Project: Where am I?

Martin Papanek

Abstract—This paper describes the approach to setting up a robot from scratch in ROS, with Gazebo as the simulation environment, and having this robot navigate a known world using the Adaptive Monte Carlo localization (AMCL) algorithm along with the ROS Navigation stack. Two different robots are presented in order to illustrate the differences in the tuning of the AMCL and Navigation stack parameters. The goal of each robot is to arrive at a specific end point starting from an unknown location in the map. It is demonstrated which parameters are most important for good localization and navigation and what were the problems.

Index Terms—Robot, Udacity, Localization, Differential Drive, Skid Steer Drive.

1 Introduction

The goal of this project is to create two simulated robots in ROS and Gazebo and use the ROS AMCL and move_base packages to allow these robots to navigate to a target goal in a virtual, known world given an unknown starting position and noisy sensors. To achieve this a new ROS package is created and robots are specified using the Unified Robot Definition Format (URDF) with Gazebo plugins to give them the ability to use features like differential driving, camera and the laser scanner inside the Gazebo simulator.

To allow the robots to navigate inside the virtual world, the AMCL algorithm is used. The robot will start with an unknown pose and orientation with respect to the world origin and the Adaptive Monte Carlo algorithm will allow to robot to figure out a distribution over the most probable poses. Then to reach the target goal, the robot needs to be able to plan a trajectory and to avoid obstacles and for this the ROS Navigation Stack (move_base) is used.

2 Background

The problem of localizing in a known environment, is an important topic in robotics. To perform their tasks, robots need to know to some degree of certainty where they are located. As they move around the world and perform their tasks, again their change in position needs to be noted.

There are three variants of the localization problem which are: position tracking, global localization and kidnapped robot problem.

Position tracking is the most fundamental problem where the robot has a known starting position which it has to update as it is moving. The challenge is that the actuators and sensors of the robot all have a degree of uncertainty in their actions or collected data. Algorithms that solve position tracking must account for this.

Global localization is a more difficult version of position tracking which additionally involves the robot starting from an unknown position. The robot must look at landmarks in its surroundings and decide where it is, but like in position tracking there is noisy data.

Finally, the most difficult variant is that of the kidnapped robot. In the kidnapped robot problem, it may happen that the robot suddenly is moved to a new unknown location, so any algorithm that localizes the robot must be able to detect that the robot has been moved and must start again. This project focuses on the task of global localization in a virtual world. The virtual world used is the Jackal Race map. This map is provided as a SDF file and it is loaded into a ROS map server node from the robot launch files.

There are two popular algorithms for localization. The Extended Kalman Filter (EKF) and the Adaptive Monte Carlo Localization algorithm (AMCL). For this project, the AMCL was chosen because it can do global localization unlike EKF, and because is a simpler algorithm to implement. In the following subsections, Kalman filters and MCL are compared and contrasted.

2.1 Kalman Filters

The Kalman filter performs position tracking by starting with a prior belief for the position, represented by a Gaussian distribution over positions, and then after an action this Gaussian is shifted (updated) by an amount based on the amount of movement performed during the action. This update is called the state prediction, because it results in a new Gaussian with more variance, that represents a new belief for where the robot is at. Then sensor measurements are taken and this will possibly shift the new Gaussian again but it will definitely decrease its variance. It will have a higher peak and will be thinner.

The problem with the Kalman filter is that the motion and measurements are assumed to be governed by linear models. Extended Kalman Filter makes it possible to utilize the Kalman algorithm also for non-linear movement and measurement models.

2.2 Particle Filters

Particle filter algorithms start with randomly sampled particles in the state space. So for example in localization, each particle will represent a position and orientation in the world. These particles have a scalar weight. As sensor measurements are made, the weights of the particles are updated so that particles which better match up to the measurements have a larger weight. The particles are then resampled, where particles with larger weights are more likely to be sampled, or in other words picked multiple times. After a number of iterations, thanks to the re-sampling, the particles will be clustered around the real position of the robot. Particle filters like AMCL are simple to implement, but they can do global localization and they do not need to assume Gaussian distributions for the state space.

3 Model Configuration

The most important part of the robots are the parameters for the AMCL package and the move_base package. These parameters make a huge difference to the performance of the robot. Other attributes of the robot like the size of the robot, the inertia values of the robot, the friction ODE parameters or the hokuyo laser scanner placement all need different parameter configurations to enable the robot to work. The best placement for the laser scanner is on the front tip of the robot so that the minimum laser distance doesn't need to be adjusted. If the camera/laser scanner is far from the edge of the robot, then the robot will think part of it is inside an obstacle which will cause issues and would need minimum laser distance adjustments.

3.1 Udacity_Bot Configuration

The first robot is the udacity_bot which has a box chassis of sizes 0.4, 0.2 and 0.1 meters and a weight of 15. It has two wheels which have a cylinder model of diameter 0.2 and 0.05 length, with a weight of 5. There are two castors on the bottom of this robot for stabilization. This robot uses a hokuyo laser scanner to determine the distance to obstacles on the map and a camera to give the user view of what is in front in RViz. These are included as Gazebo plugins. Since the robot has two wheels that it cannot turn, so it needs to use a differential drive to be able to turn and move. The differential drive allows the robot to turn by running one wheel forward and the other backwards. This differential drive is implemented in a Gazebo plugin.

3.2 My_Bot Configuration

The second robot is my_bot which has a box chassis of size 0.22, 0.2 and 0.1 with a weight of 10. It doesn't have any castors but instead has four wheels of diameter 0.2 and 0.05 length with weights of 5. Just like udacity_bot it has a hokuyo laser scanner and a camera, but unlike udacity_bot since it has four wheels the steering is more complicated. It uses a skid steer drive where the opposite sets of wheels do go backwards and forwards to turn the robot, but this movement involves the wheels skidding so friction is more important. To use skid steering a Gazebo skid_steer plugin is used.

3.3 Parameter choice

The choice of values for the AMCL and Navigation stack parameters can make a large difference in the quality of the robots movement towards the goal, or if it is able to find it at all. It also depends on the hardware used for running the simulation, because some parameters make the computations more efficient. For this project a 2.2GHz MacbookPro with 16GB RAM was used.

Most of the parameter values were chosen through trial and error and through research of either the default values on the ROS documentation pages or through consulting either the article by Zheng [1] or the MobileRobots ROS config [2] for their pioneer robot which also uses skid steering

Since the two robots were of more or less the same size and executed on the same hardware, many parameter values described in the following paragraphs were used for both the udacity_bot and my_bot. When some parameters had to be different (like the inertia values or the yaw_tolerances) it will be mentioned in the paragraph.

3.3.1 Footprint

The robot footprint needs to be set to the size of the bounding box of the robot so that the trajectory planner can use this to make the robot avoid crashing into obstacles. The footprint was set to be a rectangle of size 0.41 meters so that the bounding box is slightly larger than the robot.

3.3.2 Transform Tolerance

The transform_tolerance parameters are in common params config file and amcl config. This parameter governs how long published transforms can be used in the planner calculations and amcl calculations before they expire and new transforms are needed. If this parameter is set too low then many warning messages are printed by the amcl and the planner. Also if it is too low the robot will miss updates of the amcl and planner and will behave slowly. If this parameter is set too high then outdated transforms will be used in calculations, which will cause poor results. The value that was used in the project was 0.3 seconds for both transform_tolerance parameters.

3.3.3 Obstacle and Raycast Range

The raycast_range and obstacle_range parameters were set to 5.5 and 3.0 meters respectively. These parameters did not seem to make a large difference to the robot, but setting them to higher values like these, makes the robot recognize obstacles further away and makes it sweep out corridors and recognize that they are empty more quickly since it can do it from further away. This seems to help the robot to move faster and localize itself faster (since it can see obstacles from further) which is useful for testing.

3.3.4 Inflation Radius

The inflation_radius determines how large a region around obstacles the robot will recognize as dangerous. This affects how the trajectory planner plans the paths, but in general this parameter wasn't as important in the experiment. It was set to 0.55 meters to make the inflation size large enough to inflate the walls, but at the same time small enough so that the robot isn't too limited in its movement.

3.3.5 Max Particles

The parameter max_particles affects the localization accuracy heavily, since having more particles makes the robot localize itself more accurately which helps it navigate to the target quicker as well as plan global trajectories better. However, when this parameter is set too high, it will cause warnings to be printed because the updates are not fast enough. In the project max_particles was set to 100.

3.3.6 Odom alphas

The odom_alpha parameters are important because the default values of 0.2 are good defaults for the "diff" odom type but not for the bug-fixed "diff-corrected" odom type. To get good values for the odom_alpha parameters, the ones recommended in the documentation [3] were used.

3.3.7 Update and Publish Frequency

These are frequencies for updating the local costmap and global costmap. The update frequency affects how how often the map is updated, which involves updating where obstacles are located. The publish frequency determines how often the visual information about the map is updated. This visual information is used by for example RViz. If these parameters are set to high, it can be a big load on the hardware and cause warning about missed updates. Therefore, in the project both values were set to the recommended default value of 5.

3.3.8 Local and Global Costmap Width and Height

With the rolling_window param set to true as in the configuration of these robots, the local costmap stores information about obstacles close to the robot, which it uses when planning the local trajectory. If the local or global costmap is very large it will affect how quickly the updates can run, so very large values can caused missed updates. Secondly, a smaller local costmap will cause the global trajectory endpoint to be closer to the robot, so having a small local costmap may cause the robot to follow a longer global trajectory path but it will not struggle to connect the local trajectory to the global trajectory. For the udacity_bot and the my_bot a good value turned out to be 5.0 for both height and width. For the global costmap good values were 30.0 for both height and width, which helped avoid missed updates.

3.3.9 Inertia

In the robot definition files which are turned into URDF by xacro, the robot chassis has inertia parameters. These parameters make a big difference to the speed with which the robot is able to move. They should be computed using the formula for a solid cuboid found here [4]. So for example for a robot with mass m, width w, height h and depth d the values should be:

$$I = \begin{pmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{pmatrix} \tag{1}$$

However, for the robots in this project the values computed based on this model did not work well. According to the model for example, the udacity_bot should have 0.0625, 0.2125 and 0.25 inertia values. Most likely due to some other incorrect configuration in the urdf file or the base planner. Values that worked well for the udacity_bot are 0.3 for all diagonal values and for my_bot 0.5 for all diagonal values. With these inertia values the robots move quickly to their goal and don't get stuck in going back to the start of the global trajectory.

3.3.10 Max Velocity and Acceleration Limits

If the dwa parameter is set to false in the base planner config, then the robot needs to have a max and a min velocity for both the x and the theta parameters. This is because the local trajectory planning algorithm discretizes the range of min_velocity to max_velocity into a number of vx_samples or vtheta_samples dx, dtheta values. Then for each dx, dtheta the algorithm will simulate movement of the robot forward in time by sim_time amount using the acceleration limits of the robot. It will then rate all of the simulated paths and pick the best one. So it is important for the robot to have realistic min_velocity and max_velocity values. They don't need to be the real min or max they just need to be reachable. For this project both robots use a 1.2 max_vel_x and for the yaw angle max speed 2.0 is used for both robots.

The acceleration limits should in theory be computed from the real robot. In the case these robots do not exist, but they can still be sent the max velocity using the topic cmd_vel and then in rqt_plot we can check the accelerations as in the below image:

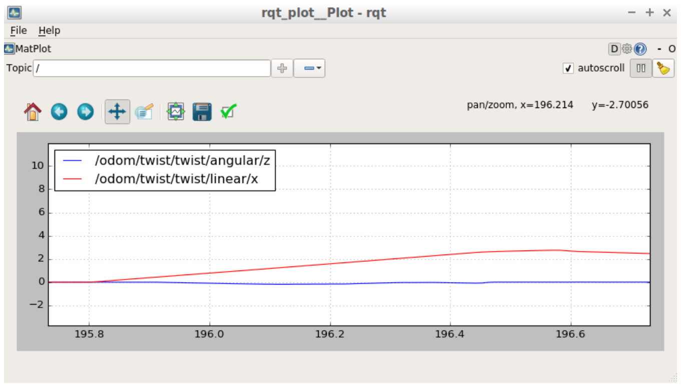


Fig. 1. Rqt plot of x velocity vs time to calculate acceleration limit

3.3.11 VX and VTheta samples

As noted in the section on max velocity and acceleration limits, the vx and vtheta samples determine how fine the sampling is from the min to max velocity ranges. For the robots in this project, a common problem was turning too much in place after reaching the final x and y position. This problem can be alleviated with a more lenient yaw tolerance, but increasing the vtheta_samples also gives the robot more choices to check if they match with the goal. So for the robots in this project vtheta_samples was picked as 50.

3.3.12 Sim Time

As mentioned in the max velocity section, to compute the local trajectory, which is meant to get the robot close to the global trajectory, the robot simulates movement ahead in time along all the sampled dx and dtheta values. This simulation is controlled by the sim_time parameter. These paths are always simple curves without multiple bends. If sim time is chosen too large then the robot will take large curving paths which may not allow it to pass obstacles. If it is chosen too low then the robot will move back and forth too much. In this project the default sim_time of 1.0 seconds was used. It has to be noted that sim_time and min x velocity and min in place theta determine how large tolerances should be. The min times the sim period should be less than the tolerance or else the robot will not be able to reach the goal.

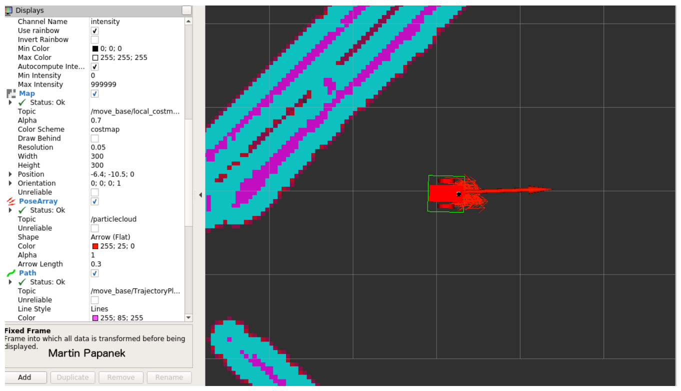
3.3.13 XY and Yaw tolerance

The xy tolerance is used to determine whether the robot is close enough to the goal in terms of its position. Equally, the yaw tolerance determines whether the robot's orientation is close enough to the goal. For the udacity_bot, which is quite precise due to its differential drive, the xy tolerance was set to 0.02 and yaw tolerance to 0.05. The my_bot's rotations and movement are less precise due to its skid steer drive engine, which causes small movements also in the xy direction when the robot rotates. So the xy was set to 0.06 and the yaw tolerance to 0.25.

4 RESULTS

The following image shows the result for the udacity_bot when it reaches the goal. In general this robot performs quite well and manages to find the correct rotation most of the time very quickly when it reaches the destination. The particle filters converge very quickly because we only use 100 particles. The robot reaches the goal on the hardware mentioned in the Parameter Choice section, in about 10 minutes on average and the path to the goal is smooth although the robot sometimes follows a longer path rather than the short one due to the small local costmap size.

The trajectory it follows is smooth, but this robot can have some difficulty initially getting onto the trajectory, so for a minute on average, it may drive in small circles until it finds the path.



Channel Name	intensity
Use rainbow	✓
Invert Rainbow	
Min Color	■ 0; 0; 0
Max Color	□ 255; 255; 255
Autocompute Inte...	✓
Min Intensity	0
Max Intensity	999999
Map	✓
Status: Ok	
Topic	/move_base/local_costm...
Alpha	0.7
Color Scheme	costmap
Draw Behind	
Resolution	0.05
Width	300
Height	300
Position	-6.4; -10.5; 0
Orientation	0; 0; 0; 1
Unreliable	
PoseArray	✓
Status: Ok	
Topic	/particlecloud
Unreliable	
Shape	Arrow (Flat)
Color	🟥 255; 25; 0
Alpha	1
Arrow Length	0.3
Path	✓
Status: Ok	
Topic	/move_base/TrajectoryPl...
Unreliable	
Line Style	Lines
Color	🟪 255; 85; 255

Fixed Frame
Frame into which all data is transformed before being displayed.

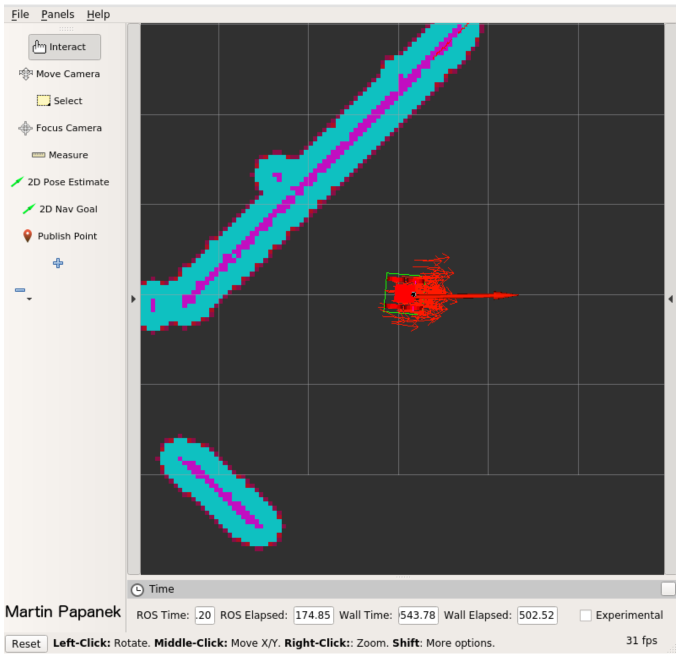
Martin Papanek

Fig. 2. Udacity bot

The following image shows the result of the my_bot when it reaches the end goal. This performs in a less stable way than the udacity_bot because it does not use castors to stabilize it, so very rarely it can flip over. Otherwise it reaches the goal even more quickly than the udacity_bot (in about 5 minutes on average) and follows a smooth path as well with quick convergence of particles.

However, as can be seen from the image, the particles do not in general converge as closely to the real position of the robot and the orientation of the robot is a bit off the real goal location. This is due to the larger tolerances.

Additionally, while the path that the robot follows is smooth, often right after it starts from the initial position it will have a difficult time to get to the global trajectory and this is also where it most often flips over, sometimes by backing into the wall slowly until it flips over.



Martin Papanek

ROS Time: .20 ROS Elapsed: 174.85 Wall Time: 543.78 Wall Elapsed: 502.52

Left-Click: Rotate. Middle-Click: Move X/Y. Right-Click:: Zoom. Shift: More options.

31 fps

Fig. 3. My bot

5 DISCUSSION

After a trial and error approach to parameter tuning, both robots localize quickly and manage to arrive at the goal in several minutes following the global trajectory without crashing directly into the walls or turning too much. However, due to the trial and error nature of the tuning and the fact that information about move_base and amcl parameters online is very sparse and often contradictory, the suspicion is that both robots are most likely misconfigured. The reason for this are most likely the inertia parameters and their surprising influence on the robot.

Inertia parameters make too much of a difference and they have not been set correctly, because the ones from the solid cuboid inertia model don't work (see section Inertia in Parameter Choice for equation 1.). This is very strange since that is the correct inertia model. The robots also react oddly to cmd_vel command with x speed command sometimes causing rotations and angular z commands make the resulting speeds fluctuate too much and often not reach the desired value. This most likely stems from some misconfiguration of the robot, but I was unable to find it, so all that seemed to be possible was to select values that made the robots navigate well despite what seems to be a more underlying issue. Given that even the inertia parameters provided by udacity seem to be different from what they should be, one would suspect that there is something wrong.

Otherwise, changing the local costmap size pretty much solves most of the robot problems so just setting that value quite low will help the robot follow the global trajectory well. This is also unusual because for example the control_-frequency parameter should be enough to solve the issues with the control update misses, but the parameter doesn't seem to work.

The max velocity does not seem to actually make the robots move much faster reliably. The inertia is a lot more reliable at helping the robots move faster. One would suppose it is because the max speed is still sampled and the robot rarely picks the max value. A larger inertia value makes the robot move faster since it is kept going by the inertia.

The udacity_bot performed better than the skid steering my_bot mainly because the skid steering rotations seems to not work correctly and required a much larger yaw_tolerance. One would suspect that this was due to either the inertia issues or issues with the ODE parameters in the collision element's surface subelement. These ODE parameters like mu, mu2, kp and kd should be set based on the material of the robot and should govern how the wheels slide on the surface, which is important for skid steer driving.

The robot at the moment cannot be 'kidnapped' and placed at a different location because once particles have converged to the old location there won't be any particles at or near the new location so the robot won't be able to localize itself there. If the robot could start again with particles distributed evenly throughout the map then it could start again after getting kidnapped. To do this the robot would have to be able to detect that it was kidnapped which it could do by checking how low the weights of the particles are. If all particles are bad, it is quite likely that the robot has been moved so it could remove all current particles and start with new ones.

In general the MCL and AMCL algorithms are very good at performing localization in a known environment and I would use them in any robot where I want to implement localization quickly and where there is knowledge of the position of at least some landmarks in the world which could be used to guide the particle filtering. One shouldn't use MCL if the robot may be kidnapped, not without adjustments like the ones described in the previous paragraph. Also the MCL algorithm would be a poor choice if no landmarks/obstacles are known and therefore there is no way to guide the particle filters. Therefore, an industry example would be a robot navigating a known warehouse where the obstacles are known. An example where MCL would be a poor choice is a robot exploring an unknown cave for instance.

6 Conclusion / Future work

The udacity_bot and my_bot robots localize themselves rapidly and can both navigate to the goal fairly reliably so the robots work as intended. However, the parameter tuning has opened many questions and leaves the possibility open that the robots may be slightly misconfigured. This is due to the odd behaviour when commands are sent through cmd_vel or the odd values for inertia that are necessary for good performance as discussed in the Discussion section.

As future work, the URDF configuration of the robots should be tested without additional localization and other complicating factors to establish correct configuration for the robots before any other sensors like hokuyo are attached and before amcl parameters, costmaps and transform_tolerances are added to the mix. This needs to be done in order to debug the robot configuration in the simplest case possible.

Furthermore, at the moment the robots were tested on a MacbookPro inside VMWare Fusion, so performance was often an issue and many parameters had to be tuned down to make update misses disappear from the amcl package logs. For performance reasons parameters like max_particles, local and global costmap sizes and transform_tolerances were lowered or increased in the case of tolerances to make the robots not miss updates and therefore function well. However, lowering these parameters makes the robots less accurate, so there is a trade-off that had to be taken due to the hardware utilized for this project.

Likewise, both robots only use one hokuyo laser sensor and do not utilize the camera for navigation at all. As future work, more sensors like RGB-D cameras or IMU units could be added and with sensor fusion the accuracy of localization would be improved.

The skid steer robot base could also be improved because at the moment it is fairly unstable. The addition of castors could help stabilize the robot, and a different weight distribution of the base could help prevent the my_bot from flipping which at the moment occasionally does occur.

References

[1] K. Zheng, "Ros navigation tuning guide," 2016.
[2] "https://github.com/mobilerobots/amr-ros-config."
[3] "https://answers.ros.org/question/227811/tuning-amcls-diff-corrected-and-omni-corrected-odom-models."
[4] "https://en.wikipedia.org/wiki/list_of_moments_of_inertia."