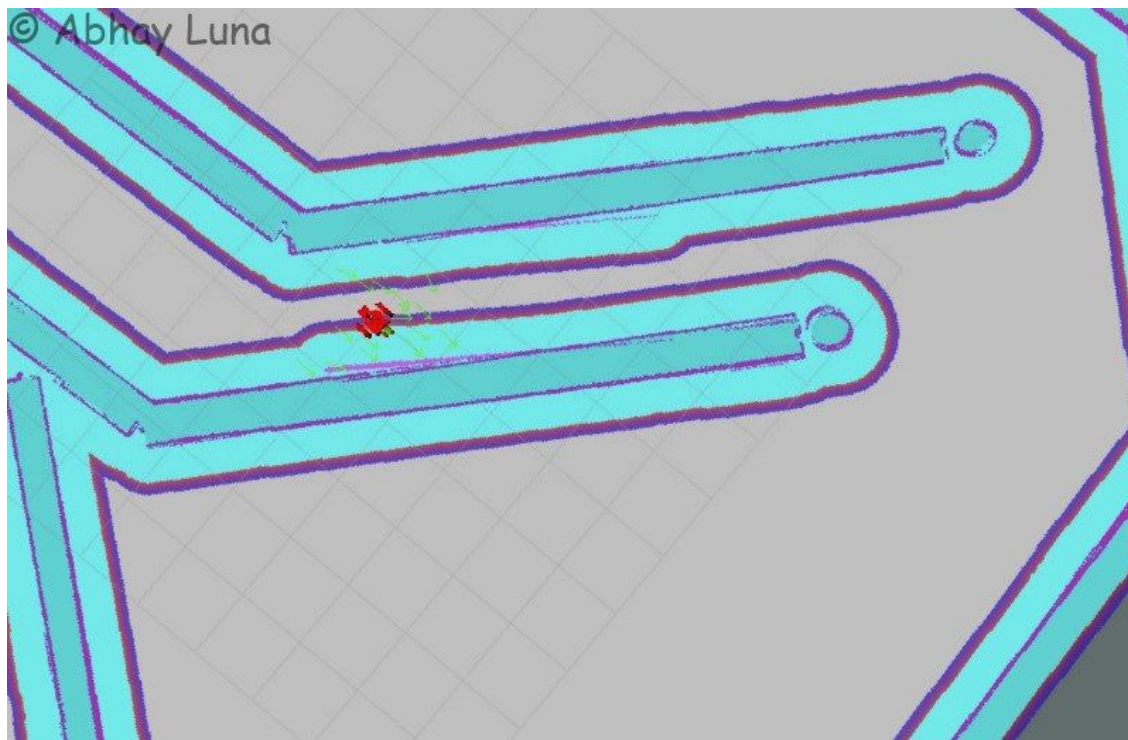
Cost added by the inflation radius to the global costmap acts as a padding around the walls.

The parameters below depend exclusively on the performance of the VM on which the tests were performed. It defines the rate the costmaps should be calculated and published and for how long they are valid. These values have been decreased until no timeout warning messages are received.

transform_tolerance: 0.2 update_frequency: 5.0 publish_frequency: 2.0

Finally, the tolerances of the goal have been lowered to get a very precise position and orientation when reaching the goal:

xy_goal_tolerance: 0.05 yaw_goal_tolerance: 0.01
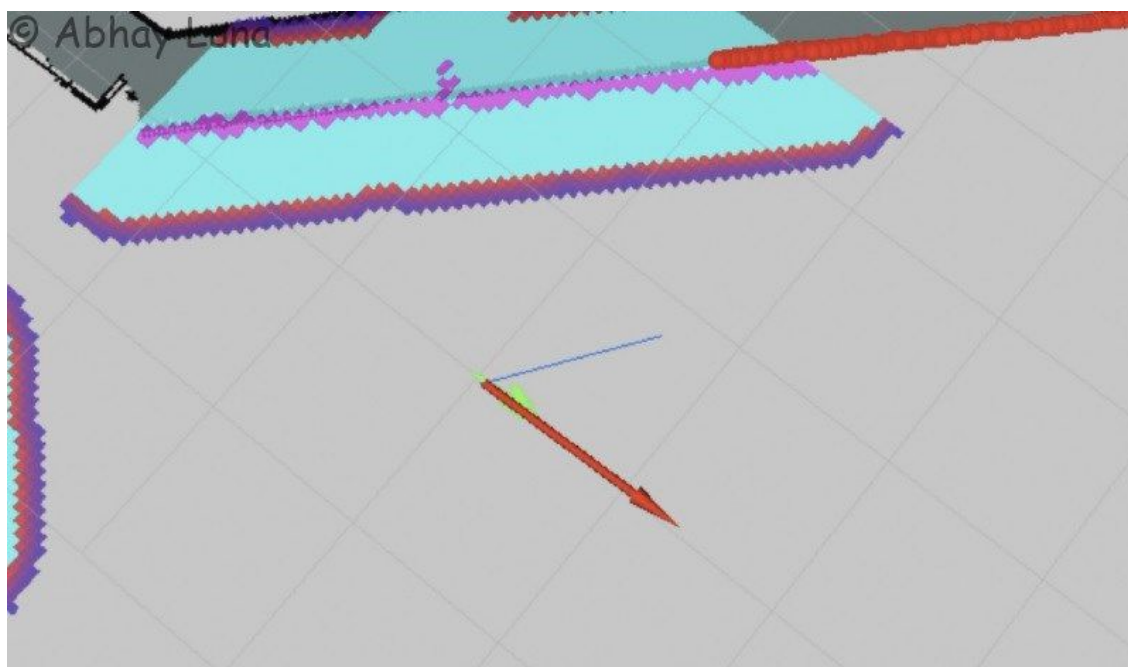
**Low tolerances result in a precise final goal position and orientation.**

Results

Both the classroom and custom robots performed similarly since they have similar mass, size and actuators. In the images below the robot navigate to the position provided by the nodenavigation_goal and it is possible to observe the AMCL particles converging quickly as soon as the robot starts moving. Thanks to the confident location estimate of the robot position, it successfully follows the global path completing the navigation with a precise final position and orientation to the goal.
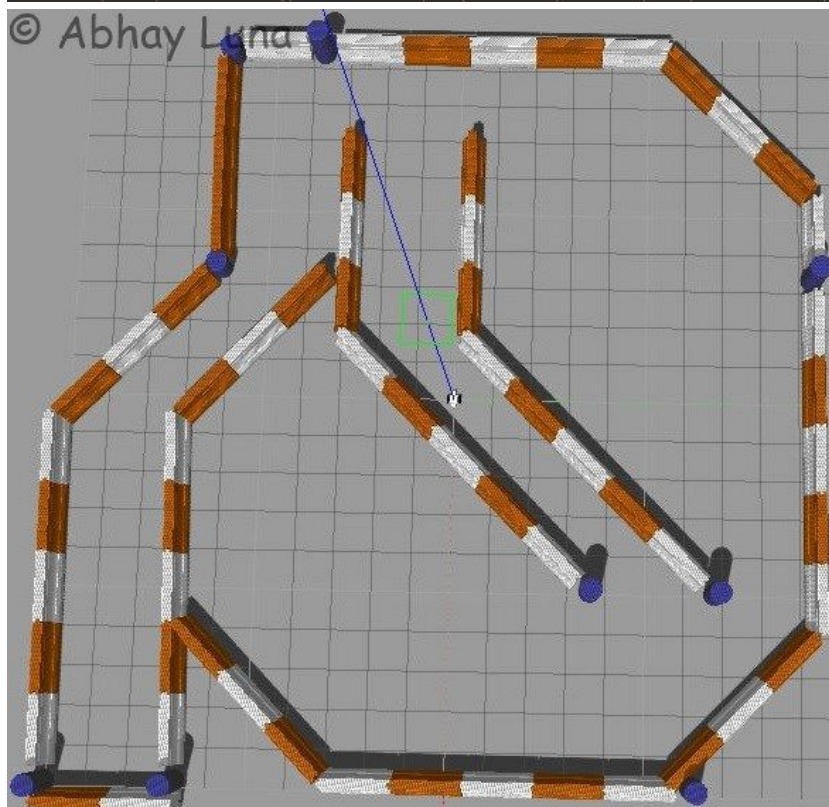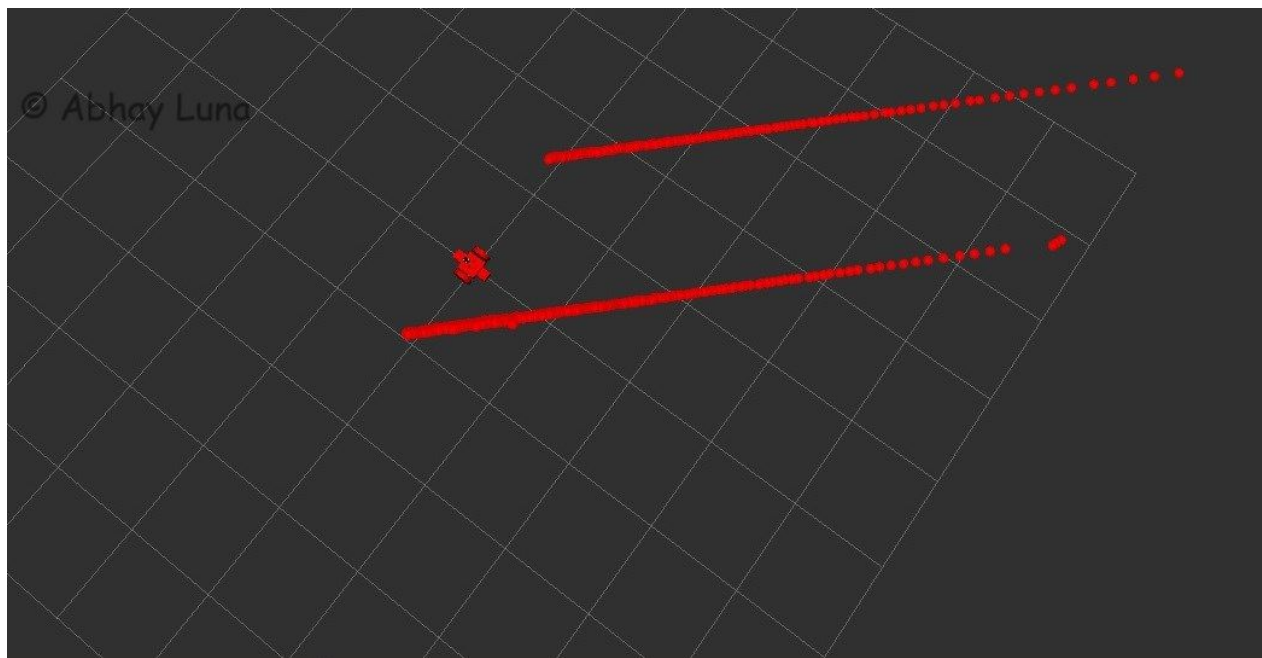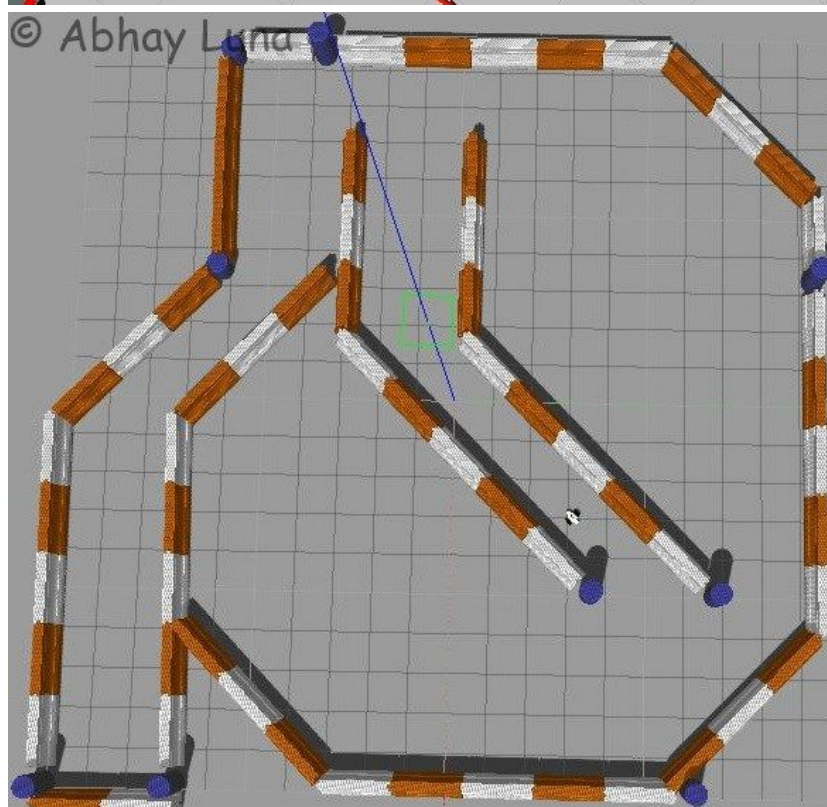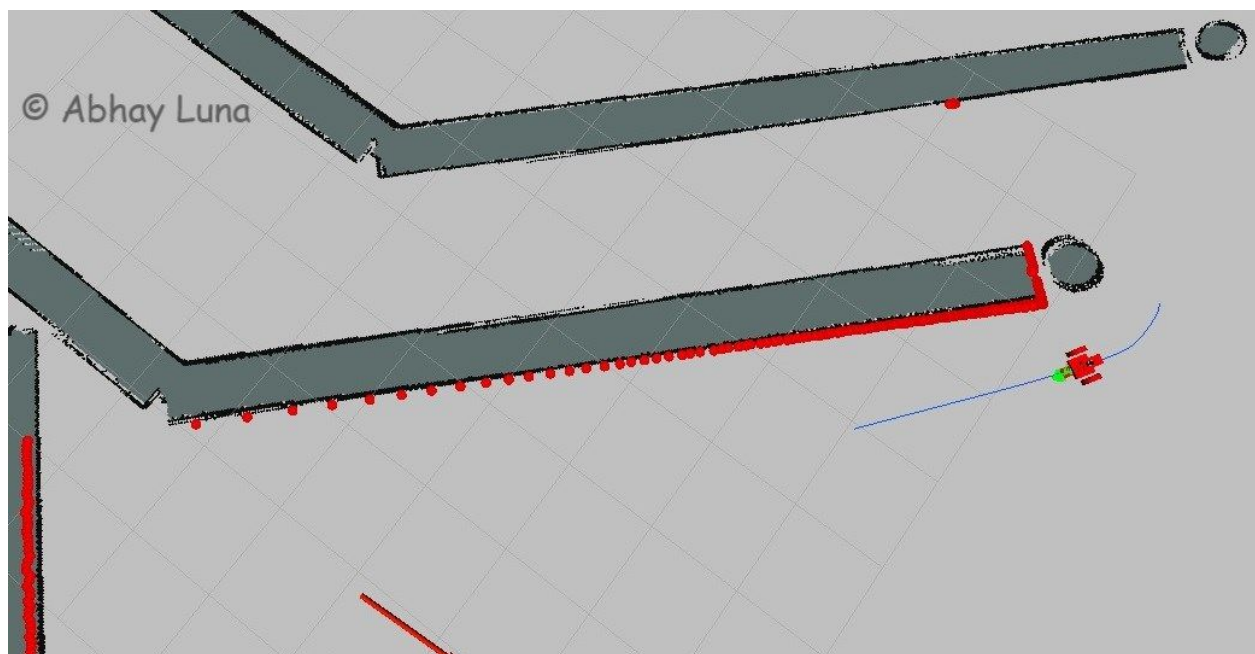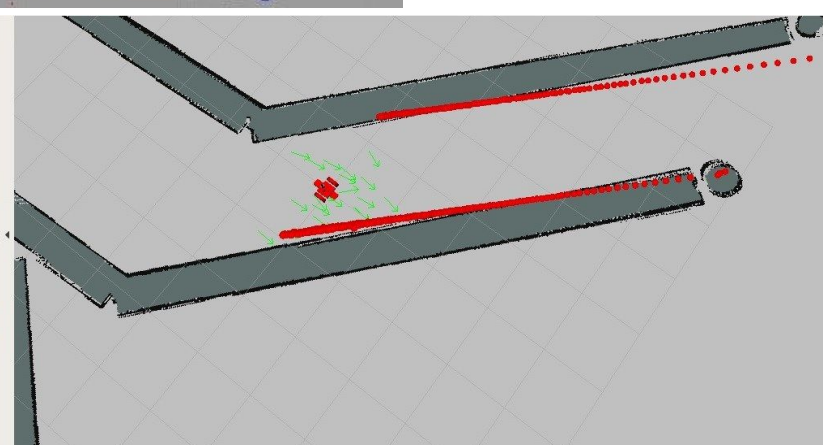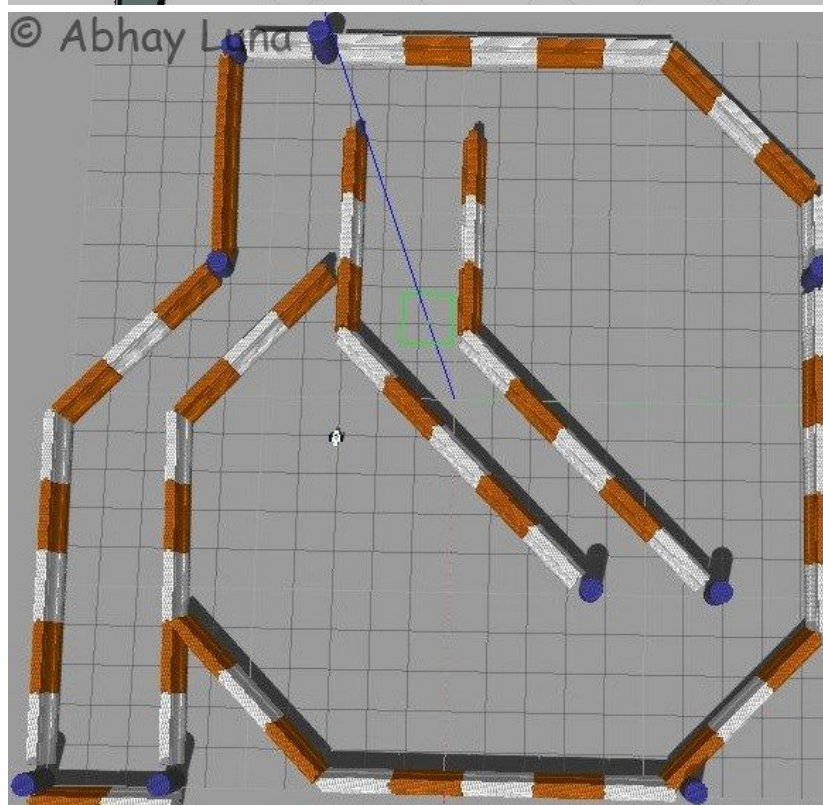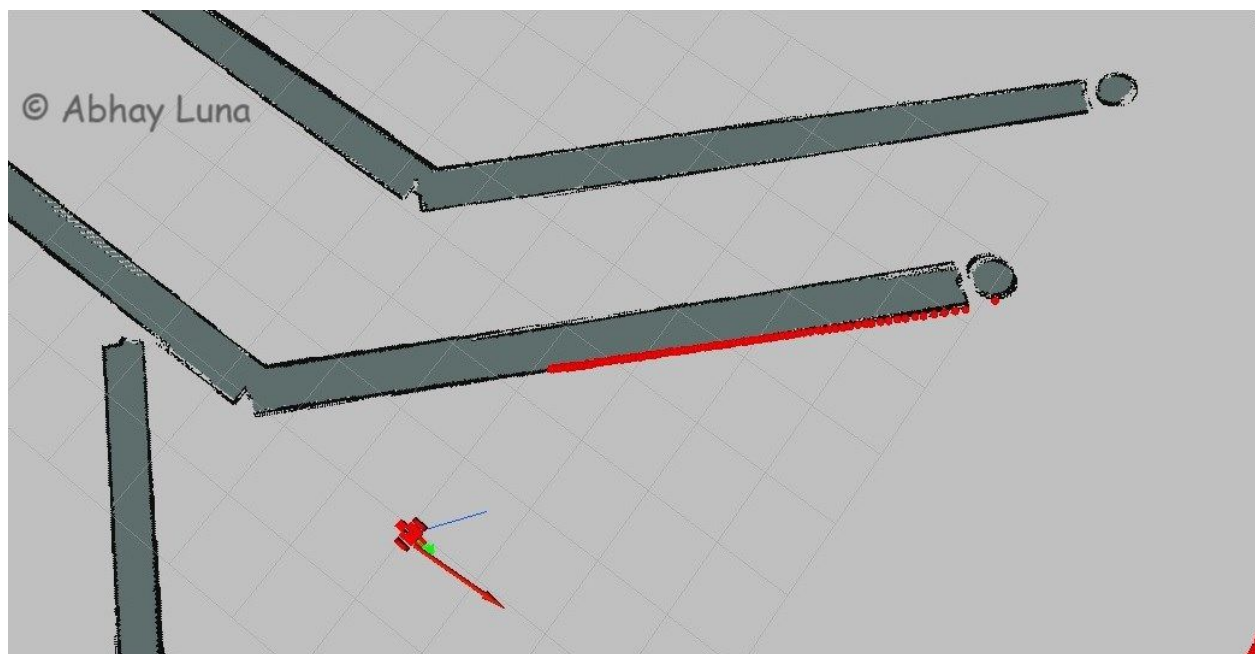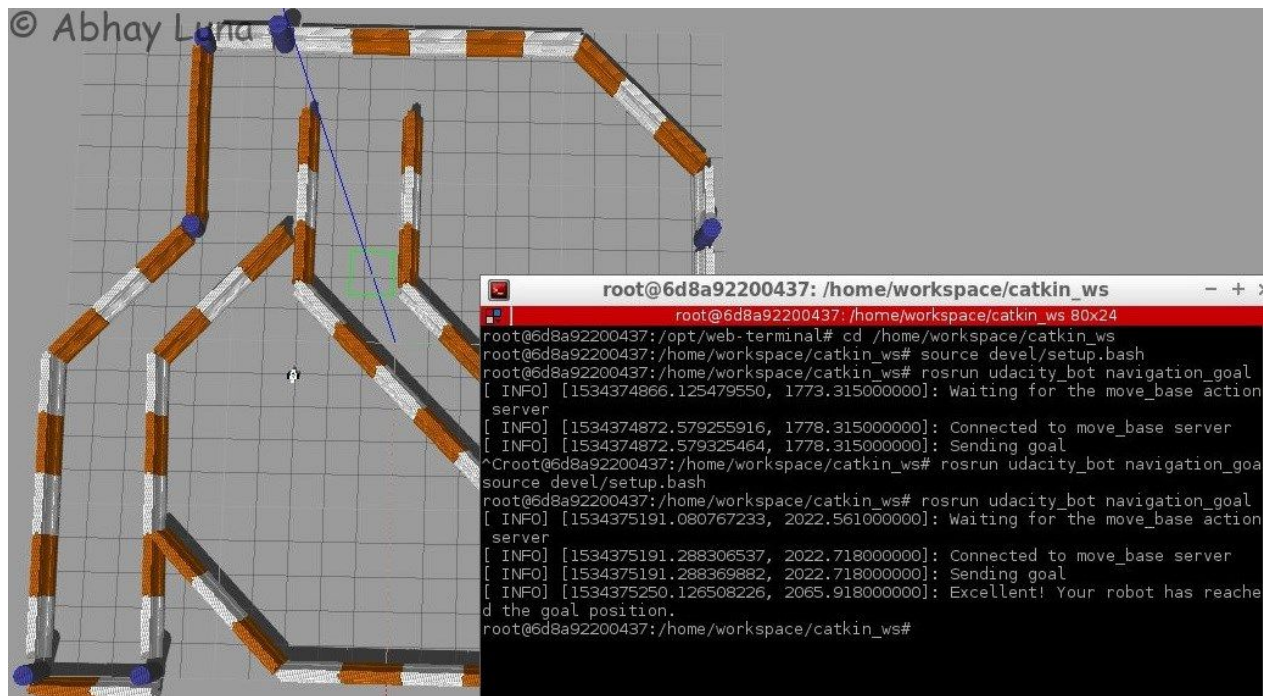
Discussion

The project took a long time, and a lot of help from the Slack channel, to discover that the local costmap size was such a critical parameter. Only after I've reduced it to a reasonable size that it became clear how the remaining parameters work. It was infuriating and embarrassing to see the robot doing the opposite of what was expected.

A second scenario was even more intriguing: after reducing gdist_scale in an attempt to reduce the goal influence over the local plan the robot would rotate in place when a goal was defined immediately across the wall, but would start moving immediately when the goal was moved somewhere else! Again the huge goal influence over the large local costmap was the reason for this surprising behavior.

Also, it is important to point out that the robot would fail to localize when moved to a different location. As discussed previously this is due the fact AMCL is a bayesian algorithms that holds an internal belief of the world that is difficult to change in case of abrupt changes in the environment.

One way to go about moving the robot from place to place would be to reset the AMCL internal state, back to a uniform distribution of particles, when placing the robot on a new location. This would reduce the kidnaped robot to a global localization problem (or "wake-up robot problem" in that context).

Future Work

This project was an excellent introduction to the ROS navigation stack. I would have to explore a variety of packages that process odometry and sensor streams and outputs velocity commands to send to a mobile base. This is a really useful tool that can be applied to all kinds of mobile robots.