The Where Am I Project

Shivang Tripathi

Abstract—This paper presents the workflow for developing the robust navigation for mobile robotics applications. The content discusses the development of the navigation system for an off-the-shelf robot as well as a custom robot model in Gazebo and the integration of ROS packages for their localization and navigation. Stress is paid on the fine tuning of the default parameters of the ROS Navigation Stack to achieve the the best possible performance with the mobile robots.

Index Terms—AMCL, ROS, Gazebo, Navigation Parameters Tuning, Localization.

1 Introduction

ROBOT localization refers to the ability of a robot to determine its pose(position and orientation) w.r.t the frame of reference of the map. Robot Navigation is the ability of a robot to localize itself in a map and plan its path to some goal pose in the map. Thus, robot localization is essential for the robot navigation. A common analogy to understand this that if someone wants to reach some a destination place in the map, one should know their current place in the map to plan a path to the destination place. Autonomous navigation is when the robot figures out a path to the goal all by itself. This autonomous navigation can be implemented by using the 'ROS Navigation Stack' which , on a conceptual level, seems easy to be used. However, implementing it on an arbitrary robot model is challenging.

The main aim of this project was to understand the process of **fine tuning the parameters** of the *ROS Navigation Stack* so that the robot leveraging it can perform navigation in a robust fashion. This was first achieved on the off-the-shelf *udacity_bot* and was then achieved on a custom robot in a given map to make it navigate from the initial pose to the desired goal pose. It was simulated in Gazebo and the data was visualized in RViz.

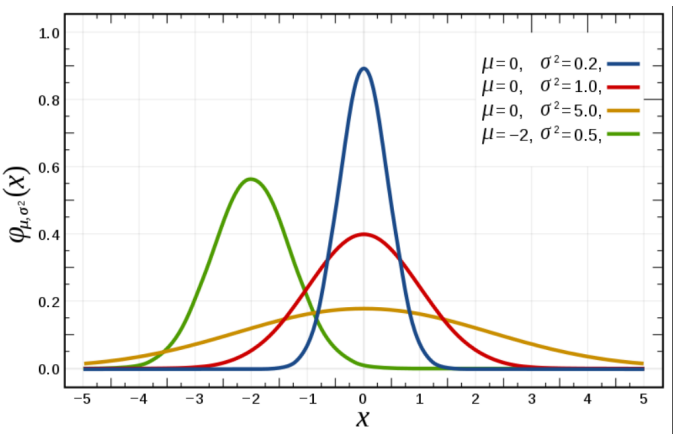

Fig. 1. A Gaussian Distribution

2 Background

Navigation is not possible unless the current pose of the robot is known. The process of finding the current pose of robot in the map(given or self generated) is called localization. The data received by the robot through various sensors is often noisy as well as the actuation commands sent to the robot are executed by the robot with some degree of error. For example: If a command is sent to the robot to move the robot by 5 meters, then the actual distance moved by the robot will be close to 5 meter mark, either before or after the 5 meter mark. Such type of situations can be modeled with the help of the Gaussian distribution.

2.1 Costmap

The costmap is an essential part of the robot navigation because the planned path generated will depend on the values of the costmap. It is an occupancy grid that is updated by the incoming readings from the sensors. It inflated the area around the obstacle according to the user-specified parameters.

ROS Navigation uses 2 costmaps to store the information about the obstacles around in the environment.

- **The Global Costmap**: Stores the *long-term plans* about the movement.
- **The Local Costmap**: Used for the *local path planning* and **obstacle-avoidance**

2.2 Kalman Filters

The Kalman Filter is an algorithm that uses the:

1) System's dynamic model
2) Known control inputs
3) Sensor readings

to approximate the state(and other varying properties) of the system that is better than the estimate obtained by only one sensor reading alone. This is the important algorithm for *data fusion* and *sensor fusion*. There are 2 **Variants** of the Kalman Filter for more generalized cases:

1) Extended Kalman Filter(EKF): for modeling the *non-linear* system.
2) Unscented Kalman Filter(UKF): for modeling *highly non-linear* systems.

These use the same underlying algorithm but the non-linear function is replaced its a linear function as approximated using its Taylor Series.

2.3 Particle Filters

The main algorithm is as follows:

1) Simulate n number of particles.
2) **Motion Update** Send the motion command to the robot.
3) Give the same motion command to all the simulated particles.
4) **Measurement Update**: Read the noisy data from the sensors.
5) Assign weights to all the particles. Higher the similarity to the robot's measurements, larger the particles weights.
6) **Resample** using the *resampling wheel*.

2.4 Comparison / Contrast

Perhaps the biggest difference between MCL and the variants of the Kalman filter is the **non-parametricity** of the particle filter central to the MCL. This means that the MCL can **approximate a vast variety of probabilistic distributions as well as multimodel situations** whereas the variants of the Kalman filter(EKF and UKF) can **only model the distribution that closely resemble the Gaussian distribution and performs poorly when belief is multimodal**. This report will only be showing the localization performance of the MCL algorithm.

Also, the computation complexity of the Kalman Filter is close to a constant [1] whereas the computation complexity of the MCL is close to linear. But in KF, the computation increases drastically when the distribution loses linearity. Thus, in the real world for modeling the non-linear probability distributions, particle filter is chosen with carefully tuned parameters as will be demonstrated in this paper.

Navigation is an essential core of robot software because this is what gives a mobile robot the very ability to be mobile. It is important because mobility opens up vast new areas for the application of the robotics in the consumer as well as the business space.

3 Simulations

Robot simulation is a cheap and fast way to validate the algorithms and parameters tuning of the various ROS packages. This is so because:

1) The real robot is not required for testing.
2) Spawning up updated instances of the robot is easy without the fear of damaging the robot.

This gives useful insights before actually building the robot using real costly hardware.

A 4 wheeled robot model with a LIDAR mounted on the top and a camera in front was modeled in *Gazebo*, the open source simulator used by ROS. This was controlled using the *skid_steer* plugin in Gazebo. Its parameters had to be tweaked to state the topics used for sending and receiving the commands to and from Gazebo robot model.

3.1 Achievements

Includes charts and graphs show how parameters affect your performance.

Below are the screenshots of the results achieve by the *benchmark model* and the *custom robot model*:

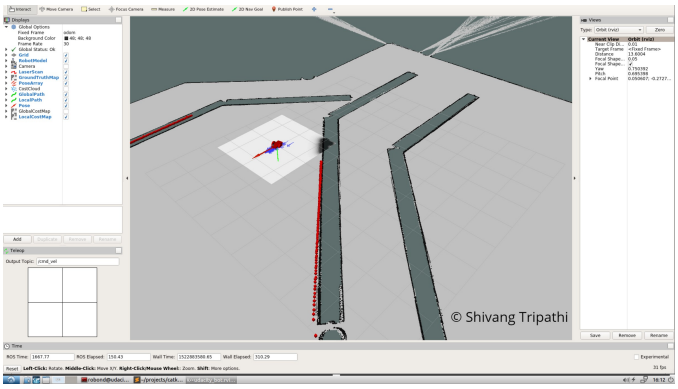

Fig. 2. Benchmark Robot Reached Desired Goal

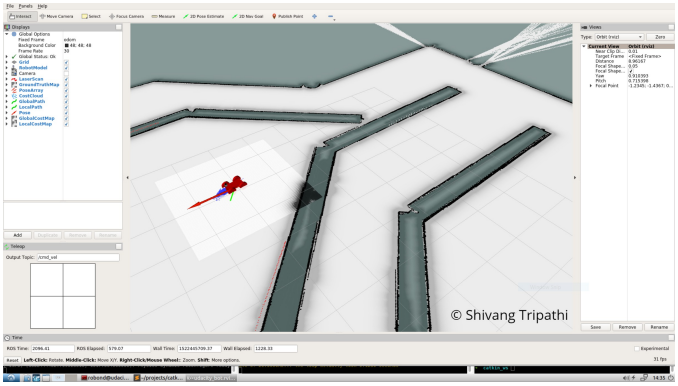

Fig. 3. Custom Robot Reached Desired Goal

The terminal displayed the message when any of the above robots reached their desired pose after localizing, planning and executing the planned path.

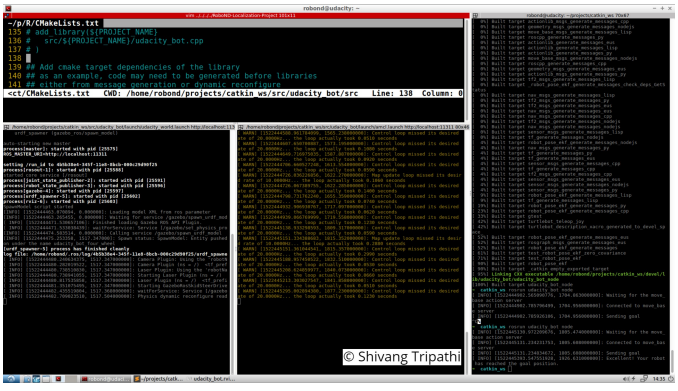

Fig. 4. Terminal when desired goal reached

3.2 Custom Model

3.2.1 Model design

The benchmark model was a basic robot model with

- Two wheel on each side
- A camera on the front
- A *Hokuyo* rangefinder on the top

A mesh file was used to visualize the rangefinder close to its real form in Gazebo. The total mass of the chassis was increased to prevent the robot jerks when higher velocities were specified.

A table showing the size of the *benchmark model*:

TABLE 1
Size of the benchmark robot model

Part	Mass	Size
Chassis	$50kg$	$0.4m * 0.2m * 0.1m$
Wheels	$5kg$	$r = 0.1m$
Camera	$0.1kg$	$0.05m * 0.05m * 0.05m$
Rangefinder	$0.1kg$	$0.1m * 0.1m * 0.1m$

3.2.2 Packages Used

The packages used for the navigation of the benchmark robot model were as follows:

1) AMCL : for localization of the robot
2) move_base : for sending
3) map_server
4) joint_state_publisher
5) robot_state_publisher
6) urdf_spawner
7) rviz

Three pre-built Gazebo plugins were used:

1) **Skid Steer Drive Controller plugin**: for controlling the four of the robot wheels by specifying *cmd_vel*.
2) **Camera Controller**: for receiving the camera feeds from Gazebo into RViz.
3) **Gazebo ROS Head Hokuyo Controller**: for controlling the Hokuyo rangefinder and publishing the laser scans on the relevant topic

3.2.3 Parameters

Localization parameters in the AMCL node should be described, as well as move_base parameters in the configuration file. You should be able to clearly demonstrate your understanding of the impact of these parameters.

There are 3 parameter configuration files: 1. Common parameters named *costmap_common_params.yaml* 2. Local parameters named *local_costmap_params.yaml* 3. Global parameters named *global_costmap_params.yaml*

Base Planner Configuration file : This configuration file is named *base_local_planner_params.yaml* which is used for tuning the parameters of the powerful *move_base* node. It is responsible for computing the command velocity for the mobile base after the plan is computed.

This guide played an essential role in the tuning of the parameters. It states the basic steps to follow to achieve optimal robot navigation.

The initial observation was that the robot does not follow the local generated map accurately.

1) Initial steep turns are executed perfectly.
2) But simple less steep turns are ignored.

The tolerance was suspected to be **too high**.

Odometry Check

The sanity checks:

- Setting *frame* to */odom*
- Setting *Decay Time* high (upto 30 seconds)

On rotating the robot, by publishing to */cmd_vel* , the *laser_scan* should fall on top of each other.

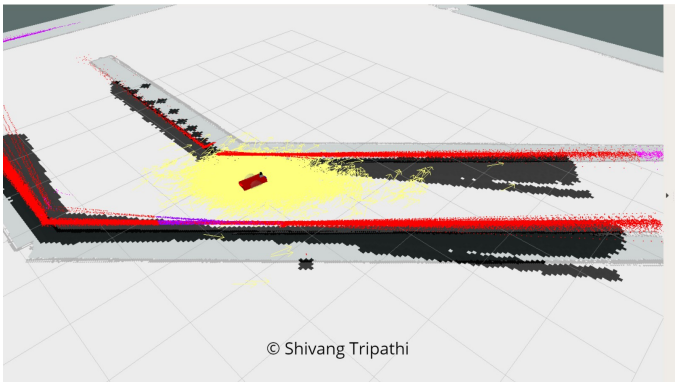

Fig. 5. The laserscans were approximately above each other

This test turned out to be successsful as the laserscans even after repeated fast robot rotations appeared on top of each other. Actually, the real problem lied at a very basic level. Simple publish on */cmd_vel* also leads to the same issue of

1) Nice robot rotation
2) Poor robot translation where the pitch of the robot changed rapidly leading to motion that was seldom straight even though velocity was specified in only the forward x direction.

Fiddling with the friction coefficients of the wheel joints did not lead to better results. After few hours, the problem was found to be the caster wheels. They were just big enough to make the wheels lose contact with the ground leading to the above-mentioned motion of the robot. Then, the odometry works fine, simple *cmd_vel* commands worked fine.

Tuning AMCL parameters

It was observed that the robot was able move to reach the goal position if the specified goal along the simple straight lines but not if around the other side of the wall.

As per the guide, running the *tf_monitor* revealed that the *tf* were being published at $57.8706Hz$.

The localization improved on adding the particle update parameter for translation *update_min_d* and reducing it from its default value to a very low value.

TABLE 2
Size of the benchmark robot model

Parameter	Value
update_min_d	0.01
update_min_a	$\pi/12$

This improved the particles accuracy in localizing the robot but they then appeared in discrete lines.

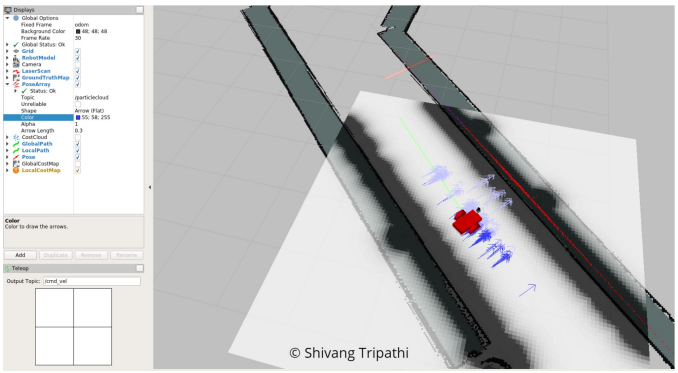

Fig. 6. Particles appeared in lines

A similar particle update parameter for rotation called *update_min_a* was added and reduced its value to $\pi/12$.

Also, added the *laser_min_range* parameter to prevent the detection of the robot wheels.

The tuning of the above two parameters greatly improved the localization seems to be better now but the local path planning is not optimal.

Tuning the move_base parameters

The planner used by move_base were:

- Local Planner: DWA_planner
- Global Planner: NavFn

Here the parameters were tuned using the DWA planner documentation.

To gain $5\ rads/s$, it took around 2 seconds, $2.5\ rad/s^2$ seems to be maximum angular acceleration.

Robot Does rotate until very close the wall. identify the parameter controlling this.

robot should turn at $1 - 1.5$ meters. Increased the *inflation_radius* from 0.9 to 1.9. Leading to better result.

Still robot seemed really slow. Thus, the max acceleration and max velocity limits in X direction were increased.

The robot kept banging into the wall. This was resolved tuning the parameters:

- *path_distance_bias*: 500 (default was 32) to keep it closer to the provided path
- *goal_distance_bias*: 420 (default was 24) to keep it closer to local goal
- *occdist_scale*: 0.95 (default was 0.01) weight to avoid obstacle by how much

While braking from higher speeds, the robot jerked to incline forward and backward leading to the detection of the floor as the obstacle by the rangefinder. Resolution being thought:

1) Increase damping of the wheel joints.
2) Reducing the obstacle_range from 5.0 to 2.0.

The latter option turned out to easily mitigate the issue. Path planner *dwa_local_planner, NavFn* seemed to work just fine with the above parameters.

3.3 Benchmark Model

3.3.1 Model design

The benchmark model was a basic robot model with

- A wheel on each side
- One caster wheel each on the front and the back
- A camera on the front
- A *Hokuyo* rangefinder on the top

A mesh file was used to visualize the rangefinder close to its real form in Gazebo.

A table showing the size of the *benchmark model*:

TABLE 3
Size of the benchmark robot model

Part	Mass	Size
Chassis	$5kg$	$0.4m * 0.2m * 0.1m$
Wheels	$5kg$	$r = 0.1m$
Camera	$0.1kg$	$0.05m * 0.05m * 0.05m$
Rangefinder	$0.1kg$	$0.1m * 0.1m * 0.1m$

3.3.2 Packages Used

The packages used for the navigation of the benchmark robot model were as follows:

1) AMCL : for localization of the robot
2) move_base : for sending
3) map_server
4) joint_state_publisher
5) robot_state_publisher
6) urdf_spawner
7) rviz

Three pre-built Gazebo plugins were used:

1) **Differential Drive Controller plugin**: for controlling the two of the robot wheels by specifying *cmd_vel*.
2) **Camera Controller**: for receiving the camera feeds from Gazebo into RViz.
3) **Gazebo ROS Head Hokuyo Controller**: for controlling the Hokuyo rangefinder and publishing the laser scans on the relevant topic

3.3.3 Parameters

The same parameters were used with the benchmark robot that gave satisfactory results. The robot was at times very jerky. All efforts to modify the parameters lead to worse performance. The best performance for the benchmark robot was achived by the parameters used by the custom robot. This was overcome in the custom robot by using the 4 wheel drive and *skid_steer* Gazebo plugin.

4 Results

Present an unbiased view of your robot's performance and justify your stance with facts. Do the localization results look reasonable? What is the duration for the particle filters to converge? How long does it take for the robot to reach the goal? Does it follow a smooth path to the goal? Does it have unexpected behavior in the process?

For demonstrating your results, it is incredibly useful to have some watermarked charts, tables, and/or graphs for the reader to review. This makes ingesting the information quicker and easier.

The screenshots of the robots reaching the goal pose was shown above that clearly show that both the robots:- *Benchmark robot model* as well as the *Custom robot* were able to finally reach the goal pose from the initial pose in which the robot was spawn. The particles as shown in RViz denoted that the robots were able to converge to a single point as the robot moved a few centimeters. This was slightly longer for the benchmark robot because its motion was jerky and consequently, the *laserscans* suffered slight distortion. This was not the case with the custom robot due to its better control using the four wheeled drive.

4.1 Technical Comparison

The custom robot worked better that the benchmark robot as factually evident from the time taken by both the robots to reach the desired goal pose from same initial pose.

94 seconds taken by custom robot. 121 seconds taken by the benchmark robot

5 Discussion

The custom robot worked better because it had better command on its position control. The 4 wheel drive was better than the 2 wheel drive with caster wheels that struggled to move straight at times. With the 4 wheels, the simple translation and rotation movements of the robot improved which also reflected in better navigation performance in reaching the desired goal as the robot localized. The *Kidnapped Robot* problem could be addressed by increasing the present value of *kld_error* from 0.000001 to a significant value. This will increase the generation of random particles at each iteration. So, if the robot is kidnapped, i.e., if the robot location is far from the high weight particles that converged at a distant location, the AMCL algorithm will still be able to recover by generation the random particles. Simple localization can be performed in areas where the map does not change at all, otherwise the updated map will need to be provided to the robot. Warehouses and factories can form the ideal scenario where localization can be useful and profitable as a robotics application.

6 Conclusion / Future work

The robot model itself can be improvised to give better navigation results. A drivetrain type mechanism must be specified to make the robot robust to jerky motion commands given to the robot.

- Make the simulation closer to the real world.
- Make the motion of the robot more smooth

Looking at other open-source robots models like Husky, the robot model description can be improved. This can lead to finding new clever concepts to improve the performance of the navigation as well as the localization of the robot.

References

[1] C. Montella, "The kalman filter and related algorithms: A literature review," 05 2011.