Where Am I?

Nagarjun Redla

Abstract—This paper describes the process of localizing a robot in a simulated environment for the Udacity Nanodegree program. The simulation is done using Gazebo and Robot Operating System (ROS) and the localization technique used is Monte Carlo Localization using the ROS amcl package.

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

1 Introduction

KNOWING where a robot is located in the world is a very important aspect of building autonomous robots. The ability of a robot to know where it is helps in navigating through different environments. For example, a common home robot like the Roomba will have to navigate through a room when cleaning, and knowing where the robot is helps plan its path through the room and to navigate. There are multiple ways of localization, but the method we use here is called Monte Carlo Localization using Particle Filters.

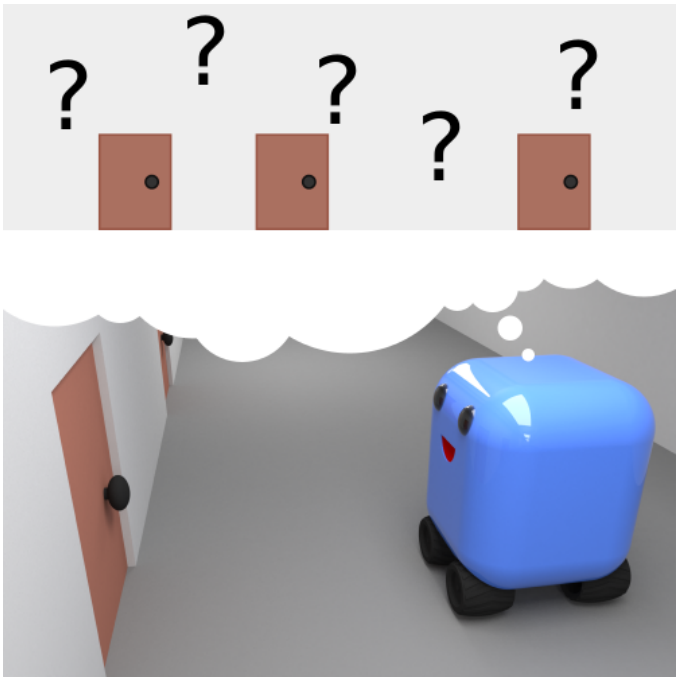

Fig. 1. Robot Localization. [1]

2 Background

Estimating the position of a robot is a very important aspect, given measurement uncertainty and noise of sensors. There have been different algorithms developed for localization, and the most widely used localization methods are Particle filters and Kalman Filters. Each has it's own advantages and disadvantages summarized in table 1.

TABLE 1
Kalman Filters vs. Particle Filters

	Particle Filters	Kalman Filters
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Memory & Resolution	Yes	No
Global Localization	Yes	No
State Space	Multimodal Discrete	Unimodal Continuous

2.1 Kalman Filters

Kalman Filters, as the name suggests, are named after Rudolf E. Kalman, although a similar algorithm was proposed earlier by Thorvald Nicolai Thiele and Peter Swerling. [2] This algorithm helped estimate the trajectory for the Apollo mission. There are multiple variants of the Kalman Filter called the Extended Kalman Filter (non-linear systems) and the Unscented Kalman Filter (highly non-linear systems)

2.2 Particle Filters

Particle filters are a set of Monte Carlo algorithms used in signal processing as filtering methods. Particle filters are widely used in robotics because of their simplistic implementation on discrete devices like digital computers.

2.3 Comparison / Contrast

As summarized in table 1, both filters have their own advantages and disadvantages. Since the model being dealt with in this project is a differential drive robot, the system is linear and Particle filters are fairly easy to implement.

3 Simulations

The simulations are done using Gazebo and RViz. An already mapped environment is provided by Udacity. The process of mapping is discussed in the final project of the Nanodegree. Gazebo simulations can be quite taxing on the host system, and running on systems without a GPU usually results in Gazebo crashing, jittery performance, frustration and in my case a laptop battery deterioration. Using a Jetson

TX2 will help significantly for people without access to a GPU system for offline simulations. Otherwise the best way to go about it would be to use Udacity's provided GPU workspaces or to spin up a VM on your own (not an easy task for inexperienced users).

If you were to replicate my project on your own system, keep in mind to change parameters that are affected by the speed of your system, like transform_tolerance. The lowest power system I used is the Jetson TX2 and the highest is the Udacity workspace. I keep switching between them depending on where I am running the project so you should try to check once before you run it.

3.1 Achievements

Both the Benchmark and Personal models were able to localize themselves in the given environment.

3.2 Benchmark Model

3.2.1 Model design

The benchmark model used is called the udacity bot. The model parameters are as described in the Nanodegree program lessons. It is a cuboid shaped robot with wheels, a hokuyo sensor and an RGB camera, as shown in picture 2.

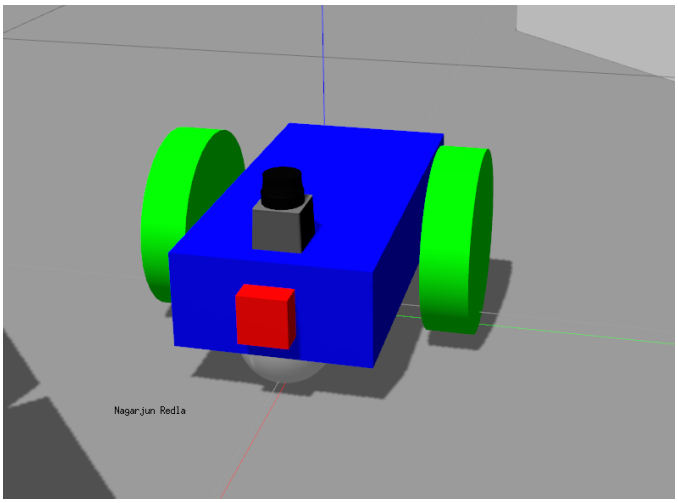

Fig. 2. Udacity Bot Model.

3.2.2 Packages Used

This project uses the move_base package for navigation and amcl for localization. The robot publishes information about odometry on /odom, camera on /udacity_bot/camera1/* and laser scans on /udacity_bot/laser/scan topics respectively. The other packages like move_base, gazebo and amcl publish and receive on their own topics. All topics can be listed by running 'rostopic list' after starting all nodes.

3.2.3 Parameters

Most of the parameters are common for both robots. Udacity bot has a robot radius of 0.25 in costmap common parameters. The common parameter changes are all present in the amcl.launch file as shown in table 2.

The parameters for min and max particles are tuned for the system, the max particles could be larger but the system starts to lag because of computational load. The odom alpha parameters is the uncertainty in the odometry information being published. The default parameters were tuned for turtlebot and had to be tuned for the robots in the project. Update min a is the angle of rotation after which an update is supposed to be performed and update min d is the distance for which a localization update is supposed happen.

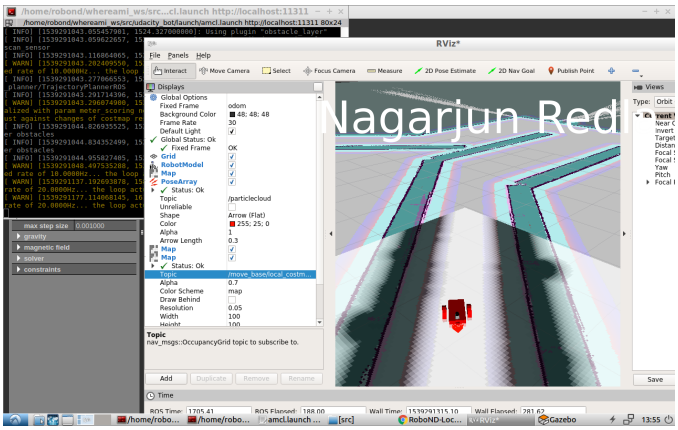

Fig. 3. Udacity Bot Navigating.

TABLE 2
AMCL Parameters

Parameter	Value
min_particles	15
max_particles	200
initial_pose_x	0
initial_pose_y	0
initial_pose_a	0
odom_alpha1	0.02
odom_alpha2	0.02
odom_alpha3	0.02
odom_alpha4	0.02
laser_max_beams	100
laser_max_range	15
update_min_d	0.05
update_min_a	0.01

3.3 Personal Model

3.3.1 Model design

Since this project builds on to the personal home robot project, a circular robot that looks similar to a Roomba was chosen. Its parameters are slightly different from the Udacity bot benchmark model. Since this project is followed by the SLAM project, I wanted to add my favorite RGB camera at the moment, the Intel Realsense D435. The models found on the official Intel Github page were very big (24 MB), so I decided to borrow another lighter model used in a fellow Udacity Nanodegree student's repository.Turns out that the lighter model runs fine on Udacity's provided workspace, but does not work one the NVidia Jetson TX2 as it is pretty compute intensive. The SLAM version can be found in my Github repository.

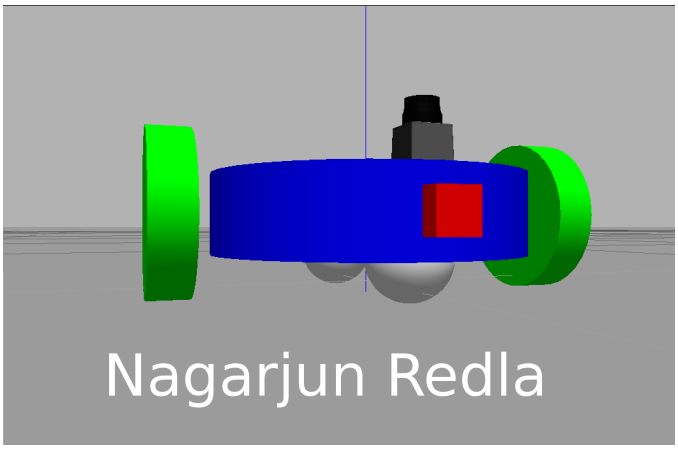

Fig. 4. Circle Bot Model.

3.3.2 *Packages Used*

The packages used were the same as the Udacity bot, but with a a change in robot radius in costmap common parameters to 0.35 since it is wider.

3.3.3 *Parameters*

The parameters were accounted for the change in dimensions and form factor of the robot. The inflation radius had to be changed since the robot is much wider than the Udacity bot since it is circular.

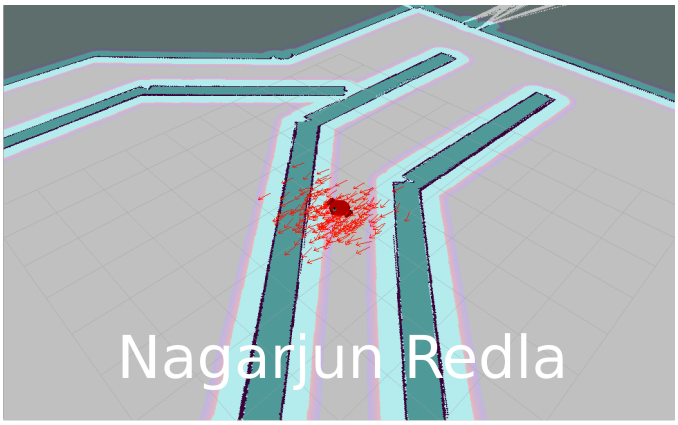

Fig. 5. Circle Bot during Initialization.

4 Results

The Particle filter algorithm begins by initializing random particles and then slowly adjusts/creates/destroys particles based on their accuracy from subsequent sensor measurements. This means that the robot's initial position is very uncertain and so the robot needs to move a bit in order to update its estimated position. Running navigation_goal without moving the robot even a little bit might lead to the robot getting stuck at a wall. Once the particles converge to the robot's position it navigates quite easily.

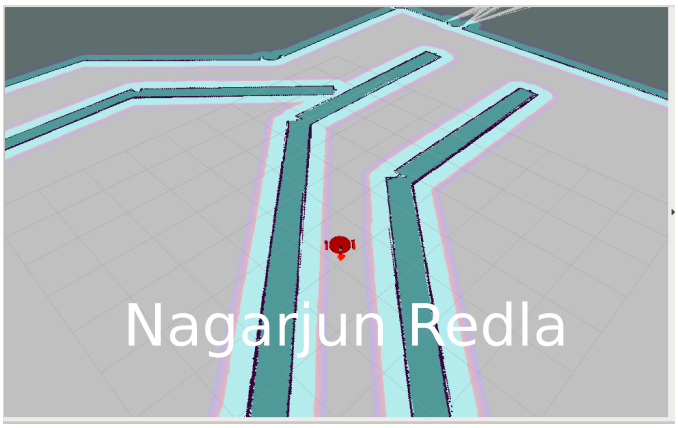

Fig. 6. Circle Bot after localization.

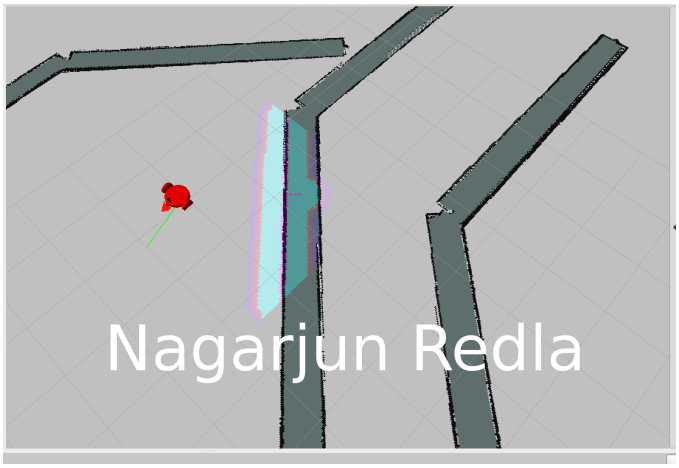

Fig. 7. Circle Bot reching goal.

4.1 Localization Results

4.1.1 *Benchmark*

The benchmark robot performed pretty well soon after configuring parameters. It is a small robot with a simple design. It reached the navigation goal without any hassles

4.1.2 *Student*

The circle bot was pretty wide since it was a circular design. It was more compute intensive to run a simulation using a circular chassis and it kept missing control loops on inferior hardware. The robot being wide didn't help in cases where it had to turn in narrow spaces before particles converged.

4.2 Technical Comparison

Using a circular robot platform improves the used space based on the amcl package used for simulation. The amcl package takes in a robot radius parameter and using a non-circular robot leads to unused portions. A circular design also helps navigate around corners instead of wedging itself to walls.

5 Discussion

This report shows different robot models localizing themselves using particle filters in simulations using ROS. The

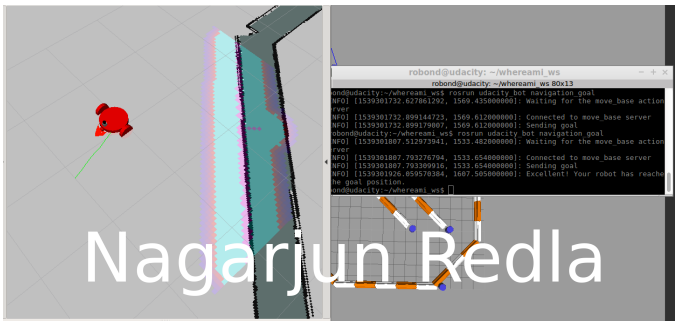

Fig. 8. Circle Bot with navigation_goal message.

robots were able to localize themselves pretty well in simulations but there might be different results in real life on the field. Sensors used in real life do not always come with a constant error distribution and might be more erratic than assumed in simulations. Wheel odometry is known to have drift as readings are taken because of wheel slip, etc. The most difficult part of the entire project was to find a consistent offline system capable of running ROS and Gazebo without failing. Udacity provided VMs weren't powerful enough for host systems without GPUs for Term 2 projects unlike the ones in Term 1 of the Nanodegree program. It is very interesting to see algorithms developed decades ago perform so well.

5.1 Topics

- Which robot performed better? The Udacity bot performed better than the circle bot because it was small enough to navigate through the narrow path in the middle of the map where it spawns. Making a smaller circular bot would make it navigate better. Circular shapes could also have been more compute intensive.
- Why it performed better? (opinion) As metntioned earlier, the circle bot was exactly as wide as the narrow path in the middle. To overcome getting stuck at a wall, I suggested earlier to try moving the robot a bit with the 2D nav goal option in RViz, usually a rotation operation will do the trick.
- How would you approach the 'Kidnapped Robot' problem? Monte Carlo Localization techniques directly don't solve the 'Kidnapped Robot' problem. Although hybrid techniques can be used [3], the other way is to use Kalman Filters, which use Landmarks for localization.
- What types of scenario could localization be performed? Localization can be performed in a feature rich known environment. For example, a long corridor with parallel walls and solid colors will not do good for localization since the robot will not have any new features to map to, although odometry information is given.
- Where would you use MCL/AMCL in an industry domain?

6 Conclusion / Future work

Different robots successfully localized themselves in known environments. The next step would be to work on the SLAM implementation project in the Nanodegree program. A good challenge would be to implement it on an actual robot. There is a robot I built with an RGB-D camera which could be used to generate a laser scan (in order to avoid high fee on a lidar) and that could be used to localize itself. The main problem though is that the move_base package does not work directly for holonomic robots like the one I am working on with omni-directional wheels. ROS has a vast community which has enough information to get started on omni-wheel robots though. The robot has been tested to run ORB_SLAM2 and should be able to handle RTAB-MAP pretty well.

6.1 Modifications for Improvement

- Base Dimension: As mentioned before, the base dimension could be reduced for the circle bot, but imagine trying to make an actual Roomba, you might need a base as big as that for a prototype cleaner to fit the different mechanical systems in it before making a more fine tuned final production phase design.
- Sensor Location: The sensor locations look fine to me, although in real life we see the RGB camera pointed upwards in indoor robots like the Roomba. This is to maintain consistency in mapping the environment. The robot will find it harder to localize itself if the ground truth environment keeps changing frequently. Think about it, you might move stuff on the floor of your room around, but when what the last time you made a significant change to the layout of your ceiling?
- Sensor Layout: Since this robot mostly moves forward, the Lidar is placed perfectly. There are a few products like the Xiaomi and Neato home service robots feature Lidars in the front. As mentioned earlier the camera could be pointing upwards for consistent mapping and localization.
- Sensor Amount: This entirely depends on final device cost constraints. There are a wide range of Roomba devices on the market. The least expensive ones don't have either a Lidar or a Camera! The lowest end ones use close distance sensing infrared sensors for obstacle detection and edge detection to avoid falling over edges. As you traverse up the price ladder of home robots, you will first notice cameras starting to pop up, this is because they use visual odometry (or visual SLAM) which could be optimized to be less compute intensive. The most expensive ones (and only in a few brands) end up including a cheap scanning Lidar (Which costs around $100 if bought separately [significantly cheaper than a Hokuyo]) since processing noisy pointclouds might need more compute resources. As other industries like self driving cars develop further and demand for home service robots increases, we might see a drop in prices of sensors and maybe every device will have all sensors. It is usually good practice to critically design a product for its intended requirements and

not to overdesign it since its a waste of resources, we might phase out of the Lidar if visual odometry methods become significantly faster. Another important thing to note is that the amcl package only utilizes lidar scans and doesn't take into account RGB camera readings.

6.2 Hardware Deployment

1) What would need to be done? Hardware implementation of ROS based robots isn't as simple as one might think. To begin with, ROS needs an on-board computer if using the bot as a standalone device. In order to get reliable odometry information from the wheels, we need time critical hardware like a microcontroller dedicated to execute motor control commands and compute odometry information from wheel encoders. An arduino running rosserial should do the trick. The other sensors like RGB camera and Hokuyo sensor usually come with common interfacing options like USB that directly plugs into the on-board computer. I would recommend teaming up with at least one other person while building an ROS based robot since going back and forth between microcontroller and microprocessors (and not to forget different programming languages) might be mentally compute intensive if you're not used to it.
2) Computation time/resource considerations? When deploying this on a standalone robot, we wouldn't need to run Gazebo simulations on the robot itself. If we needed to see the simulation you could try running a host computer (not on the robot) as a master that the robot publishes navigation information to. Since all sensor measurement tasks are offloaded to the sensors themselves, the computer would only need to run the AMCL and move_base packages. As discussed earlier, the Kalman Filter algorithm ran on Apollo's computer which used 2k of magnetic core RAM and 36k wire rope and a clock speed of under 100kHz. A raspberry pi could do the trick to run such a relatively simple ROS stack. There are many other options of single board computers on the market today like Udoo, Aaeon UP board and NVidia Jetson TX2.

REFERENCES

[1] "Wikipedia: Monte carlo localization,"
[2] "Wikipedia: Kalman filter,"
[3] S. Thrun, "Particle filters in robotics,"