Robot Localization Using ROS

Vijayasri Iyer

Abstract—This paper presents an attempt to solve the task of robot localization in a simulated ROS environment. The robot performs this task using the Adaptive Monte Carlo Localization (AMCL) algorithm in the Jackal simulator provided by Clearpath Robotics, which is are readily available as a ROS package. The robot is successfully able to localize itself, in the given environment with proper tuning of a few important parameters.

Index Terms—Robot, IEEEtran, Udacity, LaTeX, Localization.

1 Introduction

In robotic systems the navigation problem consists of three important sub-problems; localization, mapping and path-planning. Localization is the process of a robot estimating it's location in an environment, based on a combination of it's prior knowledge of the environment and sensor measurements. This paper describes the process of a custom made robot that localizes itself in a ROS (Robot Operating System) environment, using the Adaptive Monte Carlo Localization or AMCL algorithm. This algorithm along which is available as a readily available package, is successful in localizing the robot by tuning the parameters efficiently.

2 Background

As mentioned above, localization is the challenge of estimating a robot's pose in an environment given a map of the environment, and is one of the main areas of focus in robotics research. This is because, unlike a simulation which is an extremely controlled environment, real-life situations have result in much more noisy sensor measurements. The localization problem is solved using a combination of two methods 1)leveraging sensor data 2)using an approximation algorithm to filter out all of the improbable locations of the robot. There are various approximation algorithms available to solve this problem, the most common ones being the Kalman and Particle Filters. These algorithms are discussed in detail in the sections below.

2.1 Kalman Filters

The Kalman filter is a type of approximation algorithm, named after it's creator Rudolf E. Kalman. This algorithm has been used widely in a variety of navigation and control systems, including the Apollo Program in 1960. It is used to measure the state of the system when the measurements are noisy. This filter uses two main functions in an infinite loop namely state prediction and measurement update, which is also known the predictor-corrector logic. These two steps are used to recursively update the mean and variance of the initial gaussian distribution until it arrives at a distribution with a very high mean and low variance. The Kalman filter exclusively operates when all it's variables are Gaussian Distributions. There are many variants of the Kalman filter namely the 1)Multidimensional Kalman filter (linear) 2)Extended Kalman filter (non-linear) and 3)Unscented Kalman filter (highly non-linear). For real-world tasks, since they are non-linear in nature, the extended and unscented variants of the filter are more likely to provide better results.

2.2 Particle Filters

A particle filter basically uses the principle of a probability distribution to eliminate the unlikely poses (position + orientation) of a robot. Here, the current robot pose is represented using three parameters the x-coordinate, y-coordinate, and an orientation theta with respect to the global frame. In a particle filter, particles are initially spread randomly throughout the entire map. these particles are not physical, they exist only in simulation. Even the particles have x,y and theta coordinates. Hence, each particle represents a hypothesis of the robot being in that particular location. In addition, each particle has a weight which represents the difference between the location of the particle i.e predicted pose and the actual location of the robot. Importance of a particle depends on its weight, which indicates its likelihood to survive a resampling process. After several iterations of MCL and several stages of resampling the particles will converge into a probable robot location. Particle filters are said to be better than Kalman filters in situations where the belief is said to be multimodal ie we have to deal with non-gaussian distributions. The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robots position and orientation based on sensory information. This process is known as a recursive Bayes filter.

2.3 Comparison / Contrast

A comprehensive explanation of the difference between the above approximation algorithms is given below in the form of a table.

Kalman Filter	Particle Filter
Linear/Non-Linear	Non-Linear
Less flexible	More flexible
Lower computation cost	Higher computation cost
Solution is stable	Solution is relatively unstable
Only Gaussian distribution	Any arbitrary distribution

As mentioned in the above table, even though particle filters are computationally expensive, they are better suited to noisy and non-linear environments and work with non-gaussian distributions. Hence, for this task the AMCL algorithm is used consisting of a particle filter. AMCL or Adaptive Monte Carlo Localization is a variant of the MCL algorithm which dynamically adjusts the number of particles over a period of time, as the robot navigates around. This adaptive process offers a significant computational advantage over MCL.

3 SIMULATIONS

This project used the Jackal simulator created by Clearpath Robotics for testing their mobile base robot Jackal. It consists of a simple maze-like environment.

3.1 Benchmark Model

3.1.1 Model design

The benchmark model is a very simple differential drive mobile robot with four wheels; two for driving and two castor wheels for support. The size of the robot is quite small and it has two sensors attached; a 2D stereo camera and a Hokuyo Lidar. An image of the benchmark model is provided below in fig 1.

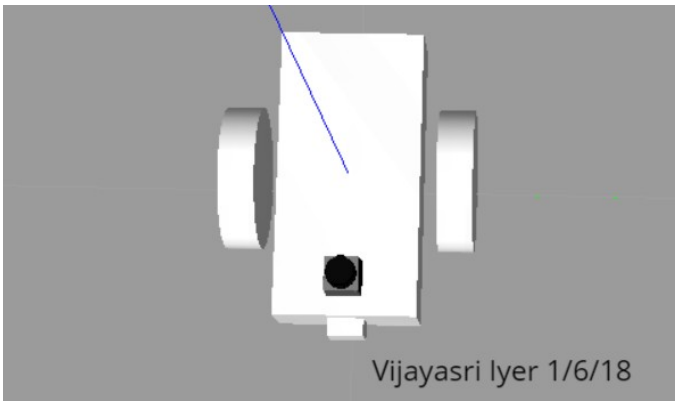

Fig. 1. Benchmark Model.

3.1.2 Packages Used

This project extensively uses the ROS navigation stack which contains a number of packages. The **amcl** package provides a readymade implementation of AMCL to localize the robot. It takes in a laser-based map, laser scans, and transform messages, and provides pose estimates as output. It subscribes to the scan, tf, initiapose and map topics and publishes to the amcl pose, particlecloud and tf topics. The amcl package provides global_localization and request_nomotion_update services which are further explained in the ROS wiki pages.
The **move_base** node in the move_base package, a ROS interface for configuring, running, and interacting with the navigation stack on a robot. It combines together a global and local planner to accomplish its global navigation task. The move_base node subscribes to the topics move_base_simple/goal, geometry_msgs/PoseStamped and publishes to the cmd_vel topic.
The **map_server** package provides the map_server node which allows ROS to use map data as a service.
The costmap_2D package is also used to build a 2D occupancy map of the given data.
The **base_local_planner** package has a controller that produces velocity commands to send to a mobile base. It publishes to the global_plan, local_plan and cost_cloud topics, and subscribes to the odom topic. The role of each package in the navigation stack is shown in the image provided below.

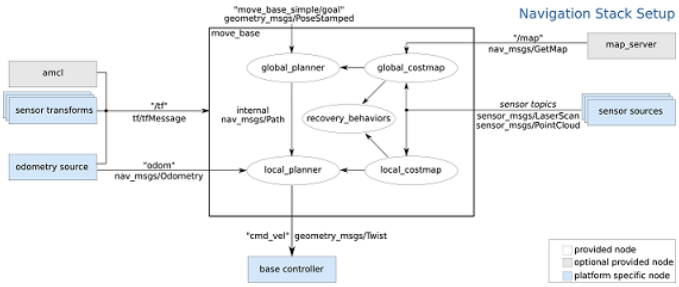

Fig. 2. Navigation Stack.

3.1.3 Parameters

In order to get the robot to localize itself correctly, the parameters of the **amcl** and **move_base** package must be tuned properly. The params of the amcl package are present in the launch file and the move_base params are in the config files.
Following are the amcl parameters:

- min_particles : 10
- max_particles : 300 (range of particles to represent the pose of the robot)
- transform_tolerance : 0.2 (rate at which transforms are published)

Following are the move_base (config file) parameters:

1. Base local planner params

- controller_frequency (move_base) : 10 (freqeuncy at which velocity commands are sent to move_base package),
  (Rest are part of the TrajectoryPlannerROS API)
- min_vel_x : 0.45
- max_vel_x : 0.1
- max_vel_theta : 1.0
- min_in_place_vel_theta : 0.4
- escape_vel : -0.5
  (all of the above define velocity limits of the robot)
- meter_scoring : true (whether the gdist_scale and pdist_scale parameters should assume that goal_distance and path_distance are expressed in units of meters or cells)
- pdist_scale : 2.0 (weight for how much the controller should stay close to the path it was given)
- gdist_scale : 0.5 (weight for how much the controller should attempt to reach its local goal)
- occdist_scale : 0.01 (weight for how much the controller should attempt to avoid obstacles)

- heading_lookahead : 1 (how far to look ahead in meters)
- acc_lim_theta : 3.2
- acc_lim_x : 2.5
- acc_lim_y : 2.5
(acceleration limits for the robot)
- global_frame_id : odom (Should be set global frame of local costmap)
- prune_plan : true (defines whether or not to eat up the plan as the robot moves along the path)

2. Common costmap params

- map_type : costmap
- obstacle_range : 0.3 (maximum range sensor reading that will result in an obstacle being put into the costmap)
- raytrace_range : 0.3 (range to which we will try to clear out in front of it, given a sensor reading)
- robot_radius : 0.5 (added padding to the robot to prevent collision)
- inflation_radius : 2.0 (added padding to the obstacles to prevent collisions)

3. Global/local costmap params

- global_frame : map/odom
- robot_base_frame : robot_footprint
- update_frequency : 5.0 (frequency in Hz for the map to be updated)
- publish_frequency : 5.0 (frequency in Hz for the map to display published information)
- width : 10.0/5.0 (dimensions of the map)
- height : 10.0/5.0
- resolution : 0.05 (resolution of the map)
- static_map : true/false (whether or not the costmap should initialize itself based on a map served by the map_server.)
- rolling_window : false/true

3.2 Personal Model

3.2.1 Model design

The custom model was desgined based on the benchmark model with a few additional changes. This model had an additional platform on top of the base platform, with the camera sensor placed on top of this new platform. An image of this model is provided below fig 3.

3.2.2 Packages Used

The packages used for this model are identical to the packages used for the benchmark model.

3.2.3 Parameters

The parameter values of the config files and the packages also remain the same for this model.

3.2.4 Achievements

The robot responded relatively well to the parameter tuning, as it sucessfully reached the goal position. The parameters that has the most impact on performance are the inflation_radius and robot_radius params. Setting the inflation radius to a relatively large value helped the robot in avoiding collisions with the obstacles. The g_dist and p_dist and occ_dist params are also important to decide the robot's affinity towards the goal and tendency to avoid obstacles and stay on track. The obstacle range and raytrace range params aren't very useful in this case since there are no obstacles introduced into the costmap. The next important params are the dimensions of the costmap, that make it easier to navigate and the controller, update and publish frequency. The transform tolerance and the min max particles also have a great impact on the result, since a better particle threshold and tolerance helps in convergence.

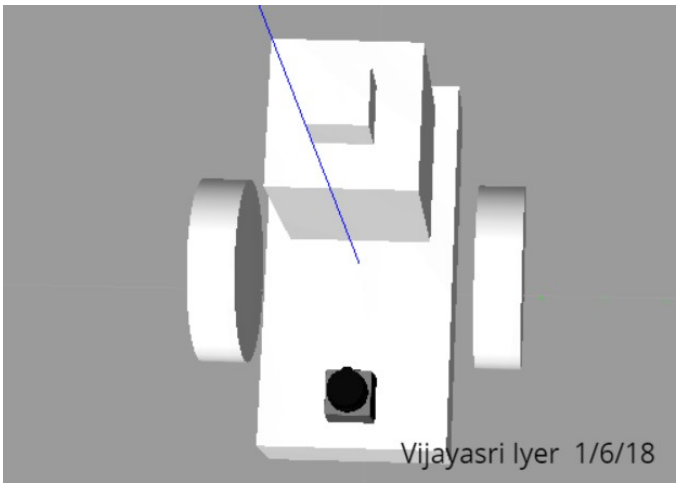

Fig. 3. Personal Model.

4 Results

The results of the experiment are moderately satisfying. The robot is able to navigate to it's goal position without having to deviate too much from the projected path, although at times, it does get stuck and rotate 360 degrees around its own axis before continuing to follow the same path. The robot takes a while to reach it's goal position, and the particles converge a little during this entire process, but there is definitely room for improvement, since the particles aren't centered around the position of the robot, indicating uncertainty. In fig 6, an image of the robot navigating in the environment is shown below.

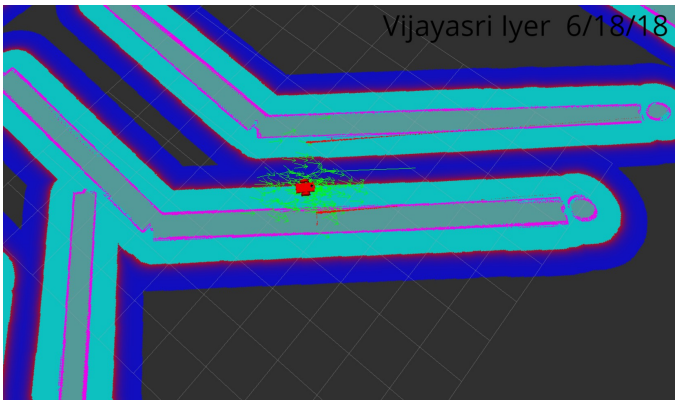

Fig. 4. Robot Navigating.

4.1 Localization Results

The images of the benchmark and personal models at the goal position of the map defined in the navigation_goal script, with the pose array are given in fig 4 and fig 5.

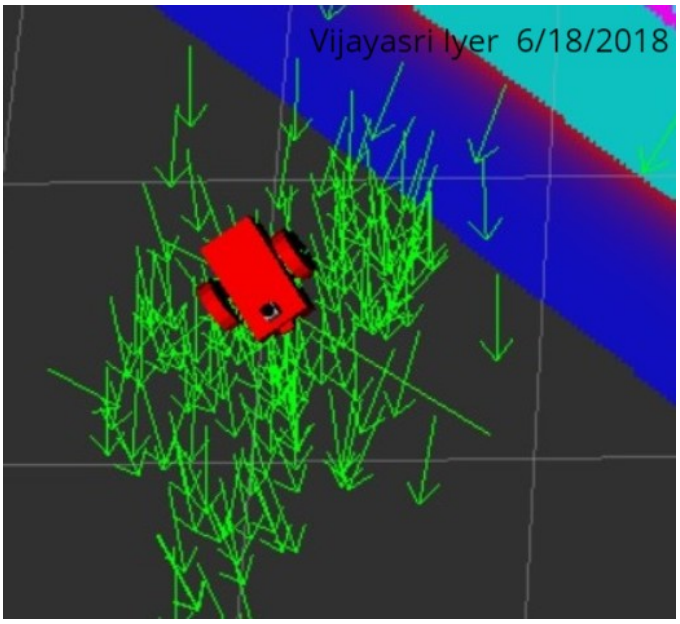

Fig. 5. Benchmark Model.

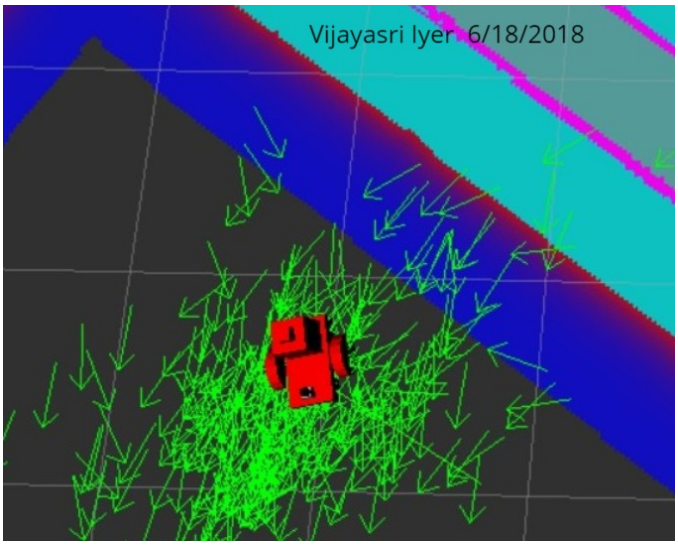

Fig. 6. Personal Model.

4.2 Technical Comparison

The benchmark and the personal models differed because the personal model had an extra platform on top of the base platform, resembling the layout of a service robot which is generally taller in terms of height. The parameters are identical in both cases. The personal model performed slightly worse than the original model, since it took more time to get to the goal and deviated from the path a couple times. This might create problems if the robot is deployed in a sensitive area such as a house crammed with furniture, because that could lead do damage. The time factor however, isn't much of an issue in case of these robots. The more important issue here is to navigate to the goal position correctly with little emphasis on time.

5 Discussion

The overall performance of both the personal and benchmark robot have some issues navigating and take a lot of time to reach the goal. This could be due a lot of reasons. In case of the benchmark robot, this can probably be improved with the tuning of parameters. For the personal model in specific, the sensor position and layout could also be posing an issue. This is because a part of the camera scene is blocked by the lidar sensor. Hence, including additional parameters, tuning them and slightly changing the layout of the personal model might help improve the quality of the experiment.

5.1 Topics

- Which robot performed better?
  The benchmark robot.
- Why it performed better? (opinion)
  Primarily due to the position of the sensors.
- How would you approach the 'Kidnapped Robot' problem?
  The Kidnapped robot problem describes a situation where the robot is switched to an arbitrary location at any time during the localization process. Particle filters do not work in this case. Hence, using SLAM algorithms which leverage visual odometry such as loop closure with SURF might be useful.
- What types of scenario could localization be performed?
  Localization can be performed better in a well-defined, relatively small indoor environment like a residential area, office space etc. Outdoor environments can use GPS for additional data.
- Where would you use MCL/AMCL in an industry domain?
  In warehouses, where worker robots have to pick up and drop off objects at specific places and navigate from the one spot to the other.

6 Conclusion / Future work

This project was an attempt to get a robot to localize and navigate to the goal in a well-defined environment. A certain amount of noise was also provided to mimic the real-world scenario. Hence, the results of this experiment will allow the robot to navigate in similar environments such as a home. Future improvements to this project, would include designing a completely new robot to solve a specific navigation task. Also, fine tuning the parameters further, to allow better convergence of particles. Another improvement would be to switch to a different kind of environment, larger and with more obstacles, so as to accurately mimic a real world environment. This solution can then be implemented on an embedded system such as the Jetson TX2 to be deployed on a physical robot.

6.1 Modifications for Improvement

- Increase the base dimensions
- Place the Lidar and cameras in such a way that they don't obstruct each other.
- Add an extra camera at the side or back of the robot to enable better data collection.

6.2 Hardware Deployment

1) What would need to be done?
   Follow the instructions given the course material for implementing the solution on jetpack running ROS.
2) Computation time/resource considerations?
   Since the Jetson TX2 is a GPU augmented system, computation time not be much of an issue. It also has a built in camera for vision. It would be useful to add the Lidar unit to the setup for improved visual sensing.