Project: Where Am I?

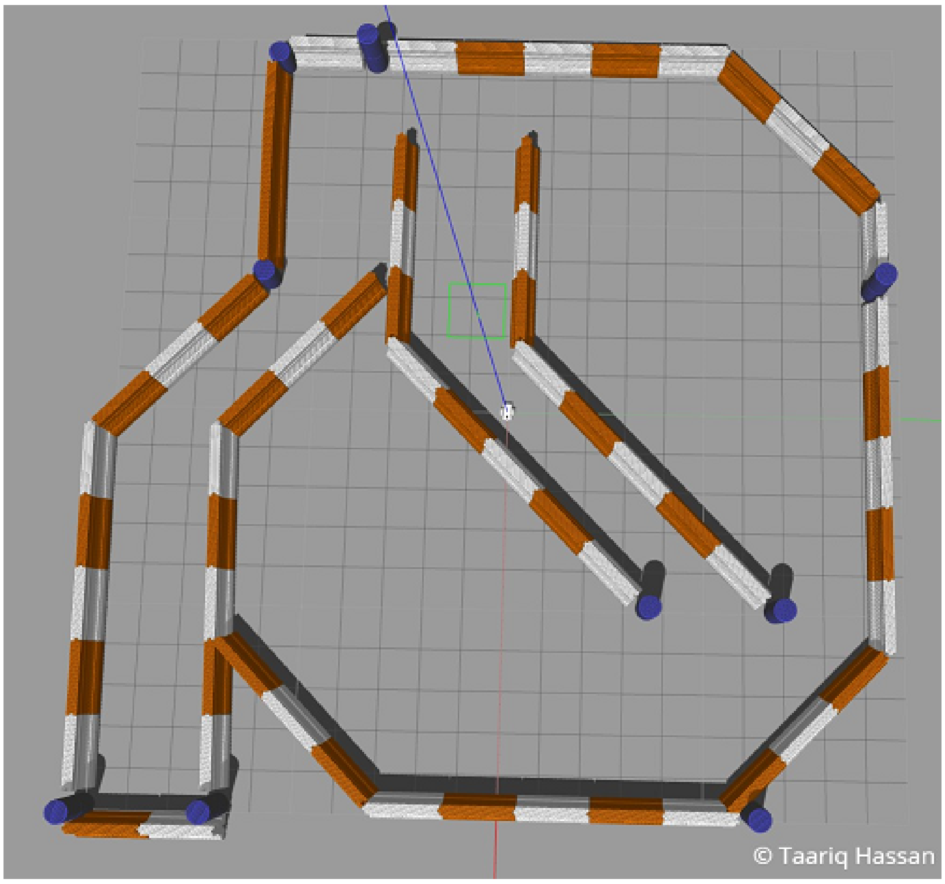
Taariq Hassan (thassan743)

Abstract

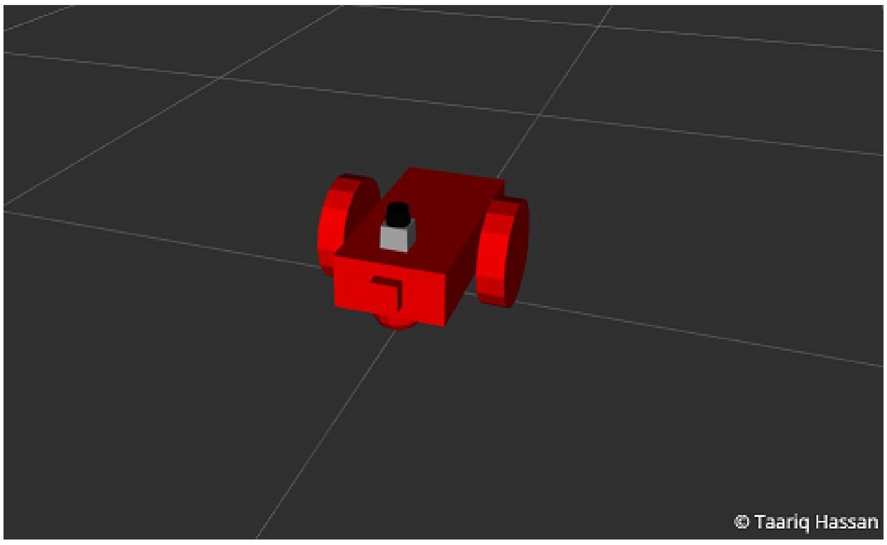
The Localisation project required using the Adaptive Monte Carlo Localisation (AMCL) technique to localise a robot within a simulated ROS environment. The environment with a known map was provided, as well as a sample robot definition. The localisation and navigation parameters were then tuned in order for the robot to reach a goal. Once the sample robot was able to navigate within the environment, a new robot definition was developed. This robot then needed to complete the same task of reaching the goal position with the same, or further tuned, parameters.

Introduction

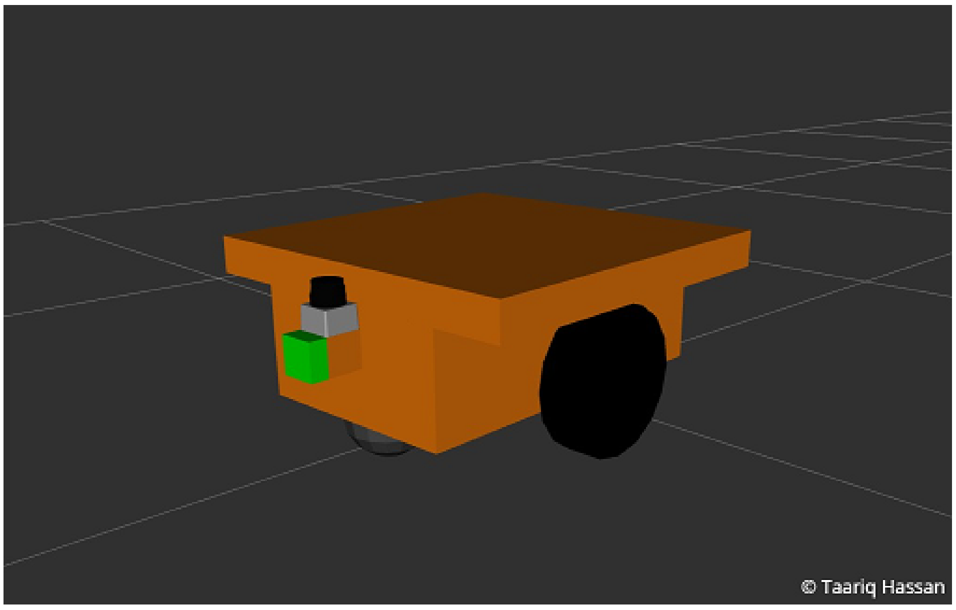
The goal of the Localisation project was to use the Navigation Stack in ROS to enable a simulated robot to localise itself in a given world and navigate towards a goal pose. A map of the environment was provided and can be seen in the figure below.



A robot definition was also provided, called `udacity_bot`, and was simulated in the given environment using Gazebo and RViz. `udacity_bot` can be seen in the figure below.



In order to localise itself, `udacity_bot` came equipped with a LIDAR which could be used with a localisation algorithm, in this case the ROS `amcl` package, to determine an estimate of its pose. Then, using the `move_base` package, the robot could navigate to the goal. However, these packages don't work for every robot configuration and environment by default. Therefore, the main goal of the project was to tune the `amcl` and `move_base` parameters such that `udacity_bot` could successfully reach the goal pose. Thereafter, a second robot definition was developed, called `table_bot`, and simulated in the same environment. `table_bot` can be seen in the figure below and will be described later.



Background

The ability of a robot to localise itself within its environment is extremely important if the robot is to be able to navigate autonomously. In order to perform localisation, a robot uses sensors such as a LIDAR or camera. Based on the outputs from these sensors, a robot can calculate its pose in a known map using a localisation algorithm. However, these sensors are noisy and introduce uncertainty in the calculated pose. Therefore, the localisation algorithm provides an estimate of the robots pose based on the sensor data.

The two most widely used localisation algorithms are the Extended Kalman Filter (EKF) and Adaptive Monte Carlo Localisation (AMCL). The Kalman Filter uses an estimate of a robots state, sensor measurements, as well as knowledge of the noise or uncertainty of these sensors in order to calculate a new estimate of the robots pose. However, the limitation of the Kalman Filter is that it only works with linear systems. The EKF extends the algorithm to work with non-linear systems.

Monte Carlo Localisation on the other hand uses a particle filter in order to estimate the pose of a robot. Particles, which represent a possible pose of the robot, are distributed throughout the known map. Then, based on the results from the robots range sensor, the weight of each particle is adjusted depending on the likelihood of the robot having the same pose as the particle. As the robot moves and more sensor measurements are taken, the particles converge on an estimate of the robots pose.

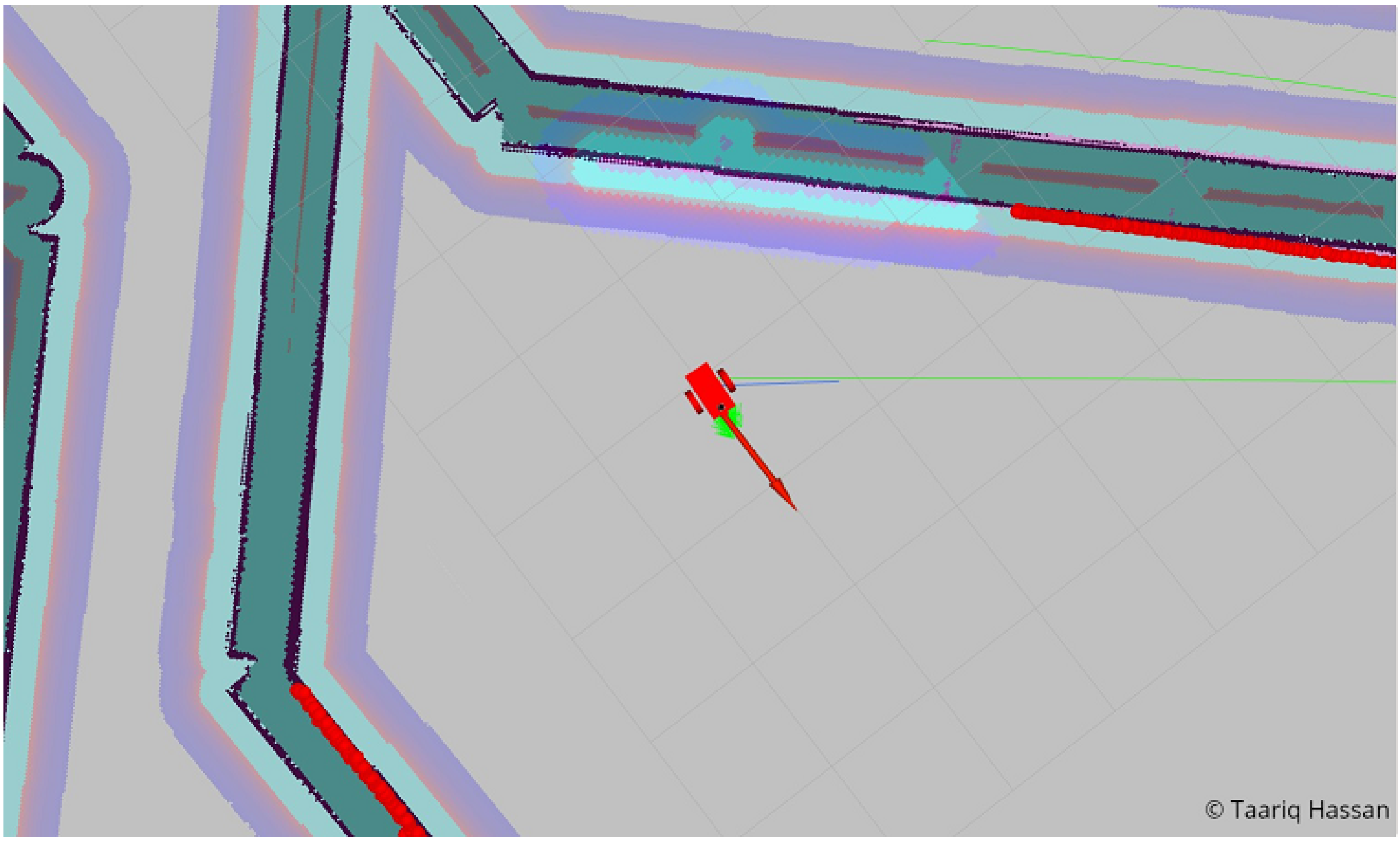
In this project, the AMCL algorithm was used. Some of the advantages that AMCL has over EKF are that AMCL is easier to implement, does not rely on the noise and uncertainty having a Gaussian distribution, and is able to perform global localisation. For these reasons, among others, AMCL is well suited for the problem posed in this project.

Results

As previously discussed, the localisation problem was tested on two different robots. The first robot was called `udacity_bot` and was provided for the project. The second robot was called `table_bot` and was based on `udacity_bot` with some changes which will be discussed in the next section.

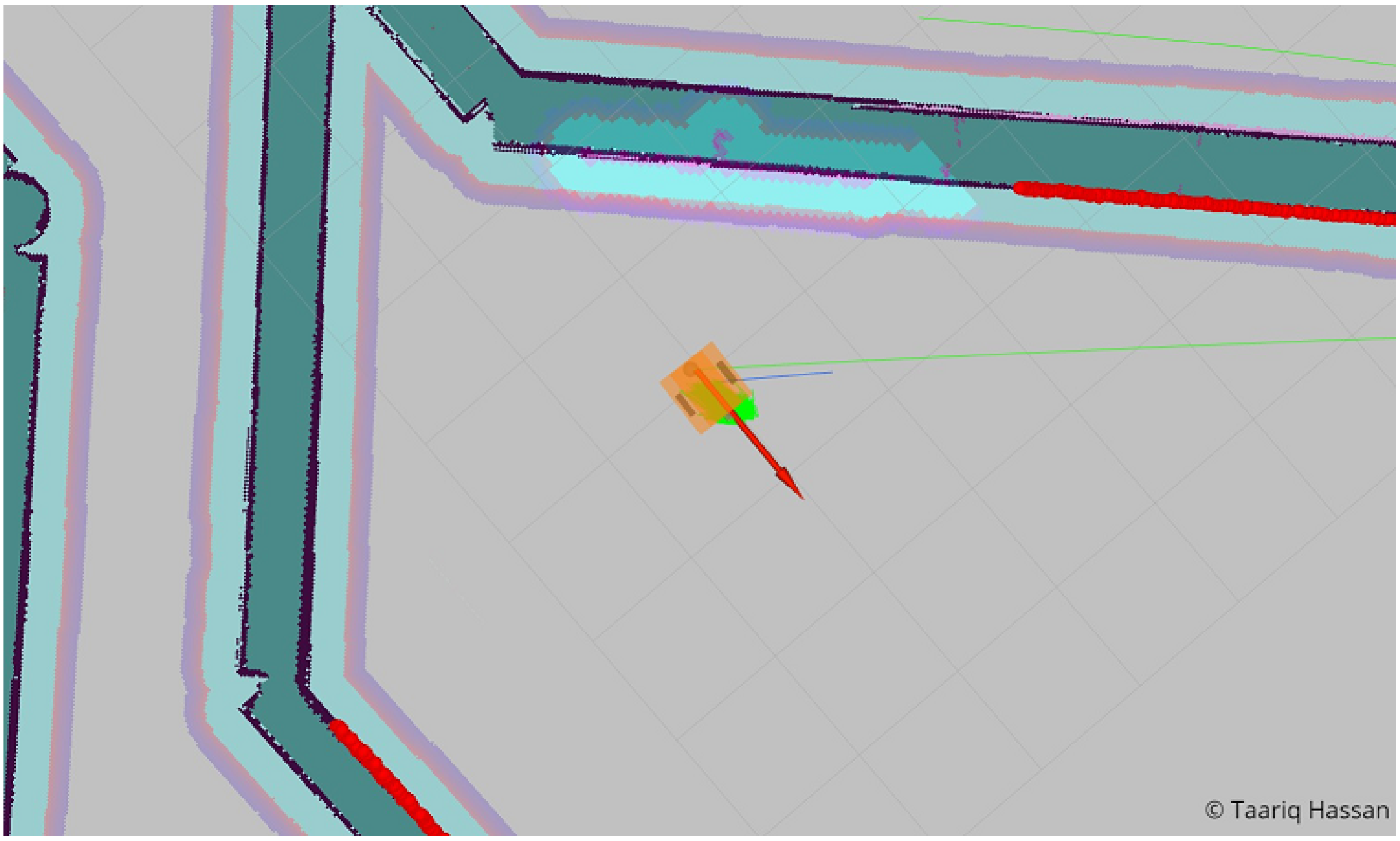
`udacity_bot`

The picture below shows `udacity_bot` located at the goal position. The PoseArray, shown by the green arrows, shows that the particles have converged and provide a good estimate of the robots pose.



`table_bot`

The picture below shows `table_bot` located at the goal position. The PoseArray, shown by the green arrows, shows that the particles have converged and provide a good estimate of the robots pose. The robot model was made partially transparent in order to see the PoseArray.



Model Configuration

`udacity_bot`

`udacity_bot` is a rectangular robot with two driven wheels on either side. There are also two caster wheels underneath the robot in the front and back in order to provide stability. The robot is equipped with a forward facing camera mounted to the front of the robot and a LIDAR mounted on top. The robot configuration is defined in the `udacity_bot.xacro` and `udacity_bot.gazebo` files. A picture of the robot was shown above.

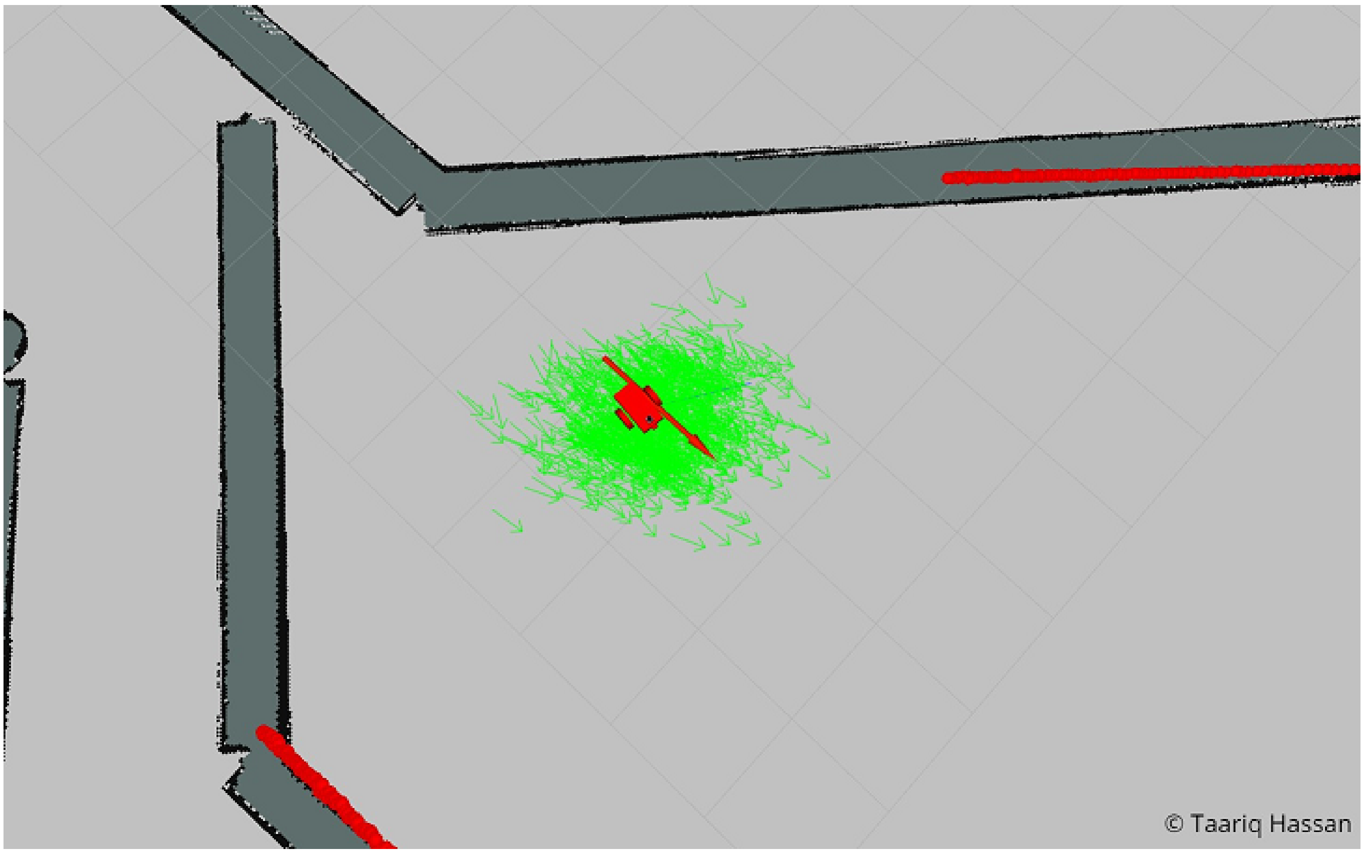
In order for the robot to reach the goal, the parameters of the `move_base` and `amcl` packages needed to be tuned. With the initial configuration, the robot could barely move and the particles in the PoseArray were very sparse indicating a poor estimate of the robots position. There were also warnings in the ROS output stating that the Control loop and Map update loop were missing their desired frequency. The provided parameters in the `costmap_common_params.yaml` file were also set to zero by default.

The first step was to change the default zero parameters in `costmap_common_params.yaml` file to reasonable values. The first two parameters were `obstacle_range` and `raytrace_range` which were set to 2.0 and 5.0 meters respectively. These parameters define how detected obstacles get added to the costmap. The next parameter was the `transform_tolerance` which specifies the amount of delay the system can tolerate. This was set to the costmap package default of 0.2 seconds. The default parameter file came with a declaration of `robot_radius = 0.3`. However, `udacity_bot` is not circular and therefore this parameter was replaced by the `footprint` parameter which allows for a more accurate definition of the size of the robot. This parameter was set based on the robot definition in `udacity_bot.xacro`. The final parameter set was the `inflation_radius` parameter. This parameter defines how far from an obstacle the robot should be before incurring a cost. The `inflation_radius` should be greater than the robots inscribed radius and circumscribed radius which are 0.2 meters and 0.32 meters respectively. Therefore, the `inflation_radius` was set to 0.5 meters to provide enough space from obstacles for the robot to navigate comfortably.

The next step was to try and resolve the control and map update loop frequency warnings. The default controller frequency in the `move_base` package is 20Hz, however the warning message stated that the loop took 0.058s, so the `controller_frequency` parameter was reduced to 10Hz. This was set in the `move_base` node section of the `amcl.launch` file. The map update frequency was initially set to 50Hz in the global and local costmap parameter files. However, the map update loop actually took 0.072s. The `update_frequency` and `publish_frequency` parameters were therefore set to 10Hz in the `global_costmap_params.yaml` and `local_costmap_params.yaml` files.

Once these parameters were changed, it was found that the robot could move, but would drive in the opposite direction from the calculated path. To combat this, the local costmap size was reduced from the default 20mx20m to 5mx5m. This was done by changing both the `width` and `height` parameters in the `local_costmap_params.yaml` file to 5 meters. After this change, the robot would drive in the correct direction and follow the path.

At this point, the robot was able to reach the goal position, however it was found that the particles had not converged well enough on the robots position. This can be seen in the figure below. Therefore, the `amcl` parameters needed to be tuned. Parameters for `amcl` were added to the `amcl` node section of `amcl.launch`. First, `min_particles` and `max_particles` were added and set to 20 and 1000 respectively. This defines the number of particles `amcl` uses when performing localisation. Working with a large number of particles can affect the performance of the simulation so the `max_particles` was reduced from the default of 5000 to 1000. The `transform_tolerance` parameter was also added and set to 0.2 seconds to match the costmap package. Finally, the biggest improvement in performance to the `amcl` algorithm came from adding the odometry model noise parameters `odom_alpha1` through `odom_alpha4`. These were set to the recommended values found in [1].



With these parameters, the robot was able to reach the goal in 2 minutes 23 seconds. The robot at the goal was shown in the results section above.

`table_bot`

`table_bot` was based on `udacity_bot` with some modifications to its design. The idea behind `table_bot` was to have a robot with a large flat surface on to which objects could be placed to be transported. This could be useful in a warehouse, used in conjunction with a robotic arm to load, move and offload objects. The design ensures that no part of the robots structure protrudes above its upper surface, so as not to be damaged and to allow objects larger than the robot to be transported. The LIDAR and camera are therefore mounted in front of the robot. The size of `table_bot` was also made larger than `udacity_bot`. The configuration of `table_bot` is defined in the `table_bot.xacro` and `table_bot.gazebo` files. A picture of the robot was shown above.

`table_bot` was found to work well with only two changes in parameters compared to `udacity_bot`. First, the `footprint` was updated in the `table_bot_costmap_common_params.yaml` file to match the size of `table_bot`. Secondly, the `inflation_radius` was decreased from 0.5 meters to 0.4 meters. Due to the increased size of `table_bot` and the narrow corridor in the map, the robot was not able to navigate down the corridor without incurring a cost with the larger `inflation_radius`. The robot would eventually get stuck in the middle of the corridor. The `inflation_radius` was therefore reduced. The robot was then able to reach the goal in 3 minutes 30 seconds.

Discussion

From the above it can be seen that both robots were able to reach the goal position successfully after tuning parameters of the `move_base` and `amcl` packages. It should be noted however that the performance of the robots are not optimised, as there are many improvements that could still be made. The packages used have an extremely large number of parameters which can be tuned. Most of these parameters were not considered for this project. If these parameters were understood and tuned, it would be possible to further improve on the performance of the robots.

One area of improvement would be to improve how well the robot follows the desired path. With the current implementation, the robot does not follow the path perfectly, especially when going around obstacles. It was found that the robot would move into the inscribed area of the costmap even though the path was on the outside of the inflation radius.

Another area to improve on is finding the goal yaw angle. The robot would often get to the goal position but then oscillate back and forth trying to find the correct yaw angle. It is possible to increase the tolerance on the goal yaw however it would be better to improve the robots ability to navigate to the correct angle.

From the project it was found that the `amcl` package performed well and was able to accurately localise the robot with minimal parameter tuning. On the other hand, most of the time was spent on the `move_base` and navigation aspect of the problem.

AMCL is clearly a very powerful algorithm for localisation within a known environment. It was successfully used to perform global localisation in this project, and did so within a short amount of time. However, AMCL would even work well for the kidnapped robot problem, where a robot, after having localised itself, is moved to a different location without knowing it. In the AMCL algorithm, the environment is sampled every iteration. Therefore, even if the robot has been moved, with the next sensor measurement it will begin to update the previous state to the new state. And since AMCL is able to adjust the number of particles, more particles can be added in order to re-localise quickly. However, this is provided that the robot is still within the same known environment.

This relates well to the idea of `table_bot` being used in a warehouse. Since the warehouse is a known environment with relatively fixed obstacles such as walls or shelves, AMCL would be well suited as a localisation algorithm.

Future Work

As mentioned previously, two areas of improvement would be the path following accuracy and goal yaw angle tracking. Then the plethora of tunable parameters in the packages could be explored in order to extract even more accuracy from the robots. However, attempting to improve accuracy too much may result in a degradation of performance if the hardware is not able to process all the data in time.

One aspect to consider when designing a robot, in simulation or reality, is to design the robot for its intended environment. This was made apparent in this project when simulating `table_bot`, since it was initially unable to drive down the corridor due to its width and the size of the inflation radius. Therefore, the environment (corridor width) and the performance parameters should have factored in to the design of the robot.

Lastly, it's important to remember that at this stage the robots are still operating in a simulated environment, but the goal is to deploy the system to hardware. Therefore, the limitations of the intended hardware must be taken into account.

References

[1] https://answers.ros.org/question/227811/tuning-amcls-diff-corrected-and-omni-corrected-odom-models/