Robot Localization Using ROS AMCL Package

Muthanna A. Attyah

Abstract—Robot localization is the process of determining where a mobile robot is located with respect to its environment. Localization is one of the most fundamental competencies required by an autonomous robot as the knowledge of the robot's own location is an essential precursor to making decisions about future actions. [1] In this paper **Extended Kalman Filter (EKF)** localization algorithm will be compared to **Adaptive Monte Carlo localization (AMCL)** algorithm then AMCL performance will be tested in two different simulated robot configurations.

Index Terms—Robot, IEEEtran, Udacity, Localization, , AMCL, EKF, ROS

1 Introduction

NAVIGATION is one of the most challenging competencies required for a mobile robot; without it a mobile robot will not be able to achieve its tasks. Success in navigation requires four building blocks:

- Perception to detect environment.
- Localization to determine position.
- Cognition to decide on achieving goals.
- Motion control to achieve desired trajectory.

Of these four components, localization has received the greatest research attention and, as a result, significant advances have been made on this front. [2] Localization is basically the answer to the question of "Where am I ?" and there are many algorithms that can be used to answer this question; examples are Histogram Filter, Information Filter, Triangulation, Markov Filter, Kalman Filter (KF), Extended Kalman Filter (EKF), and Particles Filter or Monte Carlo Localization (MCL), and Adaptive Monte Carlo Localization (AMCL). [3] [4] Different algorithms has different approaches; linear or not, probabilistic or not, 1D, 2D, 3D, taking single sensor or more, high computing requirement or less, etc. In this paper focus will be on comparing EKF to AMCL then test the performance of AMCL on two different simulated robots.

2 Background

The focus of this project will be on probabilistic 2D filters that can handle non-linear sensor data with jerky errors. to better understand these filters, a high-level description of each type is given below:

2.1 Markov localization

Probabilistic localization algorithms are variants of the Bayes filter. The straightforward application of Bayes filters to the localization problem is called Markov localization. Markov localization, uses an explicitly specified probability distribution across all possible robot positions. [2]

2.2 Kalman Filters

A second method, Kalman filter (KF) localization, is a linear quadratic estimator that uses a Gaussian probability density representation of robot position and scan matching for localization. Unlike Markov localization, Kalman filter localization does not independently consider each possible pose in the robots configuration space. Interestingly, the Kalman filter localization process results from the Markov localization axioms if the robots position uncertainty is assumed to have a Gaussian form. [2] The disadvantage of KF is the assumption that motion and measurements are linear, and the assumption that state space can be represented by a unimodal Gaussian distribution. Both assumptions are not true in most of robots hence a non-linear algorithm will be a better fit for mobile robots. [3]

2.3 Extended Kalman Filters

The extended Kalman filter localization algorithm, or EKF localization, is a special case of Markov localization which will resolve the issues of KF by handling non-linear motion and measurements. EKF localization represent beliefs by their first and second moment, that is, the mean and the covariance. [3] [4]

2.4 Particle Filters

A Particle filter is another great choice for localization if a robot system doesn't fit nicely into a linear model, or a sensor uncertainty doesn't look very Gaussian. A Particle Filter make it possible to handle almost any kind of model, by discretizing the problem into individual "particles" – each one is basically one possible state of your model, and a collection of a sufficiently large number of particles lets you handle any kind of probability distribution, and any kind of evidence (sensor data).the estimation error in a particle filter does converge to zero as the number of particles (and hence the computational effort) approaches infinity. Particle filter is also called Monte Carlo Localization (MCL). . [5]

2.5 Comparison / Contrast

While Kalman filter can be used for linear or linearized processes and measurement system, the particle filter can be used for nonlinear systems. The uncertainty of Kalman filter is restricted to Gaussian distribution, while the particle filter can deal with non-Gaussian noise distribution. In cases where abrupt sensor noise is rarely observed, both filters work fairly well. However, when sensor noise exhibits jerky error, Kalman filter results in location estimation with hopping while particle filter still produces robust localization. Particle representations often yield surprisingly simple implementations for complex nonlinear systems one of the reasons for their recent popularity. This paper also compares performance of these filters under various measurement uncertainty and process uncertainty. [6] [4] Following table summarize the differences between Extended Kalman Filter (EKF) and Particle Filter (MCL).

Item	MCL	EKF
Measurement	Raw Measurement	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Efficiency - Memory	✓	✓✓
Efficiency - Time	✓	✓✓
Easy of Implementation	✓✓	✓
Resolution	✓	✓✓
Robustness	✓✓	N.A.
Memory/Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodal Discrete	Unimodal Continuous

TABLE 1: Comparing MCL with EKF

When good computing capability is available for the mobile robot, MCL filter will be a good choice for localization. Two types of simulated robots will be used to test the performance of an Adaptive MCL filter (AMCL) in ROS environment.

3 Simulations

Next sections will describe the design details of the two simulated robots used to test performance of AMCL package in ROS/Gazebo/Rviz environment.

3.1 Achievements

Both benchmark and personal robots were able to reach set goals using AMCL algorithm after fine tuning the parameters to match the design of each robot. The major difference between the two models was the number of used wheels and the driving mode. Benchmark robot is using differential drive while personal robot was using omni drive. Skid driving was also tested on personal robot. Used LIDAR and Camera sensors remained the same in both robots. Chassis is slightly changed in personal robot to give it better shape and fit the 4 wheels. Test runs were recorded and published on YouTube:

- Differntial Drive Video.
- Omni Drive Video.

3.2 World Map

Jackal Race world map will be used in simulation; this map was originally created by Clearpath Robotics.

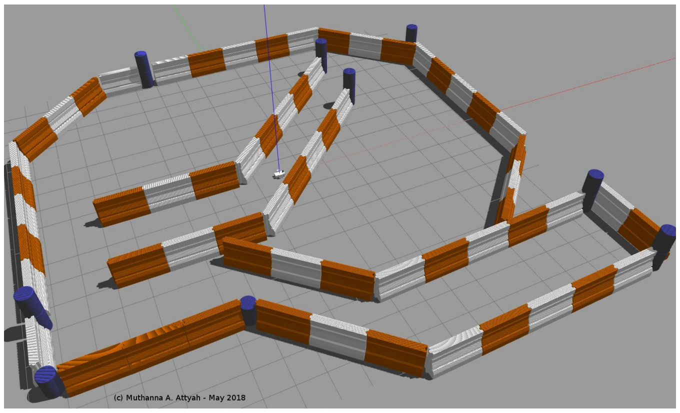


Fig. 1: Jackal Race Map

3.3 Benchmark Model

1st simulated robot (**udacity_bot**) is the benchmark model that was provided as part of udacity project.

3.3.1 Model design

Benchmark robot **udacity_bot** design includes the following components:

Item	Type	Size
robot_footprint	Link	N.A.
robot_footprint_joint	joint - fixed	N.A.
chassis	Link	box: 0.4 x 0.2 x 0.1
back_caster_visual	visual	sphere: r=0.05
front_caster_visual	visual	sphere: r=0.05
left_wheel	Link	cylinder: r=0.1 l=0.05
left_wheel_hinge	joint - continuous	N.A.
right_wheel	Link	cylinder: r=0.1 l=0.05
right_wheel_hinge	joint - continuous	N.A.
camera	Link	box : 0.05 x 0.05 x 0.05
camera_joint	joint - fixed	N.A.
hokuyo	Link	mesh : 0.1 x 0.1 x 0.1
hokuyo_joint	joint - fixed	N.A.

TABLE 2: udacity_bot design

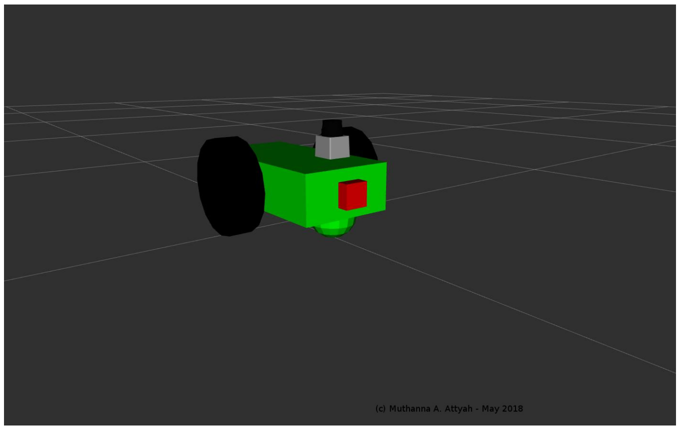


Fig. 2: udacity_bot in RViz

3.3.2 Packages Used

Following Packages were used in ROS simulated robots and environment:

map_server: provides the map_server ROS Node, which offers map data as a ROS Service. [7]

move_base: provides an implementation of an action that, given a goal in the world, will attempt to reach it with a mobile base. The move_base node links together a global and local planner to accomplish its global navigation task. The move_base node also maintains two costmaps, one for the global planner, and one for a local planner that are used to accomplish navigation tasks. [7]

joint_state_publisher: contains a tool for setting and publishing joint state values for a given URDF. [7]

robot_state_publisher: allows to publish the state of a robot to tf. Once the state gets published, it is available to all components in the system that also use tf. The package takes the joint angles of the robot as input and publishes the 3D poses of the robot links, using a kinematic tree model of the robot. [7]

amcl: a probabilistic localization system for a robot moving in 2D. It implements the adaptive (or KLD-sampling) Monte Carlo localization approach (as described by Dieter Fox), which uses a particle filter to track the pose of a robot against a known map. [7]

gazebo_ros: provides ROS plug-ins that offer message and service publishers for interfacing with Gazebo through ROS. [7]

rviz: 3D visualization tool for ROS. [7]

Other than what is described in next subsection, all topics published, topics subscribed, services, and parameters of above packages are as described on ROS wiki [7], there are no customized service, topics, messages, or parameters used in this package.

3.3.3 Gazebo plug-ins

libgazebo_ros_diff_drive.so: provides a basic controller for differential drive robots in Gazebo.

libgazebo_ros_camera.so: provides ROS interface for simulating cameras.

libgazebo_ros_laser.so: simulates laser range sensor by broadcasting LaserScan message as described in sensor_msgs.

3.3.4 Parameters

Parameters used to fine tune ROS navigation stack and localization performance are divided into 5 yaml formatted configuration files:

1) **amcl.yaml** The amcl package has a lot of parameters to select from. Different sets of parameters contribute to different aspects of the algorithm. Broadly speaking, they can be categorized into three categories - overall filter, laser, and odometry.

   Overall filter parameters:
   min_particles: Minimum allowed number of particles.
   max_particles: Maximum allowed number of particles. Particle representations can approximate a wide array of distributions,but the number of particles needed to attain a desired accuracy can be large. Min/Max number needs to be adjusted to get the required accuracy without exceeding the available computing capability in the robot system. High number of particles will cause the system to miss publishing cycles.
   transform_tolerance: Time with which to post-date the transform that is published, to indicate that this transform is valid into the future. this parameter is also dependent on your system specifications. This helps decide the longevity of the transform(s) being published for localization purposes. A good value should only be to account for any lags in the system.
   initial_pose_x (_y) (_a): Initial pose mean (x,y,yaw), used to initialize filter with Gaussian distribution.

   Laser model parameters:
   laser_model_type: likelihood field model is widely used. While it lacks a strict probabilistic interpretation, it tends to be more reliable, especially in environments with small obstacles. Also, the likelihood field model is more computational efficient than the beam model, which requires ray-tracing at run time (the likelihood field model pre-computes the field at startup, so that the run time work is simply a table lookup). Whichever mixture weights are in use should sum to 1, the likelihood_field model uses only 2: z_hit and z_rand [4]
   laser_z_hit: Mixture weight for the z_hit part of the model.
   laser_z_rand: Mixture weight for the z_rand part of the model.
   laser_max_beams:How many evenly-spaced beams in each scan to be used when updating the filter.

   Odometry model parameters:
   odom_frame_id:Which frame to use for odometry.
   odom_model_type:Which model to use, either ”diff”, ”omni”, ”diff-corrected” or ”omni-corrected”.If odom_model_type is ”diff” then sample_motion_model_odometry algorithm is used; this model uses the noise parameters odom_alpha_1 through odom_alpha4
   odom_alpha1..4: Parameters odom_alpha_1 through odom_alpha4 specifies the expected noise in odometry’s rotation estimate. used for ”diff” and ”omni” odom model.

odom_alpha5:Translation-related noise parameter (only used if model is ”omni”) captures the tendency of the robot to translate (without rotating) perpendicular to the observed direction of travel.

2) **base_local_planner_params.yaml**:**move_base** package creates and calculates a path or a trajectory to the goal position, and navigates the robot along that path. The set of parameters in this configuration file customize this particular behavior.

   holonomic_robot: Determines whether velocity commands are generated for a holonomic or non-holonomic robot. For holonomic robots, strafing velocity commands may be issued to the base. For non-holonomic robots, no strafing velocity commands will be issued.
   meter_scoring: Whether the gdist_scale and pdist_scale parameters should assume that goal_distance and path_distance are expressed in units of meters or cells.
   yaw_goal_tolerance: The tolerance in radians for the controller in yaw/rotation when achieving its goal.
   xy_goal_tolerance: The tolerance in meters for the controller in the x & y distance when achieving a goal.
   occdist_scale: The weighting for how much the controller should attempt to avoid obstacles.
   sim_time: The amount of time to forward-simulate trajectories in seconds.
   pdist_scale:The weighting for how much the controller should stay close to the path it was given, maximal possible value is 5.0.

3) **costmap_common_params.yaml** this file is for the list of parameters common to both types of costmaps; it defines which sensor is the source for observations. thats the laser sensor in this project case. Aside from that, there are a few parameters that are also required as listed below.

   map_type: What map type to use. ”voxel” or ”costmap” are the supported types, with the difference between them being a 3D-view of the world vs. a 2D-view of the world.
   obstacle_range: 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: 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.
   transform_tolerance: Specifies the delay in transform (tf) data that is tolerable in seconds. This parameter serves as a safeguard to losing a link in the tf tree while still allowing an amount of latency the user is comfortable with to exist in the system. For example, a transform being 0.2 seconds out-of-date may be tolerable, but a transform being 8 seconds out of date is not. If the tf transform between the coordinate frames specified by the global_frame and robot_base_frame parameters is transform_tolerance seconds older than ros::Time::now(), then the navigation stack will stop the robot. In .
   footprint: Specification for the footprint of the robot. This is changed for the personal robot because of the bigger footprint after adding 4 wheels instead of 2 wheels in benchmark robot.
   inflation_radius: .
   observation_sources: A list of observation source names separated by spaces.
   laser_scan_sensor: Name of the laser scanner observation source.

4) **global_costmap_params.yaml** This file consists of parameters that specify the behavior associated with local (or global) costmap. The local costmap relies on odom as a global frame since it updates as the robot moves forward. Since the costmap updates itself at specific intervals, and aims to cover a specific region around the robot it requires its own updating and publishing frequencies, as well as dimensions for the costmap. a similar set of parameters is used for the local costmap.

   global_frame: The global frame for the costmap to operate in.
   robot_base_frame: The name of the frame for the base link of the robot.
   update_frequency: The frequency in Hz for the map to be updated. In both robots frequency of higher than 10 was causing errors of missed cycles.
   publish_frequency: The frequency in Hz for the map to be publish display information. In both robots frequency of higher than 10 was causing errors of missed cycles.
   width: The width of the map in meters.
   height: The height of the map in meters.
   resolution: The resolution of the map in meters/cell.
   static_map: The costmap has the option of being initialized from a user-generated static map. If this option is selected, the costmap makes a service call to the map_server to obtain this map.
   rolling_window: 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.

5) **local_costmap_params.yaml**
   Same list of global costmap parameters as described above are used for the local costmap. [8] [9]

3.4 Personal Model

2nd simulated robot (**muth_bot**) is designed with different configuration than the benchmark robot to allow testing of related AMCL parameters.

3.4.1 Model design

Personal robot **muth_bot** includes the following components:

Item	Type	Size
robot_footprint	Link	N.A.
robot_footprint_joint	joint - fixed	N.A.
chassis	Link	box: 0.4 x 0.2 x 0.15
front_left_wheel	Link	cylinder: r=0.1 l=0.05
front_left_wheel_hinge	joint - continuous	N.A.
front_right_wheel	Link	cylinder: r=0.1 l=0.05
front_right_wheel_hinge	joint - continuous	N.A.
back_left_wheel	Link	cylinder: r=0.1 l=0.05
back_left_wheel_hinge	joint - continuous	N.A.
back_right_wheel	Link	cylinder: r=0.1 l=0.05
back_right_wheel_hinge	joint - continuous	N.A.
camera	Link	box : 0.05 x 0.05 x 0.05
camera_joint	joint - fixed	N.A.
hokuyo	Link	mesh : 0.1 x 0.1 x 0.1
hokuyo_joint	joint - fixed	N.A.

TABLE 3: muth_bot design

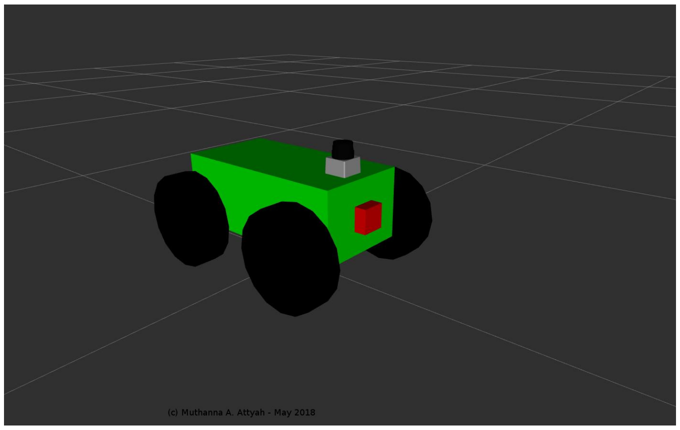


Fig. 3: muth_bot in RViz

3.4.2 Packages Used

Packages used in the 2nd robot is the same as listed in the 1st robot.

3.4.3 Gazebo plug-ins

skid_steer_drive_controller: plug-in was used to test skid steering.

object_controller: plug-in was used to test planner move (omni directional move).

libgazebo_ros_camera.so: provides ROS interface for simulating cameras.

libgazebo_ros_laser.so: simulates laser range sensor by broadcasting LaserScan message as described in sensor_msgs.

3.4.4 Parameters

In addition parameters details explained in the benchmark robot (**udacity_bot**) section; following are adjustments that was done for the modified design of personal robot (**muth_bot**): [8] [9]

4 Results

Next two sub sections will present and compare the performance results of the two simulated robots.

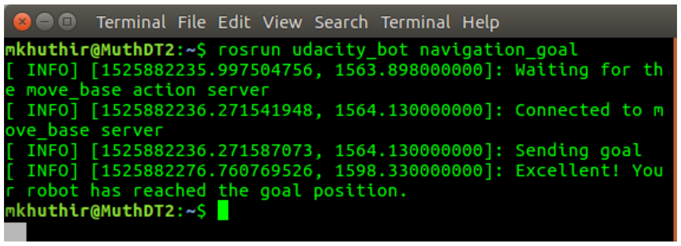
4.1 Localization Results

4.1.1 Benchmark Robot udacity_bot

Navigation goal is sent to the simulated benchmark robot by running the navigation_goal code using rosrun. As shown in below terminal output when converting the info Unix time stamps to ISO format:

Goal Sent : 2018-05-09T20:10:36.271587
Goal Reached: 2018-05-09T20:11:16.760770
Time Taken : **40.489** Seconds

Time taken for the benchmark robot to reach its goal was approximately 40 seconds. Robot was able to navigate all of the way to the goal properly avoiding walls and obstacles. Slowest speed was when the robot doing U turn at the far end of the path. Robot approached the actual goal position and it did not need to correct itself when at the goal area.



```
mkhuthir@MuthDT2:~$ rosrun udacity_bot navigation_goal
[ INFO] [1525882235.997504756, 1563.898000000]: Waiting for th
e move_base action server
[ INFO] [1525882236.271541948, 1564.130000000]: Connected to m
ove_base server
[ INFO] [1525882236.271587073, 1564.130000000]: Sending goal
[ INFO] [1525882276.760769526, 1598.330000000]: Excellent! You
r robot has reached the goal position.
mkhuthir@MuthDT2:~$
```

Fig. 4: udacity_bot time to goal

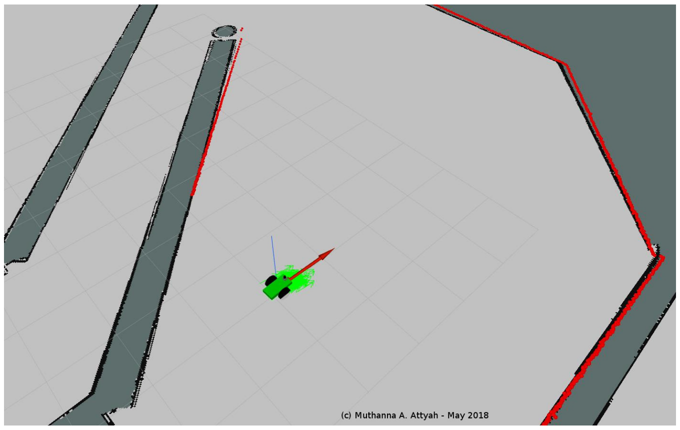


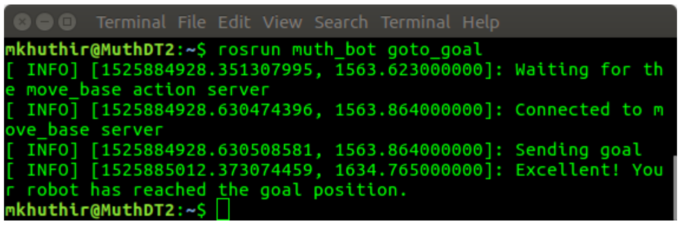
Fig. 5: udacity_bot reached target

4.1.2 Student Robot muth_bot

Navigation goal is sent to the simulated benchmark robot by running the navigation_goal code using rosrun. As shown in below terminal output when converting the info Unix time stamps to ISO format:

Goal Sent : 2018-05-09T20:55:28.630509
Goal Reached: 2018-05-09T20:56:52.373075
Time Taken : **83.743** Seconds

Time taken for the student robot to reach its goal was approximately 84 seconds which is double the time required for the benchmark robot ! Same behavior was noticed when the robot was doing the U turn, it was slowing down. Robot did not approach the exact goal position correctly, it spent big portion of the time to correct itself and stand on the exact goal position/orientation.



```
mkhuthir@MuthDT2:~$ rosrun muth_bot goto_goal
[ INFO] [1525884928.351307995, 1563.623000000]: Waiting for th
e move_base action server
[ INFO] [1525884928.630474396, 1563.864000000]: Connected to m
ove_base server
[ INFO] [1525884928.630508581, 1563.864000000]: Sending goal
[ INFO] [1525885012.373074459, 1634.765000000]: Excellent! You
r robot has reached the goal position.
mkhuthir@MuthDT2:~$
```

Fig. 6: muth_bot time to goal

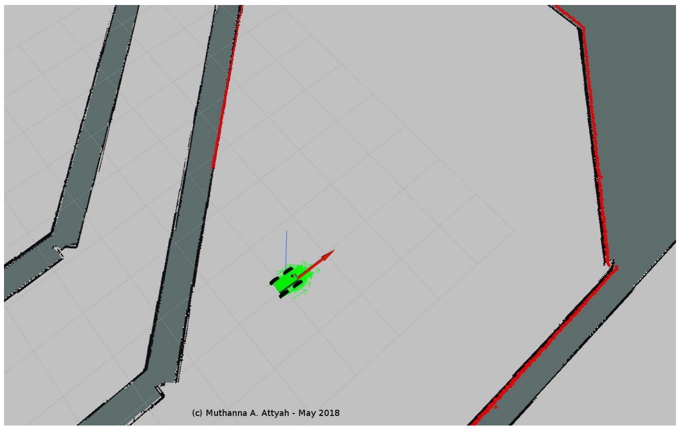


Fig. 7: Muth_bot reached target

4.2 Technical Comparison

The student robot performed worse than the benchmark robot, it took 84 seconds to reach goal while benchmark robot took only 40 seconds. both robots were able to smoothly navigate to the goal avoiding all obstacles in the world map.

5 Discussion

5.1 Topics

- Which robot performed better? Considering the time required to reach goal and the trajectory taken by both robots, benchmark robot performance was better.
- Why it performed better? Last stage of the trajectory where the robot needs to adjust both its position and orientation was taking much longer in the case of personal robot because omni drive was not as flexible as the deferential drive.
- How would you approach the 'Kidnapped Robot' problem? In 'kidnapped robot' problem a well-localized robot is tele-ported to some other place without being told. This problem differs from the global localization problem in that the robot might firmly believe itself to be somewhere else at the time of the kidnapping. The kidnapped robot problem is often used to test a robots ability to recover from catastrophic localization failures. The regular MCL algorithm with small number of particles is unfit for the kidnapped robot problem, since there might be no surviving samples nearby the robots new pose after it has been kidnapped. The solution is either by increasing the number of samples which will require higher computing power from the robot system or using a modified more effective algorithm such as mixed-MCL [10]
- What types of scenario could localization be performed? performing inspection tasks in industrial area, house cleaning robot, garden mowing robot, and retail warehouse robots.
- Where would you use MCL/AMCL in an industry domain? Mobile base robots in industrial manufacturing where manipulators are able to move performing all sorts of tasks with high accuracy, such as welding, painting, cutting.

6 Conclusion / Future work

In summary both robots were able to reach the required goals using two different designs, parameters can be further improved in both cases to get better results or make the robot ready for certain tasks/types of obstacles. Next section will explain few suggested improvements.

6.1 Modifications for Improvement

There will be always a room for improvement; robot performance can be improved by applying some of the following steps.

- **Base Dimension**: Robot base design can be further improved to simulate a more realistic design; mesh files can be used to show the actual shape of the hardware that will be used, base can be decided based on number and locations of the required sensors plus the targeted tasks that will be done by the robot.
- **Sensor Location**: At the initial design of the personal robot, LIDAR was detecting the front wheels as obstacles, sensor location was changed to resolve this issue, similarly, based on expected tasks that will

be done by robot and the hight/range of obstacles, sensors locations can be further improved.

- **Sensor Layout**: Having two LiDARs with 180 degrees of visibility each can cover 360 degrees around the robot which will improve over all localization accuracy. Another way is to have one 360 degrees LiDAR in the middle of the robot base to cover the full range.
- **Sensor Amount**: Additional types/number of sensors can also improve the robot performance and make it more robust to handle kidnapped robot situations.
- **Computing Power**: It was noticed that increasing the accuracy of maps or increasing the frequency of publishing samples was having direct impact on robot system performance and there was many cases of missed computing cycles when the publishing frequency was higher than 15Hz. Additional computing power (deploying a higher CPU or using a GPU) will for sure improve robot overall performance. A good example is using NVIDIA Jetson board instead of Raspberry PI as a local robot board and a powerful i7 Gen8 PC as a back-end PC for the robot.

6.2 Hardware Deployment

Actual robot hardware can be used to replace simulated robots and test the same AMCL algorithm on it. One great example is the Robotis TurtleBot3 Burger (shown in below figure). Burger has two differential wheels driven by two Dynamixel XL430-W250-T motors, one ball caster, one 360 degree LiDAR with detection distance from 120mm to 3,500mm. a Raspberry Pi board to run required ROS nodes that are local to the robot plus device drivers for the LiDAR, and one 32-bits ARM Cortex-M7 based controller to control wheels motors. commands to drive motors are sent from Raspberry PI ROS nodes to the ARM based Dynamixel controller over USB port. [11]

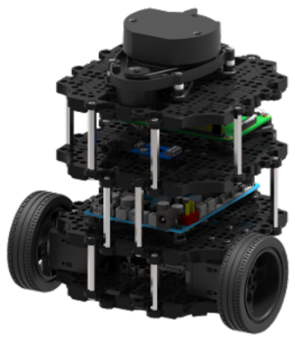


Fig. 8: TurtleBot 3

1) What would need to be done?

   Testing can be started on a simulated TurtleBot3 which is already available on GitHub repository https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git.
   Launch files will have to be changed to include all required packages described above along with adjusted parameters that will fit the robot size, driving mode, and used sensors. Once simulation is a success, node/topics names can be changed to point to the physical robot instead of the simulated one.

2) Computation time/resource considerations?

   The Raspberry PI included with the robot will be good enough to run ROS nodes that will collect data from LiDAR sensor, send it using WiFi connection to a back-end PC that will run AMCL and remaining Navigation Stack nodes. Raspberry PI will also run the ROS nodes that will drive the wheels. Computation heavy lifting will be done by the back-end PC.

References

[1] S. H. G. Dissanayake, “Robot localization: An introduction,” *Wiley Encyclopedia of Electrical and Electronics Engineering*, 15 August 2016.
[2] I. R. N. Roland Siegwart, *Introduction to Autonomous Mobile Robots*. MIT Press, 2004.
[3] K. M.-M. Ling Chen, Huosheng Hu, “Ekf based mobile robot localization,” 2012.
[4] D. F. Sebastian Thrun, Wolfram Burgard, *Probabilistic Robotics*. MIT Press, 2005.
[5] J. Reich, “What is the difference between a particle filter and a kalman filter?.”
[6] T. G. K. Nak Yong Ko, “Comparison of kalman filter and particle filter used for localization of an underwater vehicle,” 19 February 2013.
[7] “Ros wiki web site.”
[8] K. Zheng, “Ros navigation tuning guide,” September 2, 2016.
[9] Gilbert, “Ros wiki: Amcl package summary.”
[10] W. B.-F. D. Sebastian Thrun, Dieter Fox, “Robust monte carlo localization for mobile robots,” 2001.
[11] Robotis, “Robotis turtlebot 3,” 2017.