Localization

Dhawal Wazalwar

Abstract—In this project, focus is on two important aspects of Robotics: firstly on how to create a customized mobile robot model in ROS and secondly to understand and explore ROS amcl package to achieve desired localization performance. Kalman and Particle filters are two of the most popular techniques used to solve localization problem and they are described in this report highlighting their pros and cons. Parameter tuning for both amcl and move_base ROS packages is explained in depth. Results for both Udacity provided as well as Custom-Created Robot model are provided. Finally, trade off in terms of accuracy and processing time is discussed alongwith some suggested future improvements.

Index Terms—Localization, Kalman Filter, Particle Filter

1 Introduction

In mobile robotics, determining robot's pose in a mapped environment can be a very challenging task. Apart from environmental factors, sensors used are also noisy, adding to the overall uncertainty robot has to deal with. Localization algorithms like Extended Kalman Filter (EKF) and Monte Carlo Localization (MCL) perform this task by implementing a probabilistic approach to filter out noisy measurements and then track robot's exact position and orientation.

In this project *Where am I*, Adaptive Monte Carlo Localization (AMCL) ROS package is integrated with Robot to localize it accurately in the provided map. ROS move_base package is also used to define a goal position and challenge is to fine tune both amcl and navigation stack parameters to help Robot navigate in given environment and reach expected goal position.

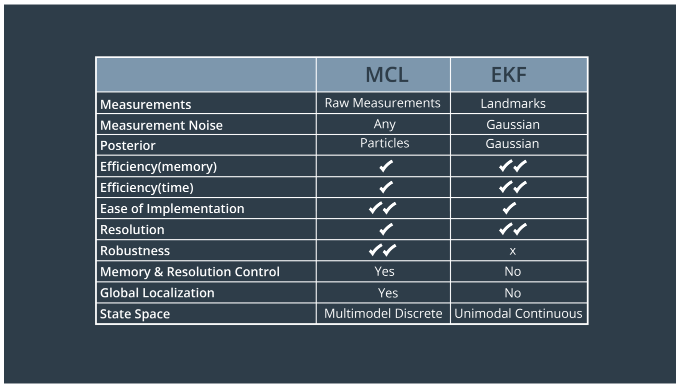
2 Background / Formulation

Traditional approaches assume Robot's environment is deterministic, fully observable and static in nature [1]. In reality, environment is actually stochastic, partially observable and dynamic creating a uncertainty around Robot. Also for designing a robust autonomous robot, sensor fusion is employed to get most accurate location estimate by combining data from multiple sensors. For example in self driving cars, odometry sensor data is combined with external sensors like LiDAR or RADAR to sense ambience around it, adding to the overall complexity in localization task.

Apart from nature of environment, information provided to Robot also determines specific localization type. Mainly there are three different localization types: local, global and Kidnapped Robot case. In local case, initial pose is known and task is to keep a track of its position as Robot navigates. In global case, localization becomes more difficult as initial pose is not provided and Robot must determine its pose relative to ground truth map. Kidnapped Robot case is an extension of Global except Robot's location can be changed randomly at any time. Although practically this is not likely to happen, this case is actually considered worst case scenario for localization task. For this project *Where am I*, map provided by ClearPath Robotics is static in nature and thereby uncertainty is mostly around Robot's surrounding environment.

Kalman filter was one of the first widely used algorithm to estimate state of the system when measurements are noisy. It models the level of uncertainty around measured value using Gaussian distribution. Kalman filter is an iterative two step process starting with an initial guess and can very quickly develop an accurate estimate of true value of the variable being measured. The problem with Kalman Filter is that it is applicable only to linear motion and measurement models. In non-linear cases, Extended Kalman Filter (EKF) is required, wherein we use local linear approximation using first two terms of Taylor series to update the covariance of the estimate. In multi-dimensional case, Jacobian matrix of partial derivatives is used, as there are multiple state variables and dimensions that need to be considered.

Monte Carlo Localization (MCL) [2]is the most popular localization algorithm used in robotics. It uses Particle Filters, virtual elements representing robot and initially they span across the entire map. In addition to position and orientation information, particles have weight value, which is actually difference between the robot's actual pose and particle's expected pose. In terms of algorithm, it is two step iterative process: first step is motion and sensor update, while the second one is resampling. During the resampling process, particle with higher weights are more likely to be picked and one with lower weights eventually die. Repeating this process for few iterations quickly filters out non-Robot particles and only keeps particle representing Robot in the map. Apart from MCL's algorithm simplicity, one major difference compared to EKF is it can handle multi-modal posterior distribution, while EKF always approximates posterior distribution as Gaussian. Figure 1 below shows more detailed comparison of EKF vs MCL. From table, it can be seen MCL offer us an option to control our computational memory and solution by allowing to change number of randomly spread out particles. For this project in ROS, Adaptive Monte Carlo Localization (AMCL) package has been used to configure number of particles dynamically while Robot navigates in provided map.



	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. 1. Comparison of MCL vs EKF

3 Custom Robot Model

A basic mobile robot model with urdf and gazebo files including plugins integration for simulation was provided. Apart from the udacity provided one, in this project another slim robot model with different sensor locations was also created. This custom model was created keeping in mind some of the common retail robot we see in warehouses or stores these days. Fig 2 below shows both robot models.

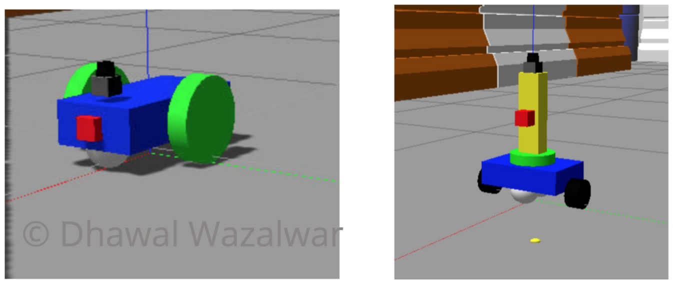


Fig. 2. Robot Models used in Localization Project; Left one is Udacity Model, while right one is Custom Slim Model

For robot URDF file, links need to be defined and for each link there are several elements like collision, inertial etc that need to be specified. In terms of robot actuation, wheels need to be added along with joint types as continuous. For Custom Slim Robot, wheel velocity parameter was reduced so that it does not fall over itself. Since both robot models are two wheeled, in .gazebo file, plugin with differential drive controller has been used. Both models are visualized in Gazebo as well as RViz, thereby creating completely launch-able ROS package.

4 Results

Both Robot models reached expected goal position, Fig 3 and 4 shows RViz images for both models at goal position. Udacity provided Robot reached in 150 seconds, while Custom Slim model took about 240 seconds. This difference in time can be attributed to difference in its parameters modified in urdf, mainly wheel size and velocity values.

5 Model Configuration

For localization performance, there are two set of parameters that required fine tuning [3]. Firstly for navigation stack i.e move_base package and secondly for amcl ros package.

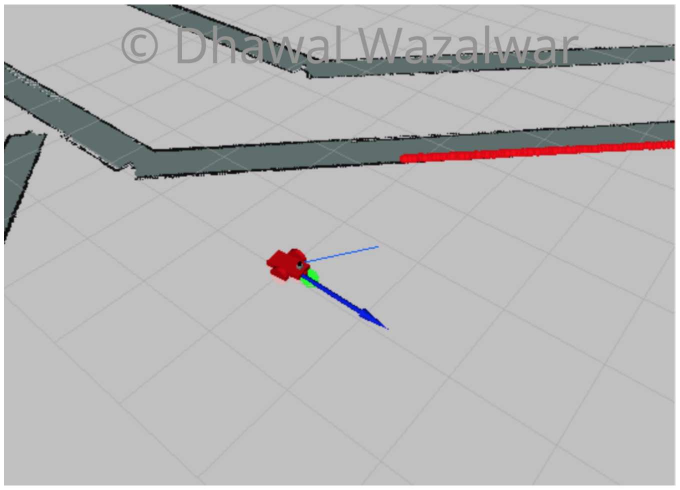


Fig. 3. Rviz image for Udacity Robot at goal position

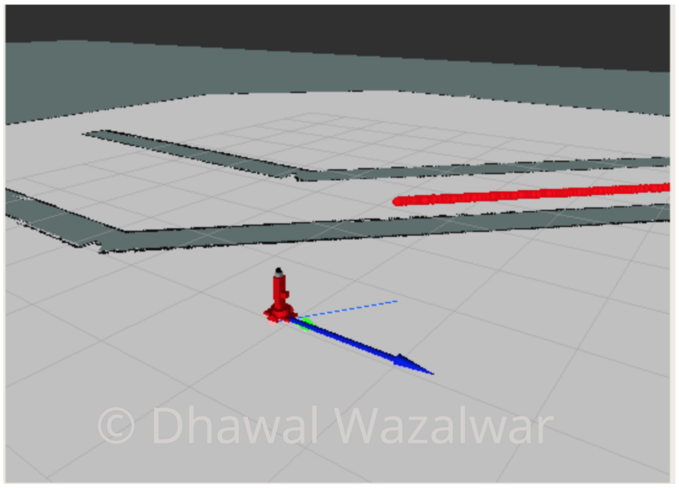


Fig. 4. Rviz image for Custom Slim Robot at goal position

Navigation stack/package computes a costmap, which is basically dividing map in grids with each each cell representing either a free space or an obstacle. In this project there are 3 types : world, global and local costmaps. World costmap is overall environment visualization from Gazebo point of view, global costmap is maintained for long term path planning for Robot from starting position. Local costmap is for immediate nearby path planning and also making sure robot is aligned to global path. In terms of parameters there are 3 separate files: costmap_common_params.yaml, global_costmap_params.yaml and local_costmap_params.yaml.

One of the first common parameters updated was transform_tolerance, this parameter defines the maximum amount of delay between the multiples coordinate frame transforms, along with any transforms corresponding to the robot and its sensors. This also helped fixed initial warning seen in transform timeout during simulation. Inflation radius defines minimum distance between the robot geometry and the obstacles in the map. Important thing to consider while selecting this value is to be aware of the narrowest pathways in the provided map. Figure 5 shows comparison of global cost map for small and large inflation radius values. As can be seen in image, obstacles gets inflated for higher inflation radius values and thereby can affect smooth navigation of robot. For udacity Robot, value of 0.15 worked well, whereas custom robot being more slim allowed to increase inflation radius and also helped it to reach goal position quicker. Obstacle range specifies the distance from robot base at which obstacle gets added to the costmap. Keeping a too high value for this parameter confused the robot at times during navigation as it seems to sense obstacles everywhere and so keeps turning constantly. An optimal value needs to be selected in such a way that we sense all obstacles correctly, for udacity robot this value was 1.0 whereas for Custom Robot this was set to be 1.5. Table 1 below shows final selected values for costmap_common_params.yaml.

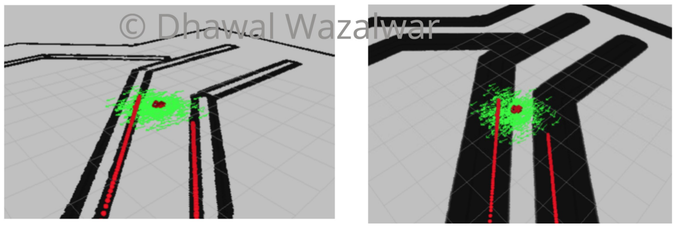


Fig. 5. Global Cost Map wrt Inflation radius: On left inflation radius is 0.1, while on right its 0.5, while right one is Custom Slim Model

Parameter	Default	Udacity	Custom
Transform Tolerance	0.2	0.15	0.15
Obstacle Range	2.5	1.0	1.5
Inflation Radius	0.55	0.15	0.2

TABLE 1
CostMap common parameters summary

Costmap resolution is set separately for global and local cost map. If we keep low width and height values, it reduces computational load but overall map accuracy is also impacted. In such lower resolution maps, obstacles seem to overlap and at narrow turns, Robot was not able to navigate smoothly and took more time. In the end, values of 40x40 for global and 20x20 for local costmap seemed to work well. Another set of important parameters are update and publish frequency, which decide overall map update cycle rate. A very high frequency value increases computational load and causes too fast updates, also giving missing map update warning during simulation. Table 2 and 3 give summary of global_costmap_params.yaml and local_costmap_params.yaml parameters.

Parameter	Default	Both Robots
Update Frequency	5.0	5.0
Publish Frequency	0.0	5.0
Width	10	40
Height	10	40
Resolution	0.05	0.05

TABLE 2
CostMap Global parameters summary

Parameter	Default	Both Robots
Update Frequency	5.0	5.0
Publish Frequency	0.0	5.0
Width	10	20
Height	10	20
Resolution	0.05	0.05

TABLE 3
CostMap Local parameters summary

In case of AMCL, there are 3 types of parameters: overall filter, laser and odometry. For this project, odometry vales are directly received from Gazebo and are same as groundtruth, so only filter and laser parameters were fine tuned. Mainly for better filter performance, deciding the range of particles size is important and in this case max size of 500 seemed to work good enough. Finally update_min_a and update_min_d parameters were set based on how frequently filter updates need to be applied. Keeping it too low means too many frequent updates which may not be necessary. Table 4 shows summary of all these 3 parameters set for AMCL. Since Custom robot has slightly lower speeds due to modified wheel and velocity parameters, update_min_a and update_min_d have to be relatively lower as compared to Udacity Robot

Parameter	Default	Udacity	Custom
Max Particles	5000	500	500
update_min_d	0.2	0.01	0.005
update_min_a	0.523	0.01	0.005

TABLE 4
AMCL parameters summary

6 Discussion

From results and localization parameters used, it can be clearly seen overall parameter tuning is clearly dependant on robot model. Custom Robot was more vertical structurally, so it was important to reduce wheel's allowed max velocity to make sure it does not trip over itself by sudden unexpected stops and turns. The localization parameters used in case of Udacity Robot did not work as expected in case of Custom Robot and custom robot also took more time to reach goal position. This tradeoff with respected to time was need to make sure we have functional accuracy.

Apart from size or feature aspects of robot model, one important to consider is the application it is intended for. While defining time vs performance requirements for localization, we need to consider safety and functional aspects separately. For example, requirements defined for home robot are not expected to be same say for outdoor applications like self-driving cars or drones. Obstacle range, inflation radius or even resolution for such outdoor applications should be first discussed in terms of safety aspects and then can be refined further to enhance functionality.

Overall AMCL algorithm is very easy to program and control compared to other localization algorithms like Kalman Filter. Although map used for this project was static in nature, AMCL would still work well in dynamic environments especially in indoor settings like home applications where changes are more restricted compared to outdoors. In terms of kidnapped robot case, AMCL performance seems mixed. For example after reaching goal position, if we reset robot position even though it localizes itself again, it seems slower to converge than expected.

7 Conclusion / Future work

Although both robot models reached the final goal position, some improvements can be still be done in terms of path they followed or overall time taken to reach. Particularly in this case, initially both robot tend to move in direction opposite, then after some time to reverse back to converge on expected path. This can be corrected with some more parameter tuning.

Also some work can be done to further enhance custom robot model, may be adding a gripper/arm or even make it four-wheeled.

References

[1] "Robot localization i: Recursive bayesian estimation," Sep 2017.
[2] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust monte carlo localization for mobile robots," *Artificial Intelligence*, vol. 128, no. 1-2, p. 99141, 2001.
[3] K. Zheng, "Ros navigation tuning guide," Sep 2016.