Mobile Robot Simulation of Adaptive Monte Carlo Localization

Cheshta Dhingra

Abstract—In this paper, two different mobile robot models are described and evaluated for performance in a simulated environment. Both simulations are run using Gazebo and RViz. The robots use Adaptive Monte Carlo Localization and the move-base plug in for navigation to successfully arrive at a goal that was either pre-determined. The benchmark model was able to reach its goal in 26 minutes, while the personal model reached it in 35 minutes.

Index Terms—Robot, IEEEtran, Udacity, LaTeX, Localization, Kalman Filter, Particle Filter, Adaptive Monte Carlo Localization

1 Introduction

One of the most integral, yet challenging components of a mobile robot is the ability to navigate through space. In order to successfully navigate, a robot must accomplish 4 sequential tasks. It must first perceive the world around it using sensors and then interpret sensor data to draw meaningful information. Second, the robot must be able to efficiently localize itself, or determine its position and pose within a mapped environment. The third and fourth tasks are cognition and and motor control – the robot must decide its next action and ultimately actuate movement in the desired direction.

Localization has received much attention over the past decade, and will be the subject of this paper. There are 3 types of localization problems: local localization, in which the robot knows its initial pose and must determine its position as it moves around the environment; global localization , in which the robots initial pose is unknown, and the robot must determine its pose relative to the ground truth map, and the ”kidnapped robot problem”, which is similar to global localization, except that the robot may be ”kidnapped” and set to a new location on the map at any point.

However, one of the primary challenges in the field is that of sensor noise, or uncertainty. For example, when using sonar, the transducer may emit sound towards an angled surface. While some of this energy will return an echo and the sonar will succeed, some of the energy may reflect away and fail to generate an echo. Thus, depending on the gain threshold of the sonar, the sensor may or may not detect the object, and the same environment can result in 2 different readings. Two popular localization algorithms that aim to overcome this uncertainty by filtering the noise are the Extended Kalman Filter Localization Algorithm (EKF) and the Monte Carlo Localization Algorithm (MCL). In this paper, through the use of MCL as well as sensor fusion, by which inputs from multiple sensors are taken into account, sensor noise will be minimized to solve the localization problem.

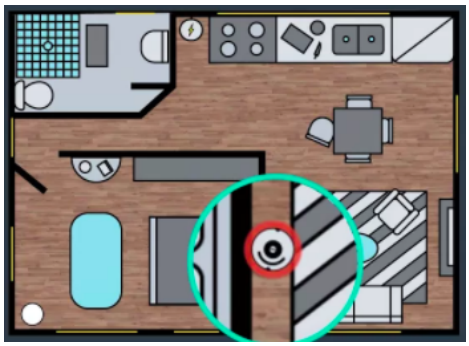

Fig. 1. The Local Localization problem, also known as position tracking.

2 Background

In this paper, both Kalman Filters and the Monte Carlo Localization algorithms are explored, and ultimately the Adaptive MCL is implemented to localize two different mobile robots in simulation. Through experimentation, the various parameters of the algorithm are tuned to optimize localization. This section elaborates on the merits of both algorithms.

2.1 Kalman Filters

The Kalman filter (KF) is one of the most practical algorithms in the field of controls engineering. It is a common filter used to estimate the state of a system when the measurements are noisy. Notably, it can estimate the value of a variable in real time as data is being collected. Unlike other estimation filters, the Kalman filter does not require a lot of data in order to make an accurate estimation, making it very useful for localization applications.

The Kalman filter makes an initial prediction of the state, taking into account the expected uncertainty of a sensor or movement by exploiting the Gaussian nature of this uncertainty. The variable could be the position or velocity of the robot. Then, the KF makes a measurement update by collecting sensor data, which also has a Gaussian distribution. The KF then takes a weighted sum of the prior

belief and measurement means to generate an updated mean. The new mean is calculated such that it is biased towards the measurement update, which has a smaller standard deviation than the prior. Importantly, the two Gaussians provide more information together than either Gaussian offered alone. As a result, the new state estimate is more confident than both the prior belief and the measurement. This means that it has a higher peak and is narrower.

Next, the robot executes a command that puts it into motion. This motion, however, also follows a Gauusian distribution. The state prediction step simply sums the means of the prior belief (that was generated from the measurement update step) and the motion executed to attain a posterior mean. A similar process generates the posterior variance, to ultimately achieve a posterior state prediction. The KF is ready for the next iteration of the measurement update.

The linear, or standard Kalman filter can be applied to linear systems, or ones in which the input is proportional to the output. However, in the field of robotics the Extended Kalman Filter (EKF) is more useful, as it can be applied to non-linear systems as real-world systems are more often non-linear than linear. For example, when a robot moves in a circle, the linear KF is quite limiting. Although the mean can be updated despite the motion or measurement functions being non-linear, the variance cannot. However, EKF can be used to apply linearization to a non-linear function using the Taylor series and Jacobians (in the case of multi-dimensional EKF), such that locally linear approximations are used to update the variance.

2.2 Particle Filters

In robotics, a particle can be conceptualized as a virtual element that resembles the robot. Each particle has a position (x and y coordinates) and orientation (theta with respect to the global coordinate frame), and represents a guess of where the robot might be located. Additionally, each particle has a weight component, which represents the difference between the robot's actual pose, and the particle's predicted post. In Monte Carlo Localiztion (MCL), particles are initially spread randomly and uniformly throughout the map. These particles are re-sampled each time the robot moves and senses its environment. Particles with larger weight are more likely to survive the re-sampling process. After several iterations of MCL, particles converse and estimate the robot's pose. Thus, the loop of the MCL goes as follows: prior belief, motion update, measurement update, resampling, new belief. In this project, the Adaptive Monte Carlo Localization algorithm is used, in which the number of particles is dynamically adjusted over a period of time.

Although the Kalman filter is an elegant and efficient localization algorithm and can provide a very accurate estimate of the robot state using onboard sensors, it does so by exploiting a range of restrictive assumptions - such as Gaussian-distributed noise and Gaussian distributed initial uncertainty. It ultimately represents posteriors as Gaussians. The restrictive nature of the belief representation makes plain Kalman filters inapplicable to global localization problems.

However, a particle filter like MCL solves the global localization and kidnapped robot problem in a highly robust and efficient way. It can accommodate arbitrary noise distributions (and non-linearities). Thus, MCL avoids a need to extract features from the sensor data.

2.3 Comparison / Contrast

There are several key differences between the EKF and MCL algorithms, as well as some similarities. While the Extended Kalman Filter relaxes some of the restraints that limit the Kalman Filter, they do so at a great computational cost and complexity. Further, the Monte Carlo Localization Algorithm does not have many of these constraints. For example:

- MCL is easier to code than EKF
- MCL can be used to represent non-Gaussian distributions while EKF is restricted to Gaussian distributions (which is not always applicable to the real world)
- in MCL, the computational memory and resolution of the solution can be controlled by changing the number of particles distributed throughout the map

The table below provides a full comparison of the two algorithms.

	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency(memory)	✓	✓✓
Efficiency(time)	✓	✓✓
Ease of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	x
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodal Continuous

Fig. 2. Comparing and Contrasting MCL and EKF

The work presented here will be using only particle filters, and specifically, the Adaptive MCL.

3 Simulations

Simulations were carried out in a Gazebo environment with the use of RViz for visualization of different aspects of the mobile robot and world. The following sections will discuss the performance of each robot individually, in addition to a description of the robot model design, packages used, and the parameters chosen for the robot to properly localize itself. The performance of the robots is evaluated through simulations, run using Gazebo and Rviz.

3.1 Achievements

The benchmark model was able to successfully reach the goal position in about 24 minutes, while chesh-bot reached it in about 35 minutes. During the course of navigation, both robots were fairly accurate in following the global path, and there were no major deviations. In both cases, the terminal output confirmed that the robot had reached it's goal.

3.2 Benchmark Model

3.2.1 Model design

The design of the benchmark model including all of it's features is described in Table 1 below. Fig. 3 is an image of the benchmark robot model.

TABLE 1
Benchmark Robot Model Design

Feature	Description
Base Shape	Cuboidal
Base Size	0.4 x 0.2 x 0.1
Castor Wheel Shape	Spherical (x2)
Csstor Wheel Radius	0.5
Wheel Shape	Cylindrical (x2)
Wheel Radius	0.1
Wheel Length	0.05
Wheel Mass	5
Sensor Types	Camera, Hokuyo Laser Rangefinder
Camera Shape	Box
Camera Size	0.05
Camera Mass	0.1
Hokuyo Shape	Box
Hokuyo Size	0.1
Hokuyo Mass	0.1
Hokuyo Position	0.15 0 0.1
Plugins	Differential Drive Controller

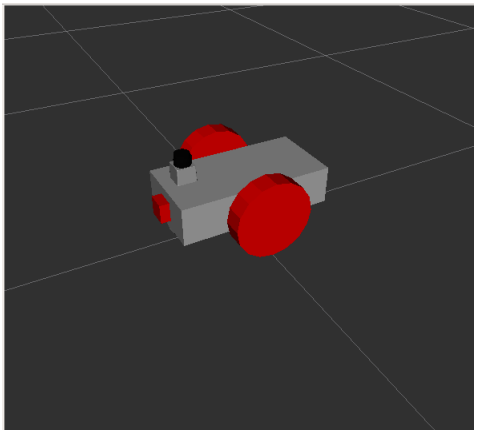

Fig. 3. Benchmark Robot Model

3.2.2 Worlds

There are 2 files that define the world. The udacity_world file defines the ground plane, a light source, and a camera. The jackal_race world defines the maze.

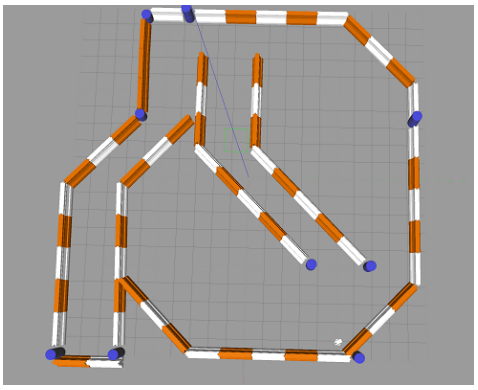

Fig. 4. The jackal_race maze created by Clearpath Robotics

3.2.3 URDF

- Udacity_bot.gazebo: - defines the camera sensor plugin, Hokuyo laser plugin and the differential drive controller.
- Udacity_bot.xacro: defines the physical features of the robot model.

3.2.4 Maps

Clearpath Robotics created a map, which was utilized in this project. jackal_race.yaml and jackal_race.pgm define the maps.

3.2.5 Meshes

The Hokuyo rangefinder is the lasar rangefinder utilized in this project. The holuyo.dae mesh can be found in this folder.

3.2.6 Launch

Launch files in ROS allow the execution of more than one node simultaneously, which helps avoid a potentially tedious task of defining and launching several nodes in separate shells or terminals. Three launch files are used:

1) udacity_world.launch - launches the robot_description launch file as well as the AMCL localization server and the jackal_race world. It also launches RViz and spawns the robot in Gazebo.
2) amcl.launch - launches the map server, places the map frame at the odometry frame, launches the localization AMCL package, the move_base package, and the tf package
3) robot_description.launch - launches joint_state publisher to send fake joint values, the robot_state publisher which sends robot states to tf and the robot_description which sends URDF to the param server.

3.2.7 Packages Used

The packages used in the project are specified below:

- Xacro - this is used to create large XML files such as the robot description files.
- ROS - the entire set of launch files, configuration files, maps, meshes, urdfs and worlds is contained in 2 ROS packages - one for each model.

- AMCL - this is the main localization package. It specifies parameters for the overall filter, laser and odometry to localize the robot.
- Move Base - this package helps navigate the robot to the goal position by creating or calculating a path from the initial position to the goal.

3.2.8 Parameters

Localization parameters in the AMCL node are described below, as well as move_base parameters in the configuration file.

```
  max_particles: 200             # (int, default 5000) Maximum allowed
number of particles
  min_particles: 25              # (int, default 100) Minimum allowed number
of particles
  kld_err: 0.01                  # (double, default 0.01) Maximum error
between the true distribution and the estimated distribution.
  kld_z: 0.99                    # (double, default 0.99) Upper standard
normal quantile for (1 - p), where p is the probability that the error on
the estimated distrubition will be less than kld_err.
  transform_tolerance: 0.2       # (double, default 0.1 seconds) Time with
which to post-date the transform that is published, to indicate that this
transform is valid into the future.
  recovery_alpha_slow: 0.001     # (double, default 0.0 (disabled))
Exponential decay rate for the slow average weight filter, used in deciding
when to recover by adding random poses. A good value might be 0.001.
  recovery_alpha_fast: 0.1       # (double, default 0.0 (disabled))
Exponential decay rate for the fast average weight filter, used in deciding
when to recover by adding random poses. A good value might be 0.1.
  initial_pose_x: 0.0
  initial_pose_y: 0.0
  initial_pose_a: 0.0
  use_map_topic: true            # (bool, default false) When set to true,
AMCL will subscribe to the map topic rather than making a service call to
receive its map.
```

Fig. 5. AMCL Filter Parameters

```
  laser_min_range: -1.0     # Minimum scan range to be considered; -1.0
will cause the laser's reported minimum range to be used.
  laser_max_range: -1.0     # Maximum scan range to be considered; -1.0
will cause the laser's reported minimum range to be used.
  laser_max_beams:  30      # (int, default 30) How many evenly-spaced
beams in each scan to be used when updating the filter.
  laser_z_hit:      0.95    # (double, default 0.95) Mixture weight for the
z_hit part of the model.
  laser_z_short:    0.1     # (double, default 0.1) Mixture weight for the
z_short part of the model.
  laser_z_max:      0.05    # (double, default 0.05) Mixture weight for the
z_max part of the model.
  laser_z_rand:     0.05    # (double, default 0.05) Mixture weight for the
z_rand part of the model.
  laser_sigma_hit: 0.2      # (double, default 0.2 meters) Standard
deviation for Gaussian model used in z_hit part of the model.
  laser_lambda_short: 0.1   # (double, default 0.1) Exponential decay
parameter for z_short part of model.
  laser_likelihood_max_dist: 2.0 # (double, default 2.0 meters) Maximum
distance to do obstacle inflation on map, for use in likelihood_field model.
  laser_model_type: "likelihood_field_prob" # (string, default
"likelihood_field") Which model to use, either beam, likelihood_field, or
likelihood_field_prob (same as likelihood_field but incorporates the
beamskip feature, if enabled).
```

Fig. 6. AMCL Laser Parameters

```
  odom_model_type: "diff-corrected"   # (string, default "diff") Which
model to use, either "diff", "omni", "diff-corrected" or "omni-corrected".
  odom_alpha1: 0.005                  # (double, default 0.2) Specifies the
expected noise in odometry's rotation estimate from the rotational
component of the robot's motion.
  odom_alpha2: 0.005                  # (double, default 0.2) Specifies the
expected noise in odometry's rotation estimate from translational component
of the robot's motion.
  odom_alpha3: 0.005                  # (double, default 0.2) Specifies the
expected noise in odometry's translation estimate from the translational
component of the robot's motion.
  odom_alpha4: 0.005                  # (double, default 0.2) Specifies the
expected noise in odometry's translation estimate from the rotational
component of the robot's motion.
  odom_frame_id: "odom"               # (string, default "odom") Which
frame to use for odometry.
  base_frame_id: "base_link"          # (string, default "base_link") Which
frame to use for the robot base
  global_frame_id: "map"              # (string, default "map") The name of
the coordinate frame published by the localization system
  tf_broadcast: true                  # (bool, default true) Set this to
false to prevent amcl from publishing the transform between the global
frame and the odometry frame.
```

Fig. 7. AMCL Odometry Parameters

3.2.9 Tuning

This section will provide an overview of the parameter tuning process for a select set of important parameters. ROS documentation as well as Kaiyu Zheng's ROS Navigation Tuning Guide were consulted during this process.

```
#
# Trajectory Planner base local planner params
#
TrajectoryPlannerROS:

  holonomic_robot: false        # For holonomic robots, strafing velocity
commands may be issued to the base. For non-holonomic robots, no strafing
velocity commands will be issued.
  controller_frequency: 10.0    # (double, default 20.0)
  oscillation_reset_dist: 0.1   # (double, default 0.05) How far the robot
must travel in meters before oscillation flags are reset
  escape_vel: -0.4              # (double, default -0.1) Speed used for
driving during escapes in meters/sec. Note that it must be negative in
order for the robot to actually reverse.
  heading_scoring_timestep: 1.2 # (double, default: 0.8) How far to look
ahead in time in seconds along the simulated trajectory when using heading
scoring
```

Fig. 8. Base Local Planner Parameters

```
map_type: costmap

obstacle_range: 3.0     # (double, default 2.5) The default maximum
distance from the robot at which an obstacle will be inserted into the cost
map in meters. This can be over-ridden on a per-sensor basis.
raytrace_range: 7.0     # (double, default 3.0) The default range in meters
at which to raytrace out obstacles from the map using sensor data. This can
be over-ridden on a per-sensor basis.
observation_sources: laser_scan_sensor
laser_scan_sensor:   {sensor_frame: hokuyo, data_type: LaserScan, topic: /
udacity_bot/laser/scan, marking: true, clearing: true}
```

Fig. 9. Move_base Common Costmap Parameters

```
#
# Trajectory Planner local cost map parameters
#
local_costmap:
   global_frame: odom
   robot_base_frame: robot_footprint
   update_frequency: 10.0              # (double, default 5.0) The
frequency in Hz for the map to be updated.
   publish_frequency: 10.0             # (double, default 0.0) The
frequency in Hz for the map to be publish display information.
   width: 5.0                          # The width of the map in meters.
(int, default 10) The width of the map in meters.
   height: 5.0                         # (int, default 10) The width of the
map in meters.
   resolution: 0.05                     # (double, default 0.05) The
resolution of the map in meters/cell.
   static_map: false
   rolling_window: true                 # Whether or not to use a rolling
window version of the costmap. If the static_map parameter is set to true,
this parameter must be set to false.
```

Fig. 10. Move_base Local Costmap Parameters

```
global_costmap:
   global_frame: map                   # (default map) The global frame for
the costmap to operate in.
   robot_base_frame: robot_footprint   # (default base_link) The name of
the frame for the base link of the robot.
   update_frequency: 10.0              # (double, default 5.0) The
frequency in Hz for the map to be updated.
   publish_frequency: 10.0             # (double, default 0.0) The
frequency in Hz for the map to be publish display information.
   width: 30.0                         # (int, default 10) The width of the
map in meters.
   height: 30.0                        # (int, default 10) The height of
the map in meters.
   resolution: 0.02                    # (double, default 0.05) The
resolution of the map in meters/cell. Hokuyo URG-04LX-UG01 laser scanner
has metric resolution of 0.01mm
   static_map: true
   rolling_window: false               # (bool, default false) Whether or
not to use a rolling window version of the costmap. If the static_map
parameter is set to true, this parameter must be set to false.
```

Fig. 11. Move_base Global Costmap Parameters

AMCL Filter Parameters

- The number of particles was reduced to between 25 and 200 down from 5000 to suit the computational power of the system.
- The transform tolerance was an important parameter that was raised to 0.1 from the default 0.1. This was done so that the ultimate value was larger than the chosen update frequency of 10Hz.
- The recovery alpha slow and fast values were set based on the recommendations of 0.001 and 0.1, respectively.
- The initial pose was set to (0, 0, 0)

AMCL Laser Parameters

These parameters were left at default, as they seemed to work well without modification.

AMCL Odometry Parameters

The odom parameters were set by trial and error to be a very small number (0.005). These parameters specify th expected noise in odometry from different components of the robot's motion. Since in this simulation environment the expected noise was very low, this approach seemed to work well.

Base Local Planner Params

- Since differential drive robots are nonholonomic, the holonomic_robot parameter was left to be false.
- controller_frequency was adjusted to half the default number, from 20 to 10 Hz to simply meet the needs of the hardware.
- Similarly, oscillation_reset_dist was set to double the default value, from 0.05 to 0.1
- The escape_vel was raised to allow the mobile robot to more easily reverse when it got stuck.

Move_base Common Costmap Parameters

- obstacle_ange was set to 3.0 to increase the maximum distance from the robot at which an obstacle will be added to the costmap
- Similarly, raytrace_range was set to 7.0

Move_base Local Costmap Parameters

- update_frequency was reduced iteratively to 10 Hz to match the hardware computational requirements
- Size was set to 5m x 5m. Decreasing the size of the local cost map utilized significant less computational power and solved the problem of missing the publish_frequency window.

Move_base Global Costmap Parameters

- update_frequency and publish_frequency wereset to 10 Hz, as above.
- Size was set to 30m x 30m
- Resolution was set to 0.2

3.3 Personal Model

The design of the personal model - chesh_bot, including all of it's features is described in Table 2 below, followed by an image of the personal robot model.

3.3.1 Worlds

Worlds for chesh_botare identical to those used for Udacity_bot.

3.3.2 URDF

- chesh_bot.gazebo: - defines the camera sensor plugin, Hokuyo laser plugin and the differential drive controller.
- chesh_bot.xacro: defines the physical features of the robot model.

TABLE 2
Benchmark Robot Model Design

Feature	Description
Base Shape	Cuboidal
Base Size	0.4 x 0.5 x 0.1
Castor Wheel Shape	Spherical (x2)
Csstor Wheel Radius	0.5
Wheel Shape	Cylindrical (x2)
Wheel Radius	0.1
Wheel Length	0.05
Wheel Mass	5
Sensor Types	Camera, Hokuyo Laser Rangefinder
Camera Shape	Box
Camera Size	0.05
Camera Mass	0.1
Hokuyo Shape	Box
Hokuyo Size	0.1
Hokuyo Mass	0.1
Hokuyo Position	0.165 0 0.1
Plugins	Differential Drive Controller

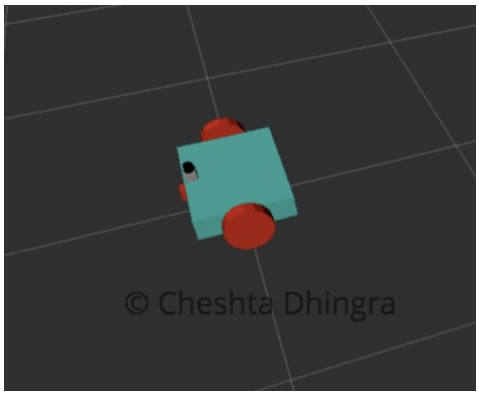

Fig. 12. Personal Robot Model

3.3.3 Maps

Maps for chesh_botare identical to those used for Udacity_bot.

3.3.4 Meshes

Meshes for chesh_botare identical to those used for Udacity_bot.

3.3.5 Launch

Launch files for chesh_botare identical to those used for Udacity_bot.

3.3.6 Packages Used

Packages for chesh_botare identical to those used for Udacity_bot.

3.3.7 Parameters

Parameters for chesh_botare identical to those used for Udacity_bot.

4 RESULTS

4.1 Localization Results

4.1.1 Benchmark Model - udacity_bot

The benchmark model was able to successfully reach the goal position in about 24 minutes without any major devi-

ations from the global path. Notably, the particle filter converges within the first minute and remains fairly consistent throughout the robot's path. The images below display the progress that udacity_bot made and the path it took to get to the navigation goal.

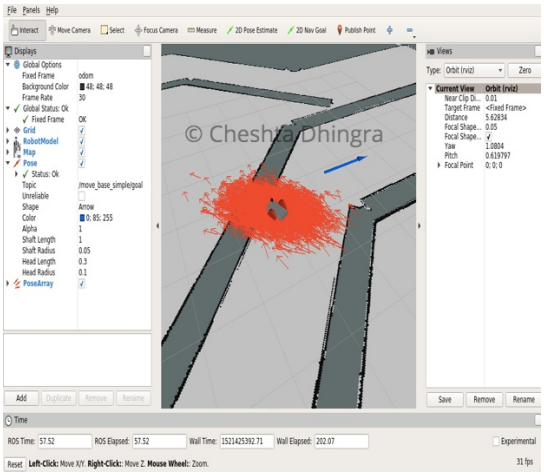

Fig. 13. The benchmark model before parameter tuning

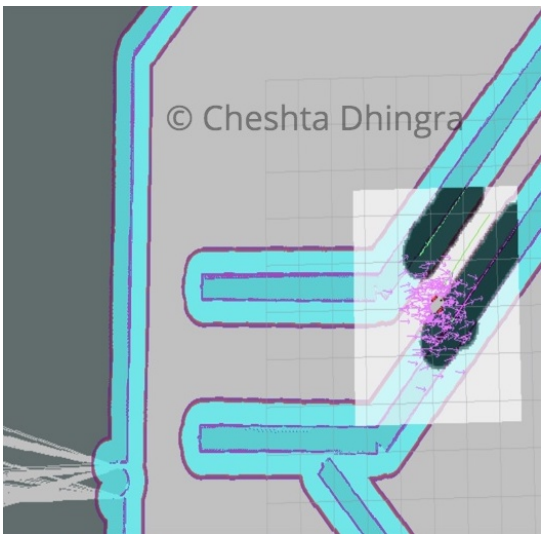

Fig. 14. The benchmark model prior to particle convergence

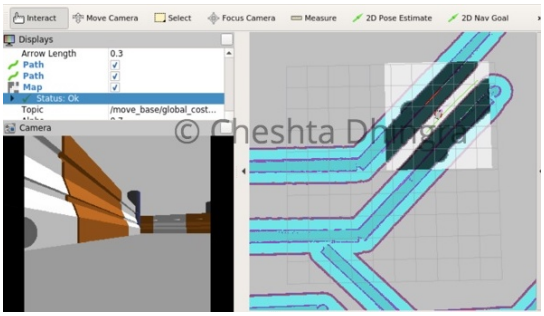

Fig. 15. The benchmark model after particles converge, on it's way to the goal

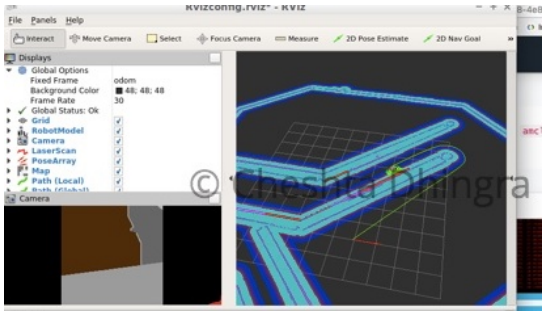

Fig. 16. The benchmark model following the global path

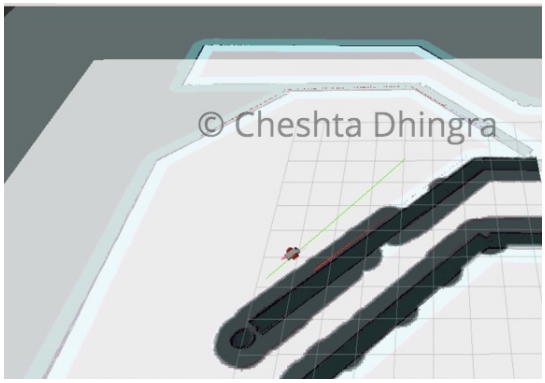

Fig. 17. The benchmark model almost at the goal

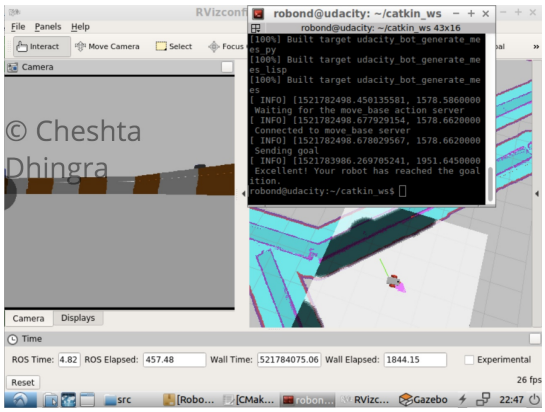

Fig. 18. The benchmark model at the goal; message displays success

Fig. 19. The benchmark model at the goal in a second run; message displays success

4.1.2 *Personal Model - chesh_bot*

Chesh-bot reached the navigation goal in about minutes. During the course of navigation. Similar to udacity_bot, it was fairly accurate in following the global path while it was in the confined zone, but once it got to a more open area it deviated from the global path. Particle filters here also converged within the first minute. The images below display the progress that chesh_bot made and the path it took to get to the navigation goal.

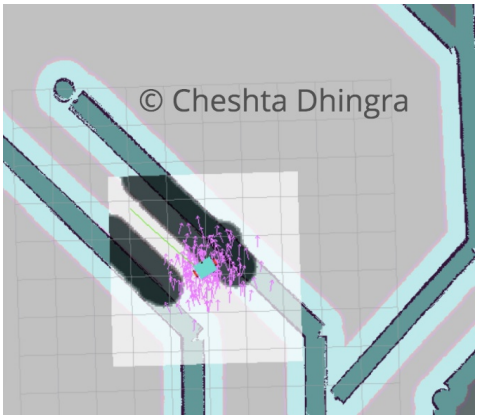

Fig. 20. chesh_bot prior to particle convergence

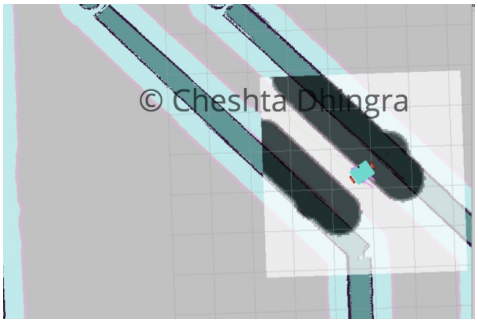

Fig. 21. chesh_bot after particle convergence

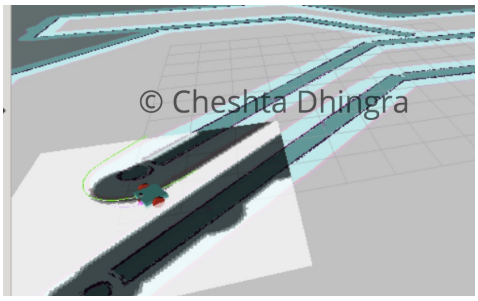

Fig. 22. chesh_bot navigates smoothly when confined

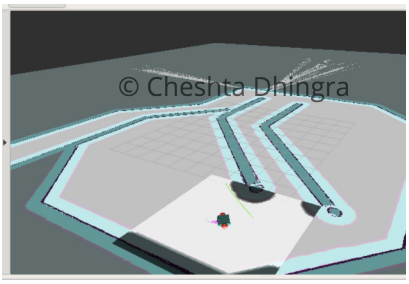

Fig. 23. chesh_bot begins to deviate in open space

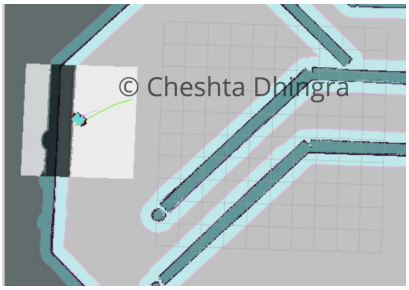

Fig. 24. chesh_bot continues to deviate in open space

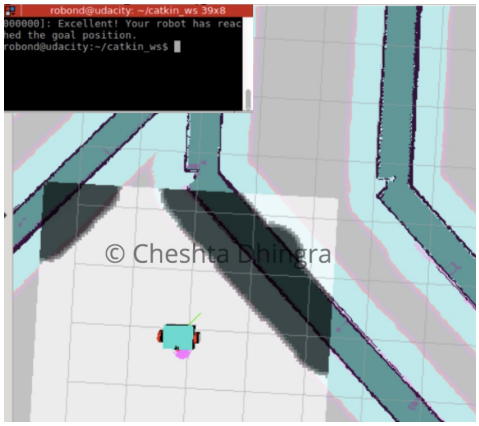

Fig. 25. chesh_bot finally reaches its goal; message displays success

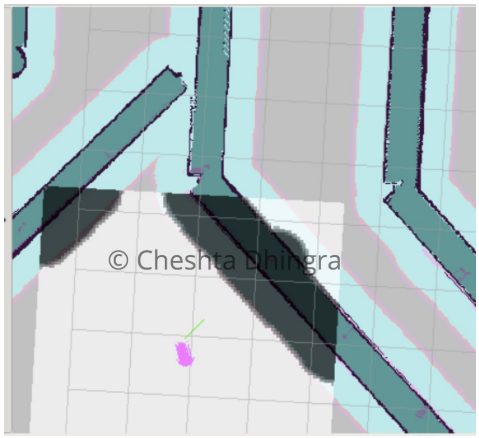

Fig. 26. chesh_bot Pose Array displayed

4.2 Technical Comparison

The key differences between the robot models are the following:

1) chesh_bot has a wider base (0.5 m instead of 0.2m)
2) chesh_bot has the Hokuyo laser rangefinder placed slightly ahead compared to udacity_bot (0.165 m vs 0.15m)

5 Discussion

While both robots reached the navigation goal, the benchmark model clearly took a more direct path and achieved the goal in the time. Given that chesh_bot is heavier than udacity_bot, the slower motion and greater deviation from the path was expected.
The placement of the laser rangefinder likely had an effect on the performance of chesh_bot, perhaps because it did not provide enough accurate information for localization. However, this is yet to be tested rigorously. One can tune the position of the Hokuyo holding all else constant in order to test this.

5.1 Topics

- The benchmark model performed better. It was able to reach the goal quicker and with fewer deviations from the path.
- This is likely due to the more streamlined structure of the benchmark model, and the fact that it is lighter than chesh_bot.
- In the kidnapped robot problem, the robot is randomly placed at a new location and must localize and navigate from the new starting point.
- Localization could be performed in a variety of industries. Recently a medical robot that helps doctors diagnose lung cancer received a nod from the FDA. The platform consists of two robotic arms attached to a long, thin tube and a videogame-style controller that the surgeon uses to manipulate the device. The tube is inserted down the patients throat and into the lungs, all the while capturing video that is sent to screens in the operating room. The surgeon uses the controller to guide the tube to the target site in the lung where it performs the procedure.
  Advanced and extremely accurate autonomous localization tools could revolutionize a device like this by removing the need for a surgeon completely!

6 Conclusion / Future work

The localization results are reasonable for both models. The benchmark model as well as the personal model show convergence of the particle filters within the first half minute of localization, and remain clustered throughout the path, indicating that AMCL is working effectively. The benchmark model reaches the goal in about 26 minutes while the personal model takes about 35 minutes. The benchmark model follows a fairly smooth path throughout its journey. The personal model follows a smooth path when bounded by walls, but deviates considerably in open space.
The project achieved its goals. However, there is great scope for improvement. One could tune the parameters further for the personal model to ensure it reaches the goal in a more straightforward manner. It would also be very interesting to deploy the models on hardware and apply it to a range of commercial products. For example, medical nanorobots could exploit localization and navigation tools to efficiently navigate the bloodstream and provide life-saving medication. The possibilities for future work in this field are virtually endless.

6.1 Hardware Deployment

An exciting area for future work could be in hardware deployment of these models. The Jetson TX2 board with ROS installed would be a great option to explore.