Robot Localization using Adaptive Monte Carlo Algorithm

Smruti Panigrahi, Ph.D.
Udacity Robotics Nandegree

***Abstract*—In this paper, a Gazebo/RViz simulation of a mobile robot navigation and localization is presented. The robot model is created using Gazebo and is equipped with a camera and a laser range finder. The task of the robot is to localize itself in the provided simulated map and perform path planning in order to navigate to the goal position. A comparison between the two well-known localization algorithms, namely, Extended Kalman Filter (EKF) and Adaptive Monte Carlo Localization (AMCL) techniques is presented. The simulation is conducted in the ROS (Robot Operating System) environment using particle filter based AMCL algorithm package for localization of the robot relative to the known map and ROS Navigation Stack is used for the path planning and navigation of the robot in the map. Two different mobile robot models are created using Gazebo to compare the performance between the two robots in the same environment using the same AMCL and Navigation Stack ROS package. In addition, the effect of tuning different parameters that impact the accuracy of the robot localization and navigation is discussed.**

Keywords—Robot Simulation, Localization, Kalman Filters, Particle Filters, Monte Carlo Localization, ROS

I. Introduction

Accurate localization is a challenging problem in autonomous vehicles and robotics applications. A lot of care is needed when designing a localization algorithm that performs robustly. Various parameters can affect how well the robot can localize itself in an environment. Robot localization mainly consists of its position coordinate and orientation angle and how well it correlates to the map. For wheeled robots, localization is achieved by analyzing sensor information such as Camera and LiDAR as well as wheel encoders. Various factors can affect the localization of a robot, such as sensor noise and environment noise. In order to overcome these challenges, robust filtering techniques and probabilistic models must be used. For the localization purpose, we investigate two state of the art algorithms based on Kalman filters and particle filters.

As a roboticist or a robotics software engineer, building your own robots is a very valuable skill and offers a lot of experience in solving problems in the domain. Often, there are limitations around building your own robot for a specific task, especially related to available resources or costs.

That's where simulation environments are quite beneficial. Not only there is freedom over what kind of robot one can build, but also get to experiment and test different scenarios with relative ease and at a faster pace. For example, one can have a simulated drone to take in camera data for obstacle avoidance in a simulated city, without worrying about it crashing into a building!

II. Background

The realworld application of the robotic techniques rely heavily on robust probabilistic algorithms to handle sensor noise and uncertainty in the mesuerements. A comparision between the most popular state estimation algorithms, namely, Kalman Filters and the Particle filters, is discussed below.

A. Kalman Filters

The Kalman filter was invented by Swerling (1958) and Kalman (1960) as a technique for filtering and prediction of linear Gaussian systems [1, 2]. Kalman filter algorithm is composed of two steps, namely prediction state and update step, as detailed in Figure 1. Kalman filters are essential in control systems due to their effectiveness in processing noisy data as well as low computational load.

Kalman filters work by iterating between measurement updates and state prediction. An initial guess is used to form a prediction. The prediction is then updated using a new measurement. The weight given to that measurement is relative to its expected uncertainty. Those with little expectancy are given a heavier weight, and vice versa. Using this data, a new prediction is formed. And the measurement update and state prediction cycle continues in a real-time environment.

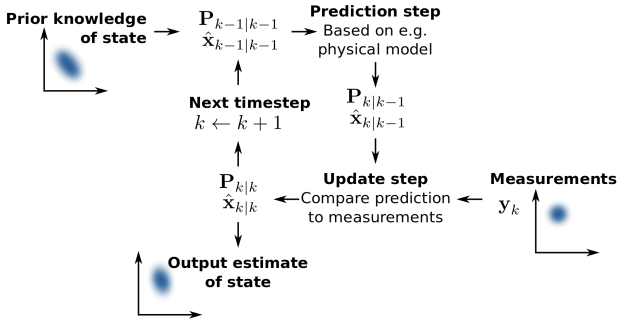

Fig. 1: Kalman Filter concept

The algorithm is detailed in Fig. 2 below.

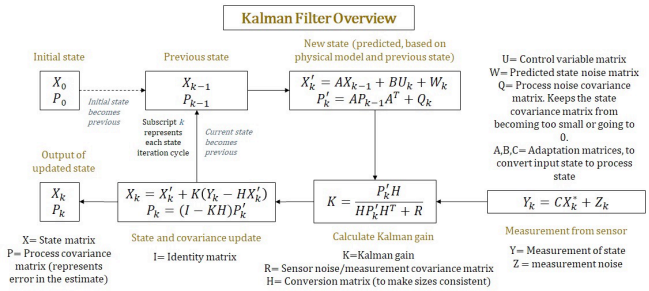

Fig. 2: Kalman Filter Algorithm

It is important to note that both the system and observation models are assumed to be linear which is not a realistic assumption. Additionally it is also assumed that the state belief is linear. As most systems in realworld are nonlinear, in order to model such nonlinearity, an Extended Kalman Filters (EKF) algorithm is used. Since the measurement and/or state transition functions are nonlinear, their distribution is non-Gaussian.

The EKF algorithm is similar to the traditional Kalman Filter algorithm, with the exception of needing to linearize a nonlinear motion or measurement function with multiple dimensions using multi-dimensional Taylor series to be able

to update the variance. This can be done by taking a local linear approximation and using this approximation to update the covariance of the estimate. The linear approximation is made using the first terms of the Taylor Series, which includes the first derivative of the function. In case of multi-dimensional case, taking the first derivative isn't as easy as there are multiple state variables and multiple dimensions. Hence a Jacobian is employed, which is a matrix of partial derivatives, containing the partial derivative of each dimension with respect to each state variable as shown in Fig. 3.

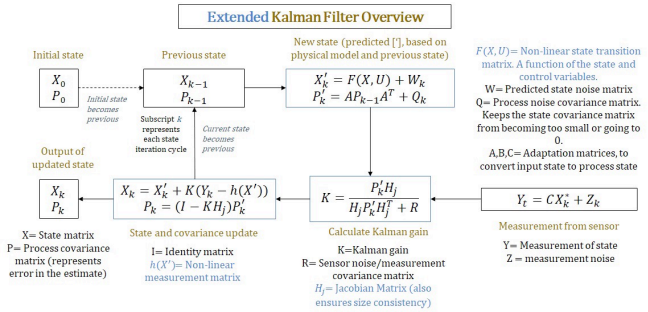

Fig. 3: Extended Kalman Filter (EKF) Algorithm

B. *Particle Filters*

Particle filters [2], are the alternative non-parametric implementation of the Bayes rule. Two steps are performed in a particle filtering process, particle initialization and particle sampling. In the particle initialization process the algorithm generates hypotheses on the pose. The hypotheses are called particles and are distributed over the robot environment. Each of the particles are hypothesis of the robot's pose represented by a 3D Pose and a weight. The weight represents the difference between the robot's actual pose and that particle's hypothetical pose.

At the start, particles are randomly and uniformly spread around the map. After each cycle, the weights of each particle are calculated and those of lower weight are discarded, while those of larger weight move onto the next resampling process. The cycle continues until all the particles that are faster away from the robot are eliminated and only the ones centered around the robot remain, eventually converging on the robot's pose.

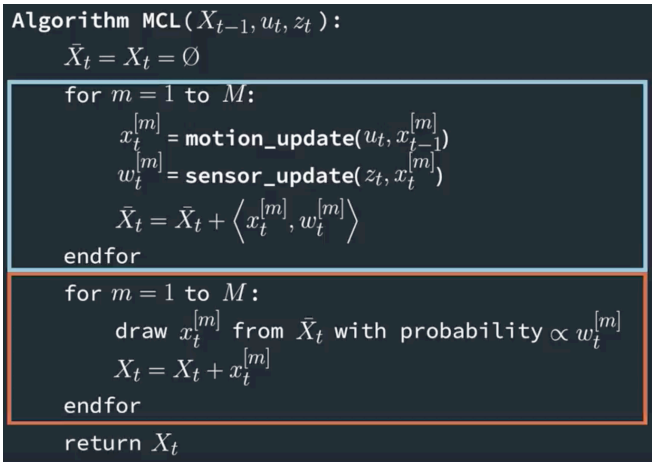

Fig. 4: Monte Carlo Localization Algorithm concept

Particle filter based localization is known as the Monte Carlo Localization (MCL) and is shown in Fig. 4. The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robot's position and orientation based on sensory information. This process is known as a recursive Bayes filter. Using a Bayes filtering approach, the state (robot's pose, including its position and orientation) of a dynamical system (the mobile robot and its environment) from sensor measurements (perception data e.g. laser scanners and odometry data e.g. rotary encoders) can be estimated. The goal of Bayes filtering is to estimate a probability density over the state space conditioned on the measurements.

C. *MCL vs EKF*

An example comparing the EKF and MCL is shown below in Fig. 5 and Fig. 6. The example considers a one dimentional problem where the robot is travelling in a hallway along the x-axis as shown below. The mobile robot in this hallway can move left and right along the x-axis. The robot is capable of mesuring odometry information and sensing the presence of the doors along the hallway.

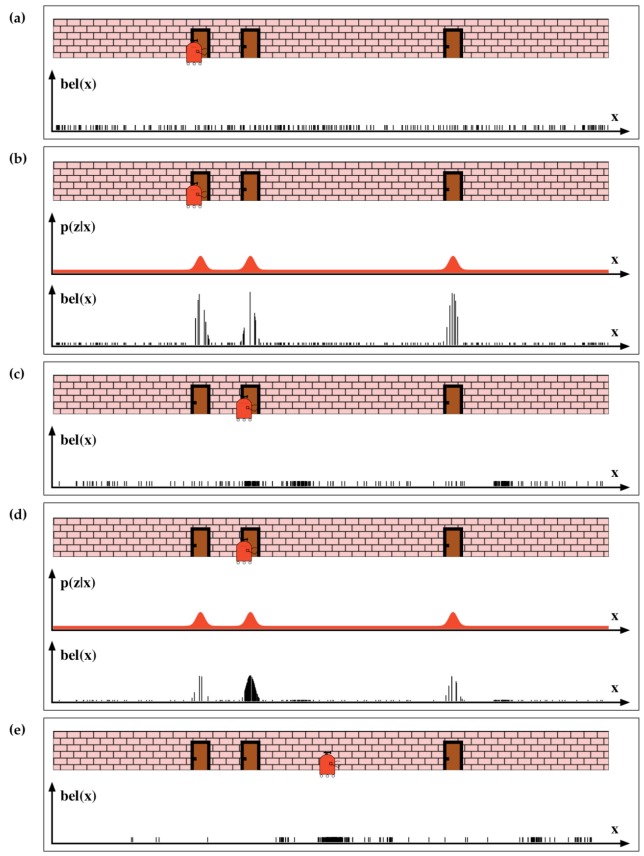

Fig. 5: MCL example from Probabilistic Robotics [2]

At time:

- t=1, Particles are drawn randomly and uniformly over the entire pose space.
- t=2, The robot senses a door and MCL assigns weight to each particle. The number of particles remains constant where as the particle's importance weights are adjusted. Measurement is updated and an importance weight is assigned to each particle.
- t=3, The robot moves down the hallway and the particle set is shifted. The set has uniform importance weight and high numbers of particles near the three likely

places. Motion is updated and a new particle set with uniform importance weights and high number of particles around the three most likely places is obtained in resampling.

- t=4, Moving on, the robot senses another door and the new measurement assigns non-uniform importance weight to the particle set.
- t=5, The robot keeps moving down the hallway causing new particle set to generate. Motion is updated and a new resampling step is about to start.

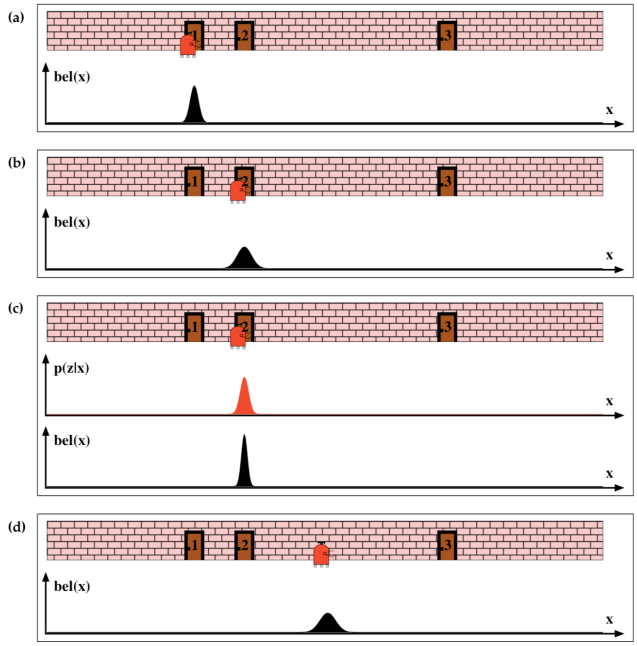

Fig. 6: EKF example from Probabilistic Robotics [2]

At time:

- t=1, Initial belief represented by a Gaussian distribution around the first door.
- t=2, Motion is updated and the new belief is represented by a shifted Gaussian of increased weight.
- t=3, Measurement is updated and the robot is more certain of its location. The new posterior is represented by a Gaussian with a small variance.
- t=4, Motion is updated and the uncertainty increases.

	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. 7: Comparison between EKF and MCL.

In order to select the ideal filter for our simulation a comparison is done between Particle and Kalman Filters as shown in Fig. 7 below.

Due to non-gausian posterior and mesurement noise, the Particle filters are best fit for the localization problem. The particle filters are capable of modeling highly nonlinear models, localize using raw laser readings without the need of landmarks, and global localization of the robot within its map. Such factors are highly critical for accurate localization.

In position tracking problems of low uncertainty, the EKF performs extremely well. Advantage of the EKF is its ability to combine and fuse data from multiple sensors. With its low memory and workload requirement, it is ideal for situations where computational power is scarce.

While the EKF is limited to Gaussian distribution, the MCL are not restricted by a Gaussian, and can handle problems of higher dimensions. This makes their range of applications extend into the nonlinear domain, where the majority of real world problems lie. One drawback of MCL is that it is much more computationally expensive than EKF. However, the memory usage and resolution can be tunes in order to meet system capacity. For real-time update an adaptive MCL (AMCL) is used. Hence, for real-time localization of the robot, the particle filter based AMCL algorithm is used in this project.

III. Robot Simulation in ROS

The Robot Operating System (ROS) [3] is used to create ROS package for the simulation of the robot. Various ROS packages are used to accurately localize a mobile robot inside a provided map in Gazebo and RViz simulation environments. Steps used to create the simulation model is discussed below.

A. Building ROS Package

Several aspects of robotics is explored with a focus on ROS, including

1) Building a mobile robot, using Gazebo, for simulated tasks.
2) Creating a ROS package that launches a custom robot model in a Gazebo world.
3) Adding sensors to the robot, and integrating it with Gazebo and RViz.
4) Integrating open-source ROS packages like the Adaptive Monte Carlo Localization (AMCL) package and the Navigation Stack package, that would allow the robot to navigate and localize itself in a particular environment or map.
5) Parameter tuning corresponding to each package to achieve the best possible localization results in order for the robot to navigate to a goal position provided in Rviz. ifferent parameters corresponding to these packages that could help you improve your results.

In this section, a brief instruction is provided to start developing ROS packages from scratch. After creating and initializing the “catkin_ws” workspace, start by navigating to the “src” directory and creating a new empty package called “udacity_bot”.

Next, inside “udacity_bot” folder, create folders, “launch”, “worlds”, “urdf”, “config”, “maps”, “meshes”, “src” that will further define the structure of the package.

B. Robot Model: Gazebo World

In “worlds”, each individual Gazebo world will be created and saved. A world is a collection of models such as the robot, and a specific environment such as the café and several other physical properties specific to this world.

The .world file uses the XML file format to describe all the elements that are being defined with respect to the Gazebo environment. The world that is created, has the elements sdf, world, and include.

C. Robot Model: Launch Files

Launch files in ROS allow us to execute more than one node simultaneously, which helps avoid a potentially tedious task of defining and launching several nodes in separate shells or terminals.

As in case of the .world file, the udacity_world.launch files are also based on XML. The structure for the file above is essentially divided into two parts. First, certain arguments need to be defined using the <arg> element. Each such element will have a name attribute and a default value. Then, the empty_world.launch file is included from the gazebo_ros package. The empty world file includes a set of important definitions that are inherited by the world that we create. Using the world_name argument and the path to the .world file passed as the value to that argument, the world can be launched in Gazebo.

D. Robot Model: Basic Robot URDF Setup

There are several approaches or design methodologies to consider when creating a robot model. For a mobile robot, basic components needed are a robot base, wheels, and sensors.

For this model, a cuboidal base with two caster wheels is created. The caster wheels will help stabilize this model. They aren't always required, but it can help with weight distribution, preventing the robot from tilting along the z-axis at times.

A URDF file is used to create the model with specific dimensions. Then a new robot description launch file is created that will help load the URDF file that defines a parameter robot_description which is used to set a single command to use the xacro package to generate the URDF from the xacro file.

Next, in the launch folder, update udacity_world.launch file in order to load the URDF robot model file in Gazebo. The gazebo ROS package spawns the model from the URDF that robot_description helps generate.

E. Robot Model: Robot Actuation

The udacity_bot requires only two wheels. Each wheel is represented as a link and is connected to the base link (the chassis) with a joint. First the links for each wheel is created using the specifications and is added to the xacro file. Each wheel contains collision, inertial, and visual elements.

Once the links are defined, the corresponding joints between the wheels and the chassis are defined next with wheel as the child link and the robot chassis as the parent link. The joint type is set to "continuous" and is similar to a revolute joint but has no limits on its rotation. It can rotate continuously about an axis. The joint will have it's own axis of rotation, some specific joint dynamics that correspond to the physical properties of the joint like "friction", and certain limits to enforce the maximum "effort" and "velocity" for that joint. The limits are useful constraints in regards to a physical robot and can help create a more robust robot model in simulation as well.

F. Robot Model: Robot Sensors

For this robot, two sensors - a camera and a laser rangefinder is used. The links and the joints for the camera and the LiDAR is added to the xacro file for the robot where the parent link is the robot chassis, and joint child link are the camera and LiDAR. As discussed above, each link should have its own visual, collision and inertial elements. ROS offers support for a lot of different types of sensors. One of them is the Hokuyo rangefinder. For the laser finder link, a Hokuyo.dae mesh file defines the shape of the object which is downloaded from the ROS.

G. Robot Model: Gazebo Plugins

Now the robot is equipped with sensors allowing it to visualize the world around it. But how exactly does the camera sensor takes those images during simulation? How exactly does your robot move in a simulated environment?

The URDF in itself can't help with the robot’s movement inside the simulation. However, Gazebo allows us to create or use plugins that help utilize all available gazebo functionality in order to implement specific use-cases for specific models. We will cover the use of three such plugins, namely

1. A plugin for the camera sensor.
2. A plugin for the hokuyo sensor.
3. A plugin for controlling the wheel joints.

The udacity_bot.gazebo file includes these three plugins. You can find them in the github repo. You will have to add this file to the urdf folder of your package.

The udacity_bot.gazebo file in the URDF folder includes these three plugins. Since we have a two-wheeled mobile robot, we will use a plugin that implements a Differential Drive Controller. The plugin takes in information specific to the robot's model, such as wheel separation, joint names and more, and then calculates and publishes the robot's odometry information to the topics that you are specifying above, like the odom topic. A velocity command is sent to the robot to move it in a specific direction. This controller helps achieve that result.

Gazebo already has several of such plugins available for anyone to work with. We will utilize the preexisting plugins for the camera sensor and the plugins for the hokuyo sensor. Both of these are already included in the .gazebo file linked previously. For each of the two sensors, camera and laser, one needs to define the topics where they will publish information or data. For the camera, it's the image_raw topic, and for the laser, it's the udacity_bot/laser/scan topic.

H. Robot Model: RViz Integration

While Gazebo is a physics simulator, RViz can visualize any type of sensor data being published over a ROS topic like camera images, point clouds, LiDAR data, etc. This data can be a live stream coming directly from the sensor or pre-recorded data stored as a bag file. RViz is the one-stop tool to visualize all the three core aspects of a robot: Perception, Decision Making, and Actuation.

In this section, the robot model is integrated into RViz to visualize data from the camera and laser sensors.

In the launch file (robot_description.launch), two nodes need to be added. One node uses the package joint_state_publisher that publishes joint state messages for the robot, such as the angles for the non-fixed joints. The second node uses the robot_state_publisher package that publishes the robot's state to tf (transform tree). The robot model has several frames corresponding to each link/joint. The robot_state_publisher publishes the 3D poses of all of these links. This offers a very convenient and efficient advantage, especially for more complicated robots (like the PR2).

Once the launch file is executed, both RViz and Gazebo are launched. In the RViz window, on the left side, under Displays: select “odom” for fixed frame, click the “Add” button and add “RobotModel”, add “Camera” and select the Image topic that was defined in the camera gazebo plugin, and add “LaserScan” and select the topic that was defined in the hokuyo gazebo plugin. This makes sure that the robot model along with sensor configurations is properly loaded into RViz.

IV. Robot Localization and Navigation

In order to localize the robot and navigate to the goal position, advanced localization techniques such as AMCL is used as described in this section. The map of the robot environment is shown in Fig. 8 below.

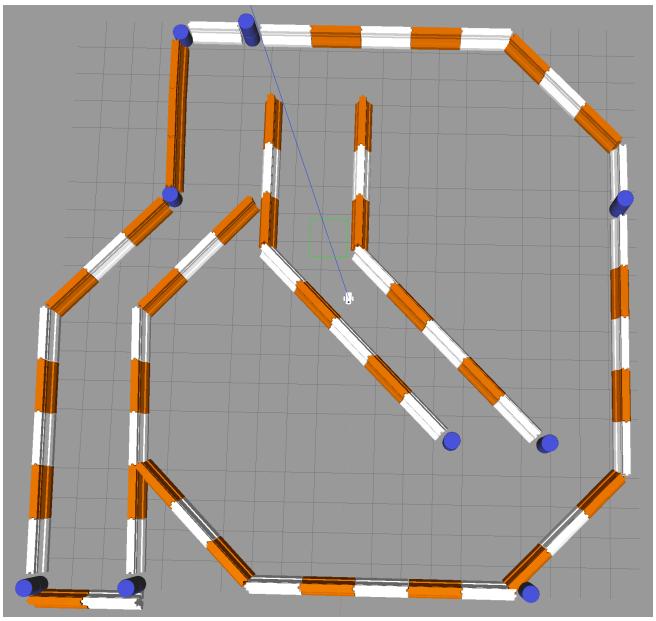

Fig. 8: Map of the enviorment where the robot will perform localization and navigation.

A. Localization: Map

So far, a robot model has been created from scratch, added sensors to it to visualize its surroundings, and developed a package for this robot to launch it in a simulated environment. But, what surroundings is the robot currently sensing? And how would the robot localize if it is not known where where the robot needs to localize?

The robot is launched in a new environment using a map created by Clearpath Robotics. Two files jackal_race.pgm and jackal_race.yaml from this project repo can be copied into the “maps” folder. The map that the robot needs to be in is generated based on jackal_race.world is copied into the “worlds” folder and the udacity_world.launch needs to be modified accordningly in order to launch the required map for the jackal. This can be done by modifying the argument world_name such that it points to jackal_race.world.

B. Localization: AMCL

Adaptive Monte Carlo Localization (AMCL) dynamically adjusts the number of particles over a period of time, as the robot navigates around in a map. This adaptive process offers a significant computational advantage over MCL. The ROS amcl package [4] implements this variant and is integrateed into the robot to localize it inside the provided map. A new amcl.launch with three nodes is created, one of which is for the amcl package.

First, the provided map is loaded using a new node for the map_server package. Previously, the robot_state_publisher helped built out the entire tf tree of the udacity_bot robot based on the URDF file. But it didn't extend that tree by linking in the 'map' frame. The amcl package does that automatically by linking the 'map' and 'odom' frames.

Then a node is added that will launch the amcl package. The package has its own set of parameters that defines its behavior in RViz and how everything relates to the robot and the provided map so that the robot can effectively localize itself. The amcl package relies entirely on the robot’s odometry and the laser scan data.

A remaping is done of the scan topic to the udacity_bot/laser/scan topic on which the hokuyo sensor publishes the laser sensor data. This topic was defined when adding the gazebo plugin for the sensor. Then, parameters are added and values for the different reference frames such as the odom or the map frames are defined. The amcl package will now be able to take in the laser and odom data, and localize the robot!

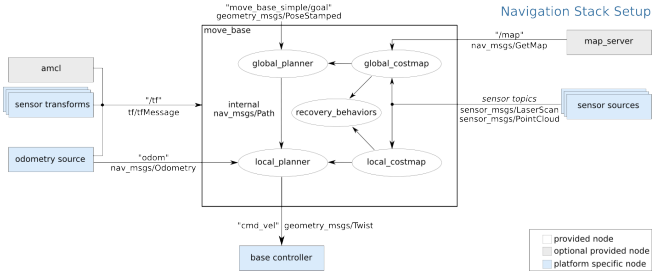

Fig. 9: An overview of the ROS Navigation Stack

However, the robot is still stationary. How will it navigate to any position to collect more information from its surroundings? That’s where the ROS Navigation Stack [5]

comes in! An overview of the navigation stack is shown in Fig. 9.

C. *Navigation Stack*

The move_base package [6] is used in order to define a goal position for the robot in the map, in order for the robot to navigate to that goal position. The move_base package is a very powerful tool. It utilizes a costmap, where each part of the map is divided into which area is occupied, like walls or obstacles, and which area is unoccupied. As the robot moves around, a local costmap, in relation to the global costmap, keeps getting updated allowing the package to define a continuous path for the robot to move along.

What makes this package more remarkable is that it has some built-in corrective behaviors or maneuvers. Based on specific conditions, like detecting a particular obstacle or if the robot is stuck, it will navigate the robot around the obstacle or rotate the robot till it finds a clear path ahead.

The global and local costmaps with a particular color scheme are shown in Fig. 10 below. On the left, the global costmap is displaying all the occupied areas based on the provided map. And on the right, the local costmap of the same region is displayed. The local costmap is only displaying what the laser sensor captures during that time period.

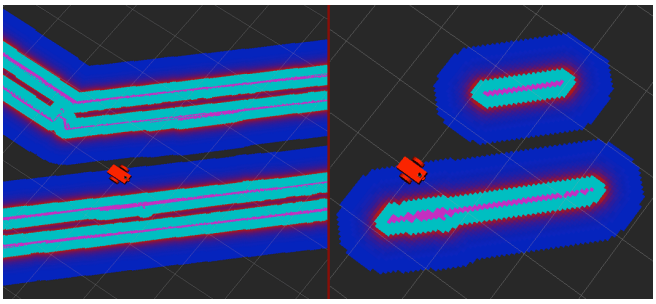

Fig. 10: Global (left) and local (right) costmap shown in RViz

Just like the amcl, or any other package, move_base also has its own set of required parameters that help it perform efficiently. Specific topics need to be remaped to allow them to take in input from odometry or laser data, and also define some configuration files for parameters or definitions pertaining to the costmaps as well as the local planner that creates a path and navigates the robot along that path.

Various parameter configuration files such as local_costmap_params.yaml, global_costmap_params.yaml, amcl.yaml, costmap_common_params.yaml, and base_local_planner_params.yaml hold the parameters that needs to be tuned properly in order for the robot to navigate in the map and localize itself accurately.

Now the map, the amcl localization and navigation nodes can be launched using the following commands.

- $ cd /home/workspace/catkin_ws/
- $ roslaunch udacity_bot udacity_world.launch

In a new terminal,

- $ roslaunch udacity_bot amcl.launch

In Rviz, select “odom” for fixed frame, click the “Add” button, add “RobotModel”, add “Map” and select first topic/map.

The second and third topics in the list will show the global costmap, and the local costmap. Both can be helpful to tune the parameters. Also add “PoseArray” and select topic /particlecloud to display a set of arrows around the robot. Each arrow is essentially a particle defining the pose of the robot that your localization package created. Your goal is to add/tune the parameters that will help localize your robot better and thereby improve the pose array. At this stage the RViz setup can be saved in a configuration file and launch RViz with the same configuration every time. This will make the process more efficient for you!

In order provide a goal for the Robot, in RViz, in the toolbar, select “2D Nav Goal”, click anywhere else on the map and drag from there to define the goal position along with the orientation of the robot at the goal position. The robot should start moving towards the defined goal position!

The amcl.launch file might show a lot of warnings, the map in RViz might be flickering, or the robot might not be moving too well or at all. All of these are expected if the parameters are not tuned properly. In the next section, parameter tuning is discussed to fix such issues.

V. Parameter Tuning

Exploring, adding, and tuning parameters for the amcl and move_base packages are some of the main goals of this project. In this section, we experiment with some parameters that will help the robot localize and navigate

Before we jump into the list, let’s take a step back and focus on what we are aiming to achieve. The move_base package will help navigate the robot to the goal position by creating or calculating a path from the initial position to the goal, and the amcl package will localize the robot. But how do we know how certain the algorithm is of the robot’s pose?

That’s what the PoseArray, in RViz, helps us with. The PoseArray depicts a certain number of particles, represented as arrows, around the robot.

The position and the direction the arrows point in, represent an uncertainty in the robot’s pose. This is a very convenient, albeit a slightly subjective, method to understand how well your algorithm and the tuned parameters are performing. Based on the parameters and what values are selected, as the robot moves forward in the map, the number of arrows should ideally reduce in number. The algorithm rules out some poses. This is especially the case for when the robot is closer to walls - it is more certain of its pose because of the laser data, as opposed to when it is roaming in an open area for too long.

similar results should be expected around the goal position. Depending upon how the robot has to orient itself, it might have slightly poorer results. But overall, the PoseArray should be narrow, centered around the robot, and pointing in the right direction.

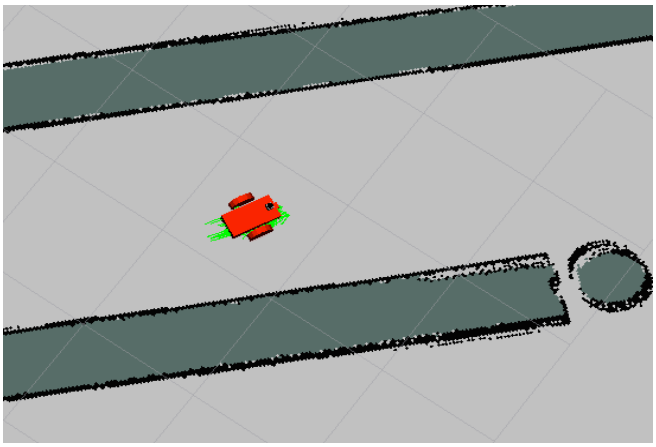

Fig. 11: PoseArray centered arounf the robot.

The aim for this project is to make sure that the localization results are as certain of the robot's pose as the fig. 11. If the spread of the arrows, or particles, is too large then the robot is highly uncertain of its position, and the same goes for which direction the majority of the arrows are pointing in.

A. Transform Timeout

There are three different types of maps that are being created/generated in RViz when amcl.launch is launched.

- The world map - The visualization of the world or environment from Gazebo.
- The global costmap - Created by the navigation stack/package. A costmap, which you will learn more about in future lessons, essentially divides the map into a grid where a cell could represent free space or an obstacle. The global costmap is used to generate a long-term path for the robot, such as the path to the goal position from the robot's starting position.
- The local costmap - The local costmap is used to generate a short-term path for the robot. For example, a path that attempts to align itself and the robot with the global path.

The tf package, helps keep track of multiple coordinate frames, such as the transforms from these maps, along with any transforms corresponding to the robot and its sensors. Both the amcl and move_base packages or nodes require that this information be up-to-date and that it has as little a delay as possible between these transforms.

```
[ WARN] [1526848801.233300330, 1548.725000000]: Costmap2DROS transform timeout.
Current time: 1548.7250, global_pose stamp: 1548.6470, tolerance: 0.0000
[ WARN] [1526848802.579933194, 1549.725000000]: Costmap2DROS transform timeout.
Current time: 1549.7250, global_pose stamp: 1549.6470, tolerance: 0.0000
```

Fig. 12: Transform tolerance warnings

The warning message in the above image in Fig. 12 is indicative of the maximum allowed delay to either be not defined or to be too low for the system to compensate for. This maximum amount of delay or latency allowed between transforms is defined by the transform_tolerance parameter in amcl.launch and costmap_common_params.yaml file. Tuning the value for this parameter is usually dependent on the system performance.

Once the transform_tolerance variable is defined and tuned properly, the map visualization in all the three maps in RViz should not have any issues, and the warning should disappear. Only to be replaced by a new warning.

B. Map Update Loop

The warning shown in Fig. 13 seems to indicate that the map or costmaps are not getting updated fast enough. The update loop is taking longer than the desired frequency rate of 50 Hz or 0.02 seconds. More processing power will help fix the issue, but that's not always feasible. Since it's an issue related to the map or the costmaps, by tunin the configuration parameter for the move_base node should fix the issues. Other parameters for the costmap can be found in [7].

```
[ WARN] [1526850958.405785521, 1526.425000000]: Map update loop missed its desir
ed rate of 50.0000Hz... the loop actually took 0.0420 seconds
[ WARN] [1526850958.443950166, 1526.452000000]: Map update loop missed its desir
ed rate of 50.0000Hz... the loop actually took 0.0270 seconds
```

Fig. 13: Map update warning

A larger and more detailed map will result in a large global costmap, and would use more resources. Apart from tuning the frequency with which the map is getting updated and published, the dimension and resolution of the global and local costmaps can also be modified.

Modifying these parameters can help free up some resources, however, decreasing the resolution of the map by too much can lead to loss of valuable information too. For example, in case of small passages, low resolution might cause the obstacle regions to overlap in the local costmap, and the robot might not be able to find a path through the passage.

If the robot is not following the path and runs into the walls more parameters need to be tuned in the costmap_common_params.yaml config file. These parameters are very important in defining how the costmaps get updated with obstacle information and how the robot might respond to those while navigating.

- obstacle_range - For example, if set to 0.1, that implies that if the obstacle detected by a laser sensor is within 0.1 meters from the base of the robot, that obstacle will be added to the costmap. Tuning this parameter can help with discarding noise, falsely detecting obstacles, and even with computational costs.
- raytrace_range - This parameter is used to clear and update the free space in the costmap as the robot moves.
- inflation_radius - This parameter determines the minimum distance between the robot geometry and the obstacles. Setting a really high value for this parameter, can make the obstacles (the walls of the environment) seem to be "inflated" as can be seen in Fig. 14 below. An appropriate value for this parameter can ensure that the robot smoothly navigates through the map, without bumping into the walls and getting stuck, and can even pass through any narrow pathways.

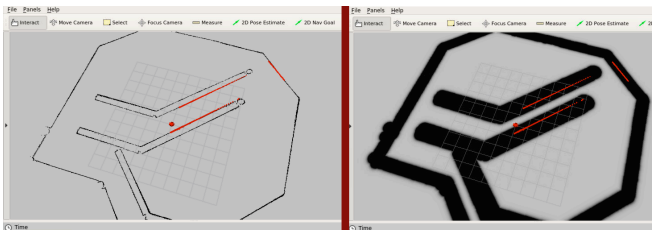

Fig. 14: Costmap with and without the inflation

Next, we need to identify and tune parameters for the amcl node in the amcl.launch file, to achieve better results in localization.

C. *AMCL Parameters*

The AMCL package [8] has a lot of parameters to select from. Different sets of parameters contribute to different aspects of the algorithm. Broadly speaking, they can be categorized into three categories - overall filter, laser, and odometry. The parameters are listed in the amcl.yaml in the config folder.

Overall Filters

- min_particles and max_particles - As amcl dynamically adjusts its particles for every iteration, it expects a range of the number of particles as an input. Often, this range is tuned based on your system specifications. A larger range, with a high maximum might be too computationally extensive for a low-end system.
- initial_pose – This is set to the position to [0, 0].
- update_min* - amcl relies on incoming laser scans. Upon receiving a scan, it checks the values for update_min_a and update_min_d and compares to how far the robot has moved. Based on this comparison it decides whether or not to perform a filter update or to discard the scan data. Discarding data could result in poorer localization results, and too many frequent filter updates for a fast moving robot could also cause computational problems.

Laser

There are two different types of models to consider under this - the likelihood_field and the beam. Each of these models defines how the laser rangefinder sensor estimates the obstacles in relation to the robot.

The likelihood_field model is usually more computationally efficient and reliable for an environment such as the one you are working with. So you can focus on parameters for that particular model such as the -

- laser_*_range
- laser_max_beams
- laser_z_hit and laser_z_rand

Tuning of these parameters will have to be experimental. While tuning them, observe the laser scan information in RViz and try to make sure that the laser scan matches or is aligned with the actual map, and how it gets updated as the robot moves. The better the estimation of where the obstacles are, the better the localization results.

Odometry

odom_model_type – Since the robot is a differential drive mobile robot, it's best to use the diff-corrected type. There are additional parameters that are specific to this type - the odom_alphas (1 through 4). These parameters define how much noise is expected from the robot's movements/motions as it navigates inside the map.

Identifying and tuning all these parameters will help localize the robot robustly.

D. *Launching*

Using a C++ node, navigation_goal.cpp, the navigation goal is provided to the robot. This file is comipled automatically by including the add_executable command in the CMakeLists.txt file.

When launching the project, after we have launched udacity_world.launch and amcl.launch, in a new terminal run the following -

$ rosrun udacity_bot navigation_goal

The above will run the node, and the robot will start navigating towards the goal position defined in the navigation file.

VI. Results and Discussions

Two physical robot models were created using Gazebo, udacity_bot (Fig. 15) and my_bot (Fig. 16). They are very similar in shape and size except the my_bot has an extra mount for the LiDAR sensor in the front of the robot.

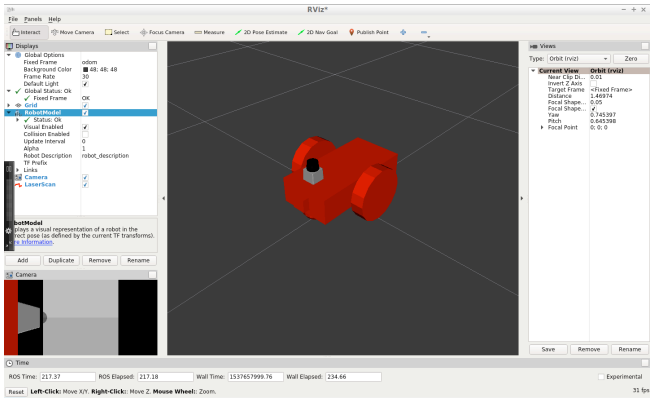

Fig. 15: Udacity_Bot robot model

Both the LiDAR and the Camera sensors in my_bot are elevated to 0.2m higher than the udacity_bot sensor position. Also, the camera in my_bot is moved back by 0.025m in the x-direction. This provides different viewing angle and perspective for my_bot.

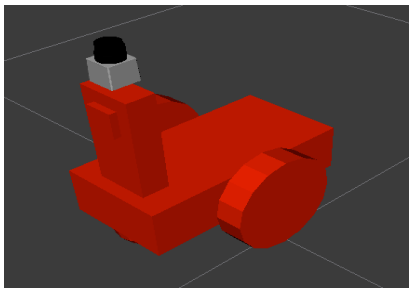

Fig. 16: My_Bot robot model

The move_base package incorporates map, odometry and sensor data to evaluate the best path, and provides the robot with the associated velocity commands. It provides the interface between the path-planning nodes, base_local_planner and base_global_planner and the costmap nodes, local_costmap and global_costmap. The amcl node performs probabilistic localization by implementing the Adaptive Monte Carlo algorithm, thus keeping track of the robot's pose with respect to its environment.

The package costmap_2d creates an occupancy grid map whichi s a 2D representation of the space around the robot. The generated map is composed of superimposed layers of static, obstacle, and inflation layer. In the obstacle layer, each detected cell in the grid is categorized as an unknown, occupied, or free space. These values are then passed to the planners, base_local_planner and base_global_planner which calculates the cost of each cell, and then inflates the value of the cells according to their respective parameters. Costmaps are covered more in the section 5.

For map update, the tollerance parameters, xy_goal_tolerance and yaw_goal_tolerance are both set to 0.2 which helps in optimizing the goal position and orientation. The transform_tolerance and inflation_radius are set to 0.2 to match the system capacity and the frequency of the map update and to allow enough robot and the wall in order for the robot to navigate without hitting the walls.

The update_frequency and publish_frequency is set to 10 Hz for both the local and global costmap. The resolution for both the local and the global cost map is fixed at 0.02. The height and the width of the local costmap is chosen to be 5 meters and that of the global costmap is 30 meters.

The base local planner algorithms begin by sampling possible velocities in x-y and yaw. Each velocity sample is then projected forward in time for anywhere between 0.5 to 10 or more seconds, based on the processor power available. Each projection‘s trajectory cost is then calculated based upon the trajectory‘s proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Trajectories that result in a collision with an obstacle are discarded, and the trajectory with the lowest cost is selected. The trajectory‘s associated velocity is then sent to the robot. The formula used to calculate trajectory costs, C, is [9]:

$$C = pdist\_scale * dP + gdist\_scale * dG + occdist\_scale * C0$$

Where, dP is the distance to path from the endpoint of the trajectory in map cells or meters depending on the meter_scoring parameter, dG is the distance to local goal from the endpoint of the trajectory in map cells or meters depending on the meter_scoring parameter, C0 is the maximum obstacle cost along the trajectory in obstacle cost (0-254). The parameters for the local planner are located in base_local_planner_params.yaml.

The holonomic_robot parameter in the base trajectory planner defines whether the robot can travel omnidirectionally. Being a differential drive robot, ours is not holonomic, hence, the value false is used. The pdist_scale parameter defines the weight for how much the planner should stay close to the global path. A high value allows for local paths that deviate less from the global path. The parameter value used is 0.6. The gdist_scale parameter defines the weight for how much the robot should attempt to reach the local goal. A high value allows flexibility in the local path. The parameter value used for the robot is 0.8. The occdist_scale parameter defines the weight for how much the planner should attempt to avoid obstacles. The value can be 0.01.

The static_map parameter specifies if a static map is being used. The rolling_window parameter specifies whether the costmap will remain centered around the robot as it moves. If the static_map parameter is set to true, this parameter must be set to false.

Table 1: The parameters used for the LiDAR in order to tune the AMCL package

Laser Scanner Parameters	**Parameter Value**
laser_model_type	likelihood_field_prob
laser_likelihood_max_dist	2.0
laser_min_range	0.5
laser_max_range	10
laser_max_beams	30
laser_z_hit	0.99
laser_z_short	0.1
laser_z_max	0.05
laser_z_rand	0.01
laser_sigma_hit	0.2
laser_lambda_short	0.1

Table 2: The parameters used for the Odometry in order to tune the AMCL package

Odometry Parameters	**Parameter Value**
odom_frame_id	odom
odom_model_type	diff-corrected
base_frame_id	robot_footprint
global_frame_id	map
tf_broadcast	false
odom_alpha1	0.01
odom_alpha2	0.01
odom_alpha3	0.03
odom_alpha4	0.03

Table 3: The parameters used for the filtering in order to tune the AMCL package

Filter Parameters	**Parameter Value**
use_map_topic	true
min_particles	25
max_particles	250
kld_err	0.01
kld_z	0.99
transform_tolerance	0.2
recovery_alpha_slow	0.001
recovery_alpha_fast	0.1
initial_pose_x	0.0
initial_pose_y	0.0
initial_pose_a	0.0

The static_map is false for local_costmap_params.yaml whereas it is true for the global_costmap_params.yaml. The rolloing_window is then set as true and false for the local and global costmaps rspectively.

For AMCl, the parameter values used are listed in the Tables 1, 2 and 3. The global_frame_id chosen as the map, defines the transform associated with publishing costmap in RViz,

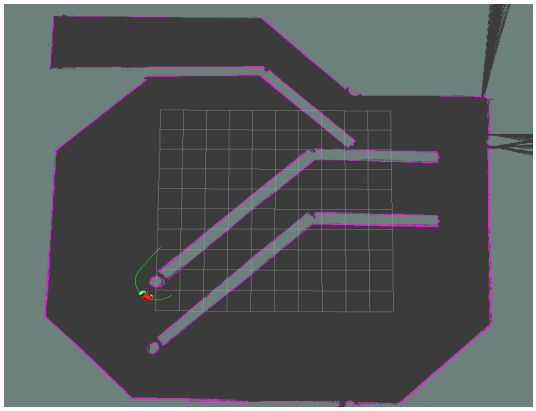

Fig. 17: My Bot localizing and naviagting towards the goal position while manuvering around the corner.

In both the udacity_bot and my_bot, all of the parameters stayed the same in order for the robot to localize well and navigate to the goal position except, one parameter, ostacle_range. The obstacle_range for udacity_bot is 3.0 whereas for my_bot it is 4.0. For both robots raytrace_range in the costmap parameter is set to 7.5.

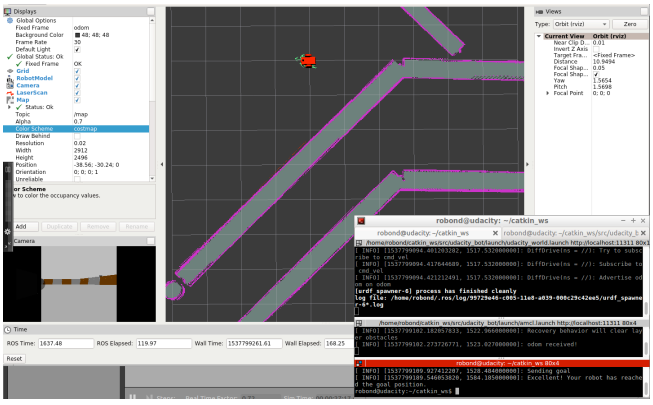

Fig. 18: Udacity_Bot reached goal

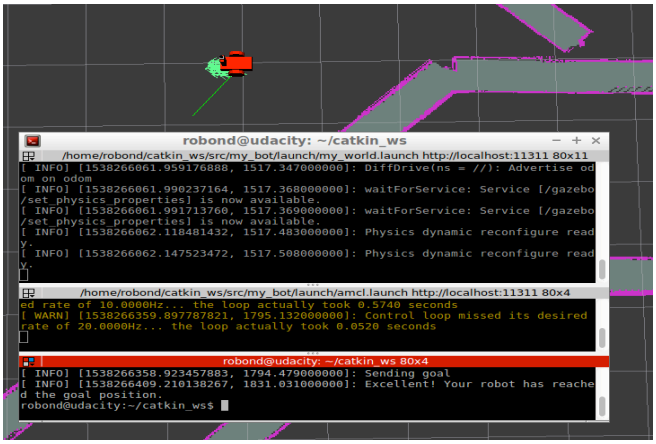

Fig. 19: My_Bot reached goal

The udacity_bot and my_bot generated very smooth local and global paths. Both models were able to successfully reach the goal position within 2 minutes. An example of my_bot navigation is shown in Fig. 17. Fig. 18 and Fig. 19 show the udacity_bot and my_bot at the goal positions. AMCL has proved to be a reliable method of localization, given that an adequate map and initial pose is provided. The most successful implementations would be in environments with a static or predictable environmental layout with little human intervention to disrupt the environment. The most obvious applications would be in manufacturing and assembly plants.

VII. Conclusion

Two robots, udacity bot and my_bot were modeled in gazebo. The dimensions, inertia and other parameters of the robot chassis, wheels, sensors, joints etc. were defined in the .xacro file. A Jackal race map was provided for the robots. Then the AMCL and Navigation Stack ROS packages were used to localize the robot in the map and to navigate the robot to a goal position. Various parameters involved in the costmap and localization optimization were tuned in order to detect obstacls and maintain appropriate distance from the walls while optimizing the local and global path. Since my_bot and udacity_bot's physical models were very similar except for the sensor locations, most of the parameters were the same for both the robots except for the obstacle_range parameter. Changing the robot's overall physical dimensions significantly would require more parameters for the AMCL and Navigation Stack to be optimized for that specific robot.

Acknowledgment

The authour would like to thank Udacity Inc. for providing some basic directions for parameter tuning for AMCL and Navigation Stack ROS packages.

References

[1] Thrun, S. (2002, August). Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence* (pp. 511-518). Morgan Kaufmann Publishers Inc..

[2] Thrun, S., Burgard, W., and Fox, D. (2005). *Probabilistic robotics*. MIT press.

[3] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y., 2009, May. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software* (Vol. 3, No. 3.2, p. 5).

[4] http://wiki.ros.org/amcl

[5] http://wiki.ros.org/navigation

[6] http://wiki.ros.org/move_base

[7] http://wiki.ros.org/costmap_2d

[8] http://wiki.ros.org/amcl#Parameters

[9] http://wiki.ros.org/base_local_planner