Robot Localization

Shu-man Lin, *Student at Udacity*

***Abstract*—Two movable robots are built using URDF to simulate a robot to perform path planning in a Gazebo environment. The AMCL probabilistic localization package in ROS uses a particle filter to track the pose of a robot against a known map which is visualized in RViz. Overall filter, laser model, and odometry model parameters used to configure the AMCL node are modified to fit each robot model. Parameters associated to the base local planner package which provides a controller that drives the mobile base are also tuned to fit the physical constraints of the robot within its environment. Further, global and local costmap parameters are chosen to control the ranges in which obstacles and free space is updated on the costmap for navigational decisions made. Each robot starts out randomly oriented in the center of a given map and localizes a randomly scattered batch of particles to estimate its position in the world so that it can successfully navigate to a fixed goal.**

***Index Terms*—Localization, Kalman filters, Particle Filters, AMCL, URDF, Robot Simulation**

I. Introduction

TO what extent can a robot's pose be recovered relative to a given map from sensor measurements and controls? This is the key question addressed in mobile robot localization. The three types of localization problems that stem from this question are the position tracking problem, global localization problem, and the kidnapped robot problem. In the position tracking problem, the robot is initialized with its starting position and must act on additional odometry updates provided by its sensors as it moves to its goal location. More generally, the global localization problem involves a robot uncertain of its initial position and requires the robot to estimate its position from scratch. Even more difficult is the kidnapped robot problem in which the robot first successfully localizes in its environment before it is teleported to another location without being told. In this case, the robot may act on estimated pre-teleportation position incorporated with future odometry and sensor readings to make false predictions based on the false assumption that it knows its current position after teleportation.

Solutions to these problems include the implementation of Kalman filters and variations of particle filters, in particular Monte Carlo localization [1] [2] [3]. These solutions formulate the problem in terms of estimating state from sensor data by forming probability distributions over the space of all world states of which probabilistic algorithms can be applied to filter for the best guess state. However, sensors are noisy and make the internal belief that a robot is in any given state unpredictable. To set up this probabilistic framework, consider the environment such as the robot's pose, configuration of the robot's actuators, robot velocity, and locations and features of surrounding objects in the environment to be characterized by the variable x_t at any time t. In addition to an environmental representation, two fundamental types of interactions between the robot and its environment: measurement data and control data are represented by z_t and u_t respectively. Examples of measurement data include camera images and range scans that give a snap shot of a momentary state of the environment, while examples of control data include setting the velocity of the robot and odometry information which carry information about the change of state in the environment. Given a state x_t at any time t, with sensor measurements, $z_0, ..., z_t$ and controls $u_0, ..., u_t$, there are two probabilities that describe the dynamical stochastic system of the robot and environment: the state transition probability and the measurement probability. The state transition probability is the probability distribution of state x_t conditioned on the previous state and current robot control and is denoted by: $p(x_t|x_{t-1}, u_t)$. It represents the evolution of that state. The measurement probability is the sensor measurement z_t conditioned on the current state x_t and is denoted by $p(z_t|x_t)$. It captures the probability of a given measurement generated by the environment. Together, these distributions are used in Bayes filter algorithm to calculate the belief: $bel(x_t) = p(x_t|z_{1,...,t}, u_{1,...,t})$ or the robot's internal knowledge about the state of the environment and its own state. Kalman filters and particle filters employ different methods to represent belief so that robot position with respect to its environment can be properly estimated. Kalman filters solve the position tracking problem and adapted Monte Carlo localization solves both the global localization problem and kidnapped robot problem.

In this project, the global localization problem and position tracking problem are solved through adapted Monte Carlo localization to navigate two different robots to a goal location in a simulated environment. The task at hand is to tune the controls and measurement device parameters to provide optimal belief estimation of the robot's location so that autonomous navigation is efficient and obstacles are avoided. In section **II.** the challenges in the problem domain is elaborated and solutions are discussed in depth. Section **III.** covers justification for parameters chosen for each robot. Section **IV.** shows the results obtained in navigational proficiency for each robot. Section **V.** discusses results and explains how AMCL can be extended to solve the kidnapped robot problem and used in other areas in industry. Finally, section **VI.** identifies areas in which the robot models can be improved and explains improvements that can be made to increase accuracy and decrease processing time.

II. Background

To tackle the problem of position tracking, Kalman filters represent belief by the mean μ_t centering around its predicted position and the covariance Σ_t representing its uncertainty.

Shu-man Lin, e-mail: impetusengine@gmail.com

The Kalman filter algorithm cycles between state predictions and measurement updates in which an initial prediction of position is given and is subsequently modified by motion measurements and sensor measurements. In linear Guassian models of Kalman filters, both the next state probability: $p(x_t|u_t, x_{t-1})$ and the measurement probability: $p(z_t|x_t)$ are linear in its arguments with added Gaussian noise and the initial belief $bel(x_0)$ represented by normal distributions to ensure that the predicted belief $bel(x_t)$ is always Gaussian. In its prediction stage, to obtain updated belief of approximate position: (μ_t, Σ_t) at time t, the controls u_t and previous state μ_{t-1} first provide a motions update to the belief denoted by $(\bar{\mu}, \bar{\Sigma}_t)$. Then the Kalman gain, K_t, is computed which specifies the degree to which the sensor measurement z_t is incorporated into $(\bar{\mu}_t, \bar{\Sigma}_t)$. In its measurement stage, the final mean and covariance: (μ_t, Σ_t) is computed by adjusting it in proportion to the Kalman gain and the deviation from actual measurement. For extended Kalman filters, more general functions replace the constant coefficients of the linear equation with state variable x_t and measurement variable z_t and a linearization is performed to approximate the true belief by a Gaussian. The limit to EKF comes from the fact that state transitions and measurements are approximated by Taylor expansions. When applying EKF's it is important to make sure uncertainties are small and functions are not multi-modal otherwise approximate linear beliefs would fail to capture the nonlinearities expressed in the robot's true belief. High accuracy of extended Kalman filters are returned if the state of the system is known with relatively high accuracy. Larger covariance tends to introduce high error upon linearization. In particular, a good initial approximation of robot position must be made in order for the robot to make a good estimate of where it is in the future. Therefore, Kalman filters are suited to address the position tracking problem but not the global localization problem or the kidnapped robot problem [1].

To solve the global localization problem, adapted Monte Carlo localization represents the belief $bel(x_t)$ by a set of n particles $\chi_t = \{x_t^{[1]}, x_t^{[2]}, ..., x_t^{[n]}\}$. The initial belief, $bel(x_0)$ is obtained by randomly generating n particles scattered randomly in the environment with uniform importance weight, $1/n$ assigned to each particle. In MCL, for each particle i, the state of the particle at time i, $x_t^{[i]}$, is first calculated by a function on motion controls u_t and the previous state $x_{t-1}^{[i]}$. A measurement is then made whereby sensor measurements z_t, and the current state $x_t^{[i]}$ determine a weight $w_t^{[i]}$ for each particle. Motion, $x_t^{[i]}$, and measurement, $w_t^{[i]}$, are then integrated into an equation to update the collection of particles $\bar{\chi}_{t-1}$ to a new collection: $\bar{\chi}_t$. Subsequently, in the process of resampling, the particles from $\bar{\chi}_t$ are sampled with probabilities proportional to $w_t^{[i]}$ and the new set of particles, χ_t, asserting new robot position, is created by this batch of samples. In adapted MCL, the resampling process is modified where either the i-th particle is replaced with a random pose in χ_t or resampling of the i-th particle occurs as in MCL. Two parameters α_{slow} and α_{fast} are pre-set to calculate values w_{slow} and w_{fast} which are values used to add random poses to χ_t with probability $max(0, 1 - w_{fast}/w_{slow})$. The values w_{fast} and w_{slow} track the short and long term average measurement likelihood: $p(z_t|z_{t-1}, u_t, m)$. If a decay occurs in measurement likelihood, short-term likelihood becomes worse than long-term likelihood and the number of random samples increases. With respect to global localization, at first, a randomized array of particles scattered across the environment starts with robot position as a guess by taking the average of particles with high uncertainty which corresponds to a situation in which the short-term average of the measurement probability is much worse than its long-term average. To ensure that particles do not localize in the wrong position, more random poses are inserted at this stage. After several more measurements, the particles become more certain of their location and cluster around the robot's true position. In this case, both short and long-term average of measurement likelihood increases and fewer random poses are drawn as the robot now only needs to track its position [1].

III. Model Configuration

The simple design of udacity_bot consists of a rectangular box chassis with front and back caster underneath the robot for stabilization and a left and a right wheel for movement. A hokuyo sensor sits on top of the chassis towards the front part of the car and a camera is attached to the front side of the car. Udacity_bot is pictured in Fig. 1.

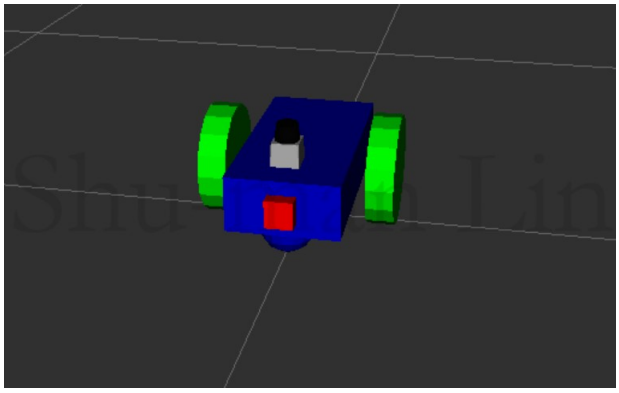

Fig. 1. Picture of Udacity_bot

On the other hand, mazebot is a three wheeled robot with a spherical chassis and flat disk belt intersecting the center of the sphere and sitting parallel to the ground to stabilize the robot. The back two wheels run on differential drive and are connected to a pair of rectangular box shaped casters with a continuous joint along the y-axis. On the opposite side of the casters, the chassis is connected to the casters with a fixed joint. In front, a caster is attached continuously to a third wheel and to the chassis so that the caster spins freely about the z-axis. A camera is placed facing forward sitting above the flat disk belt and the hokuyo sensor is attached to the top of the spherical chassis. Mazebot is pictured in Fig. 2.

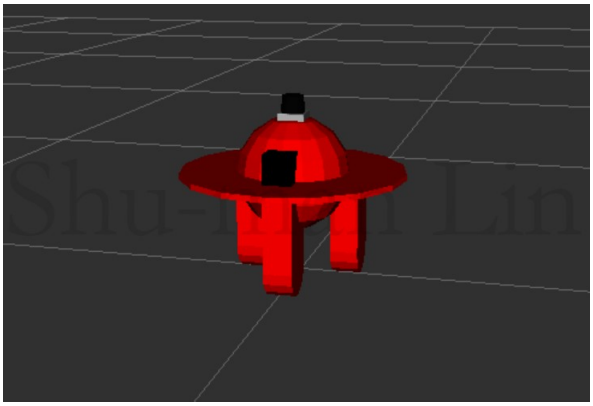

Fig. 2. Picture of Mazebot

The navigation stack [4] assumes that the robot is configured in the following manner 3.

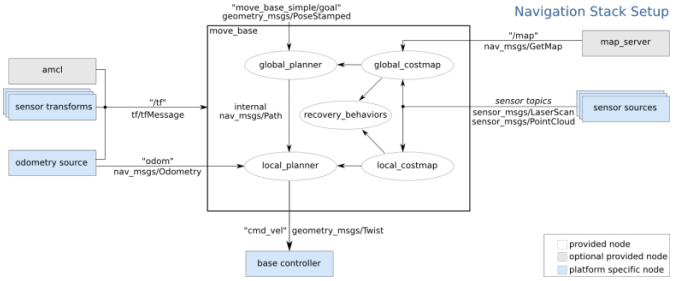

Fig. 3. Navigation Stack

This section explains the parameters that were tuned in the amcl node, global costmap node, local costmap node, common costmap node, and local planner node. The costmap nodes store information about obstacles in the world. The local costmap node is used for local planning and obstacle avoidance while the global costmap node is used to create long-term plans over the entire environment [5]. The common costmap node contains parameters shared by both the local and global costmaps. The base local planner node is responsible for computing velocity and acceleration commands to send to the mobile base of the robot and gives weight and time to forming the forward trajectory of the robot's path [6]. The amcl node controls parameters associated to the probabilistic localization system for a robot moving in 2-D [7]. It implements adapted Monte Carlo localization to track the pose of the robot against a known map.

A. Common Costmap Parameters

1) observation_sources: list of sensors that will pass information to the costmap

2) sensor_name: in brackets, this sets the parameters of the sensor named. Parameters include: "sensor_frame" name, "data_type" of message the topic uses, "topic" published to, "marking" obstacles added, and "clearing" obstacles added

3) obstacle range: maximum range sensor reading in meters for marking obstacles to be updated in costmap

4) raytrace range: maximum range sensor reading in meters for marking freespace

5) footprint: the ordered vertices of a robot outlining its polygonal shape projected onto the x-y plane with the origin centered at $(0, 0)$

6) robot_radius: the approximate radius of a robot in meters

7) inflation_radius: minimum distance in meters of an obstacle to have equal obstacle cost to obstacles at greater distance within obstacle range

8) cost_scaling_factor: set inversely proportional to the cost of a cell; setting it higher will make the decay curve more steep; when costmap curve is steep, the robot tends to be close to obstacles

9) transform_tolerance: specifies the delay in transform data that is tolerable in seconds; if the transform between the coordinate frames specified by the global_frame and the robot_base_frame parameters is transform_tolerance seconds older than ros::Time::now(), then the navigation stack will stop the robot

For udacity_bot, the common costmap parameters used are given in the following table:

Udacity_bot Common Costmap Parameters	
Parameter	**Value**
observation_sources	laser_scan_sensor
laser_scan_sensor	{sensor_frame: hokuyo, data_type: LaserScan, topic: /udacity_bot/laser /scan, marking: true, clearing: true}
obstacle_range	3.0
raytrace_range	5.0
footprint	[[0.2, 0.2], [-0.2, 0.2], [-0.2, -0.2], [0.2, -0.2]]
robot_radius	0.25
inflation_radius	0.6
cost_scaling_factor	0.5
transform_tolerance	0.4

For mazebot, the common costmap parameters used are given in the following table:

Mazebot Common Costmap Parameters	
Parameter	**Value**
observation_sources	laser_scan_sensor
laser_scan_sensor	{sensor_frame: hokuyo, data_type: LaserScan, topic: /mazebot/laser /scan, marking: true, clearing: true}
obstacle_range	1.0
raytrace_range	1.0
footprint	[[0.21,0.21], [0.21,-0.21], [-0.21,-0.21], [-0.21,0.21]]
robot_radius	0.25
inflation_radius	1.0
cost_scaling_factor	0.1
transform_tolerance	0.2 (default)

The footprint and robot_radius of both robots are set to be a little larger than the actual size of the robot. In practice robot_radius overrides footprint if both are specified. It is checked that the the robot fits adequately within the circle drawn with prescribed robot_radius so that extra room is prepared for obstacle avoidance but is not so unecessarily large that it grossly misapproximates the boundary of the robot that it needs to be aware of. For udacity_bot obstacle_range and raytrace_range are both set higher than the obtacle and raytrace ranges for mazebot. In retrospect, udacity_bot was made to successfully navigate to the goal before mazebot and setting the ranges high so that the robot could plan ahead seemed reasonable in the beginning, however assessing a costmap with closer obstacle range and more direct impact may be far superior and would reduce computation time as demonstrated with mazebot, granted, there may also be other physical constraints prohibiting udacity_bot from moving very fast as well. However, for udacity_bot, due to a lag in the system caused by system overload, transform_tolerance needed to be increased to address these costmap update issues. As discussed in [8], inflation_radius controls how far away the zero cost point is from the obstacle and cost_scaling_factor is inversely proportional to the cost of a cell. Setting it higher will make the decay curve steep. A steep costmap curve corresponds to a robot moving closer to obstacles. Going down narrow corridors, the best path is one in which the robot would prefer to stay as far away from the walls on either side and will maintain a middle path trajectory. Thus, a very low cost_scaling_factor is fixed for both robots. It is also good to make sure that the robot can round corners without bumping into them but not to make so wide a turn to the extent the robot starts to wander off in another direction. For this reason, the inflation_radius is set to be more than double the robot_radius.

B. Base Local Planner Parameters

1) max_vel_x: maximum forward velocity allowed for the base of the robot in meters/sec

2) min_vel_x: minimum forward velocity allowed for the base of the robot in meters/sec

3) max_vel_theta: maximum rotational velocity allowed for the base of the robot in radians/sec

4) min_vel_theta: minimum rotational velocity allowed for the base of the robot in radians/sec

5) min_in_place_vel_theta: minimum rotational velocity allowed for the base while performing in-place rotations in radians/sec

6) acc_lim_theta: rotational acceleration limit of the robot in meters/sec^2

7) acc_lim_x: acceleration limit of the robot in meters/sec^2

8) acc_lim_y: acceleration limit of the robot in meters/sec^2

9) escape_vel: speed used for driving during escapes in meters/sec; set to a negative number so that robot will operate in reverse

10) holonomic_robot: determines whether velocity commands are generated for a holonomic or non-holonomic robot; for holonomic robots, strafing velocity commands will be issued to the base

11) sim_time: the amount of time to forward simulate trajectories in seconds

12) sim_granularity: the step size, in meters, to take between points on a given trajectory

13) meter_scoring: activates pdist_scale and gdist_scale parameters when calculating cost

14) pdist_scale: weight for how much the controller should stay close to the global goal path it was given

15) gdist_scale: weight for how much the controller should stay close to its local goal path, also controls speed

16) occdist_scale: weight for how much the controller should attempt to avoid obstacles

17) vx_samples: the number of samples to use when exploring the x velocity space

18) vtheta_samples: the number of samples to use when exploring the theta velocity space

19) prune_plan: defines whether or not to remove points off the plan as the robot moves along the path. If set to true, points will fall off the end of the plan once the robot moves 1 meter past them.

For udacity_bot, the base local planner parameters used are given in the following table:

Udacity_bot Base Local Planner Parameters	
Parameter	**Value**
max_vel_x	5.0
min_vel_x	-5.0
max_vel_theta	5.0
min_vel_theta	-5.0
min_in_place_vel_theta	0.5
acc_lim_theta	3.2
acc_lim_x	2.5
acc_lim_y	2.5
escape_vel	-0.1 (default)
holonomic_robot	false
sim_time	10.0
sim_granularity	0.025 (default)
meter_scoring	true
pdist_scale	1.5
gdist_scale	0.5
occdist_scale	0.01 (default)
vx_samples	3 (default)
vtheta_samples	20 (default)
prune_plan	true (default)

For mazebot, the base local planner parameters used are given in the following table:

Mazebot Base Local Planner Parameters	
Parameter	**Value**
max_vel_x	1.0
min_vel_x	-1.0
max_vel_theta	1.0
min_vel_theta	-1.0
min_in_place_vel_theta	0.5
acc_lim_theta	3.0
acc_lim_x	2.5
acc_lim_y	2.5
escape_vel	-0.3
holonomic_robot	false
sim_time	3.0
sim_granularity	0.005
meter_scoring	true
pdist_scale	3.0
gdist_scale	0.1
occdist_scale	0.02
vx_samples	20
vtheta_samples	40
prune_plan	false

For udacity_bot, the velocity of the robot did not increase by very much despite setting it at a high limit value. The velocity values for mazebot were set lower so that it did not destabilize itself by going too fast on an unpredictable path to run into walls. However, by setting sim_time, sim_granularity, pdist_scale, gdist_scale, and occdist_scale properly, the robot was able to navigate a better obstacle avoiding path so that it could run faster with more assurance that it would not hit an obstacle. The sim_time was set just high enough so that mazebot would be able to predict a trajectory to safely and efficiently navigate around corners. Sim_granularity was set to be lower so that trajectory points would effectively avoid running through obstacles. The cost function used to score each trajectory comes in the following form:

$$
\begin{aligned}
cost = \ & \text{pdist\_scale*(distance to path from the endpoint} \\
& \text{of the trajectory in map cells)} \\
& + \text{gdist\_scale*(distance to local goal from the endpoint} \\
& \text{of the trajectory in map cells)} \\
& + \text{occdist\_scale*(maximum obstacle cost along the} \\
& \text{trajectory in obstacle cost)}
\end{aligned}
$$

The objective is to form a trajectory with the lowest cost. Pdist_scale is set higher relative to the gdist_scale and occdist_scale to make trajectories conform to the global path set to lead the robot to the goal. Gdist_scale is set lower to reduce variation in path since the global path is well determined. Occdist_scale is set close to the default value but increased by a small amount to ensure collisions are not an issue. On the other hand, udacity_bot has a significantly higher value of sim_time to guarantee that it can plan to turn around corners. Similar to mazebot, udacity_bot must have a high pdist_scale relative to gdist and occdist scale to maintain the right track. But udacity_bot is given more flexibility in local path planning with a higher gdist_scale than mazebot while occdist_scale is enough to keep obstacle collision at bay. Further. vx_samples and vtheta_samples are taken to be values as suggested in [8]. It is suggested that angle velocity samples should be taken to be larger than translational velocity samples because turning is generally more complicated than moving straight. For udacity_bot, these values are untouched and are left as default values.

C. Global Costmap Parameters

1) global_frame: coordinate frame the costmap should run in, for global costmap this is set to $/map$ frame

2) robot_base_frame: coordinate frame the costmap should reference for the base of the robot, for global costmap this is base_link

3) update_frequency: frequency in Hz for the costmap to be updated

4) publish_frequency: frequency in Hz for costmap to publish display information

5) static_map: determines whether or not the costmap should update itself based on a map given by the map_server, for global costmap this is set to true

6) rolling_window: determines whether costmap will remain centered around the robot as the robot moves through the world, for global costmap this is set to false

7) width: width of the map in meters

8) height: height of the map in meters

9) resolution: resolution of the map in meters/cell

For udacity_bot, the global costmap parameters used are given in the following table:

Udacity_bot Global Costmap Parameters	
Parameter	**Value**
global_frame	map
robot_base_frame	robot_footprint
update_frequency	15.0
publish_frequency	15.0
static_map	true
rolling_window	false
width	10.0 (default)
height	10.0 (default)
resolution	0.02

For mazebot, the global costmap paramters used are given in the following table:

Mazebot Global Costmap Parameters	
Parameter	**Value**
global_frame	map
robot_base_frame	robot_footprint
update_frequency	10.0
publish_frequency	10.0
static_map	true
rolling_window	false
width	5.0
height	5.0
resolution	0.02

As suggested in [8] resolution is reduced to 0.02. It is suggested in [9] that the width and height of the global costmap should be large enough to ensure the path planning is consistent over the whole map. The value set in [9] is 50x50. For the udacity specific map, the start and goal choice and environmental partitions and freespaces allows us to get away with restricting map size to 10x10 for the udacity_bot and even 5x5 for mazebot. Further, while 5 is the default value for update_frequency with a default 10x10 map, for udacity_bot, this is increased to 15.0 and for mazebot, this is increased to 10.0 without much more reason than that it works for good planning. Perhaps smaller global costmap size against a larger map would require higher frequency updates when new maps are acquired within the restricted area of the costmap against the whole background when rolling_window is true and static_map false. In addition, publish_frequency controls the rate at which the visualization of the global costmap is updated per second and needs to be set to avoid overload of the system. Without much reason but to publish as fast as is updated, the value of publish_frequency for udacity_bot and mazebot is set to be the same as the value for update_frequency.

D. Local Costmap Parameters

1) global_frame: coordinate frame the costmap should run in, for local costmap this is set to $/odom$ frame

2) robot_base_frame: coordinate frame the costmap should reference for the base of the robot, for local costmap this is base_link

3) update_frequency: frequency in Hz for the costmap to be updated

4) publish_frequency: frequency in Hz for costmap to publish display information

5) static_map: determines whether or not the costmap should update itself based on a map given by the map_server, for local costmap this is set to false

6) rolling_window: determines whether costmap will remain centered around the robot as the robot moves through the world, for local costmap this is set to true

7) width: width of the map in meters

8) height: height of the map in meters

9) resolution: resolution of the map in meters/cell

For udacity_bot, the local costmap parameters used are given in the following table:

Udacity_bot Local Costmap Parameters	
Parameter	**Value**
global_frame	odom
robot_base_frame	robot_footprint
update_frequency	15.0
publish_frequency	15.0
static_map	false
rolling_window	true
width	5.0
height	5.0
resolution	0.02

For mazebot, the local costmap parameters used are given in the following table:

Mazebot Local Costmap Parameters	
Parameter	**Value**
global_frame	odom
robot_base_frame	robot_footprint
update_frequency	15.0
publish_frequency	15.0
static_map	false
rolling_window	true
width	4.0
height	4.0
resolution	0.02

As suggested in [8] resolution is reduced to 0.02. As suggested in [9], a minor value for height and width of the local costmap should be used to avoid overload of the system. The author of [9] sets it to 2.5x2.5 whereas the default value is 10x10. The udacity_bot local costmap is set to a frame of 5x5 while mazebot has a 4x4 width by height setting. These values along with proper sim_time are taken to suit a minimum path planning length to avoid paths that may cause the robot to run into walls because of a slight separation in obstacles creates a path through a space to the farthest end of the global path that is not a separation wide enough for robot passage or if the robot gets confused and tries to plan a shorter path to goal that mistakenly goes through a wall. In [9] it is explained that high update rate of the local costmap is required to ensure that dynamic obstacles are perceived as soon as possible. The author in [9] sets the update_frequency to the default value to 5Hz, but the value is increased to 15Hz for both udacity_bot and mazebot. Publish_frequency for the

purpose of visualization is also maintained at the same rate as update at 15Hz.

E. AMCL parameters

1) min_particles: minimum allowed number of particles

2) max_particles: maximum allowed number of particles

3) transform_tolerance: time of which to post-date the transform that is published

4) odom_model_type: the type of model used here is "diff-corrected"

5) odom_alpha1: the expected noise in odometry's rotation estimate from rotational component of robot's motion

6) odom_alpha2: the expected noise in odometry's rotation estimate from translational component of robot's motion

7) odom_alpha3: the expected noise in odometry's translation estimate from the translational component of robot's motion

8) odom_alpha4: the expected noise in odometry's translation estimate from the rotational component of robot's motion

9) recovery_alpha_slow: exponential decay rate for the slow average weight filter; this parameter is the short-term average of the measurement likelihood, used in deciding when to recover by adding random poses

10) recovery_alpha_fast: exponential decay rate for the fast average weight filter; this paramter is the long-term average of them measurement likelihood, used in deciding when to recover by adding random poses.

11) laser_model_type: either beam, likelihood_field, or likelihood_field_prob

12) laser_likelihood_max_distance: maximum distance to do obstacle inflation on map for use in the likelihood_field model

For udacity_bot, the AMCL parameters used are given in the following table:

Udacity_bot AMCL Parameters	
Parameter	**Value**
min_particles	50
max_particles	300
transform_tolerance	0.2
odom_model_type	diff-corrected
odom_alpha1	0.05
odom_alpha2	0.05
odom_alpha3	0.01
odom_alpha4	0.05
recovery_alpha_slow	0 (default, disabled)
recovery_alpha_fast	0 (default, disabled)
laser_model_type	likelihood_field
laser_likelihood_ max_distance	5.0

For mazebot, the AMCL parameters used are given in the following table:

Udacity_bot AMCL Parameters	
Parameter	**Value**
min_particles	100 (default)
max_particles	300
transform_tolerance	0.05
odom_model_type	diff-corrected
odom_alpha1	0.02
odom_alpha2	0.02
odom_alpha3	0.05
odom_alpha4	0.01
recovery_alpha_slow	0.001
recovery_alpha_fast	0.1
laser_model_type	likelihood_field
laser_likelihood_ max_distance	2.0 (default)

For optimal localization results for both robots, the odom_alpha values are set to nonzero values close to zero since a precise sensor requires some noise in order to make an accurate probabilistic prediction for position with a well-conditioned resampling step. The min number of particles is reduced to 50 from an original default value of 100 and the max number of particles is reduced to 300 from a default value of $5,000$ in the udacity_bot model since the high particle number is unecessary in this goal finding task. Since the initial particle array is locally spread in a region about the origin of the map at $(0, 0)$, which also happens to be the origin position of the robot, a sparser initial array of particles is enough to make a reasonable first prediction of the robot's initial position which reduces the harder global localization problem to the easier position tracking problem faster. This also allows for the recovery_alpha_slow and recovery_alpha_fast values to remain at their default value of being disabled since these values are only needed to solve a more severe global localization problem in which the occasional random particle is injected into the distribution to avoid localization failures when a decay in measurement likelihood occurs. The min_particles used for mazebot is set to the default value but the max_particles is reduced to 300. The recovery_alpha_slow and recovery_alpha_fast values are the same as used in [9] but this is unecessary to set for this project with the same reason as discussed for udacity_bot.

IV. Results

A. Results for Udacity_bot

Udacity_bot begins with a particle distribution that is on average a relatively close approximation to its true initial position as shown in 4 and 5.

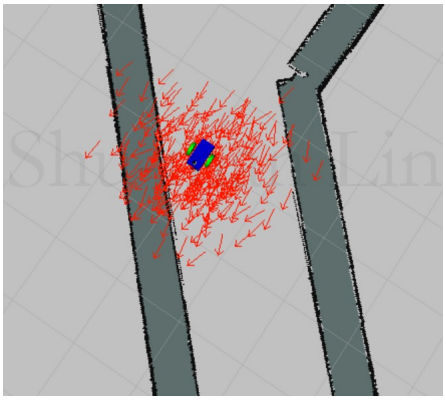

Fig. 4. Initial estimated pose of Udacity_bot with initial particle array

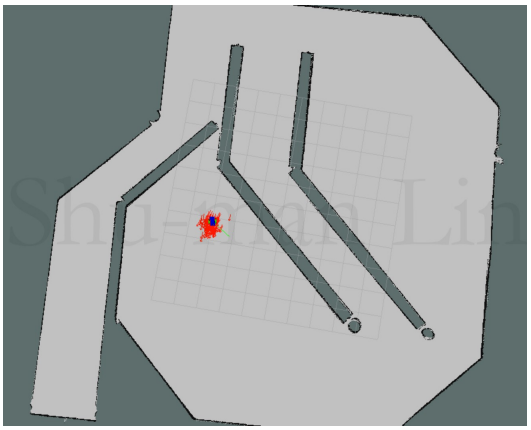

Fig. 5. Initial estimated pose of Udacity_bot in world map

Udacity_bot takes approximately 16.5 minutes to reach the goal and when it comes to a close approximation of its final position, it takes the robot about 3.5 minutes to orient itself in the correct direction. From start to finish, udacity_bot follows local trajectories that oscillate along the estimated global path to the goal but eventually makes it to the end with a particle cloud around it that gives a good approximation of its changing states as it inches forward to the goal. A video of the robot's autonomous navigation is provided in the following link: https://youtu.be/eGwkyLj_-Wc. Nearing the goal, the robot takes small incremental steps to locate the final position then spins in place to find the final yaw angle. The end particle distribution is shown in 6 and 7.

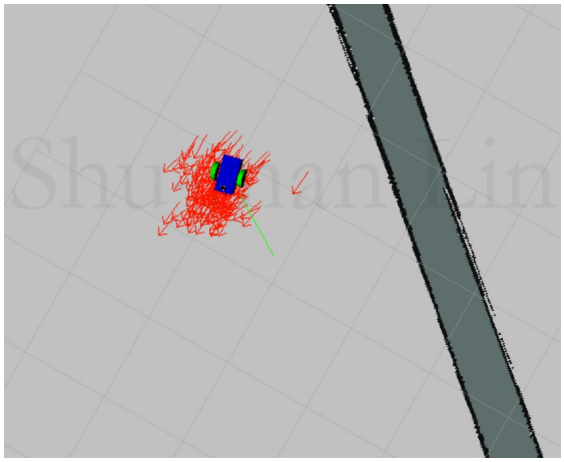

Fig. 6. Final estimated pose of Udacity_bot with final particle array

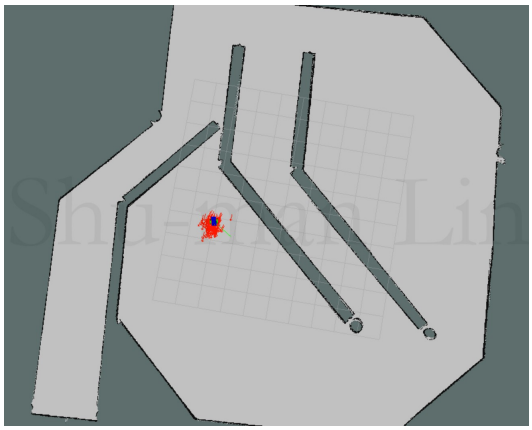

Fig. 7. Final estimated pose of Udacity_bot in world map

B. Results for Mazebot

Initially, mazebot has its initial position approximated by an array of particles close to its true initial position as shown in 8 and 9.

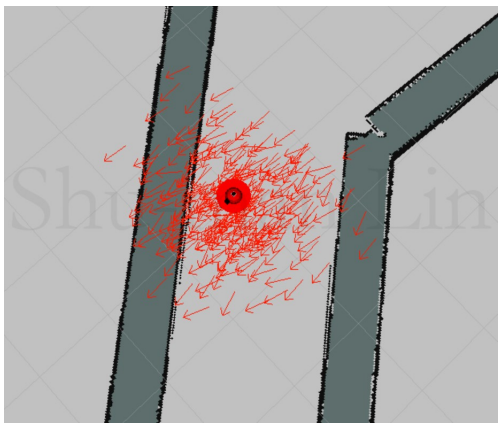

Fig. 8. Initial estimated pose of Mazebot with initial particle array

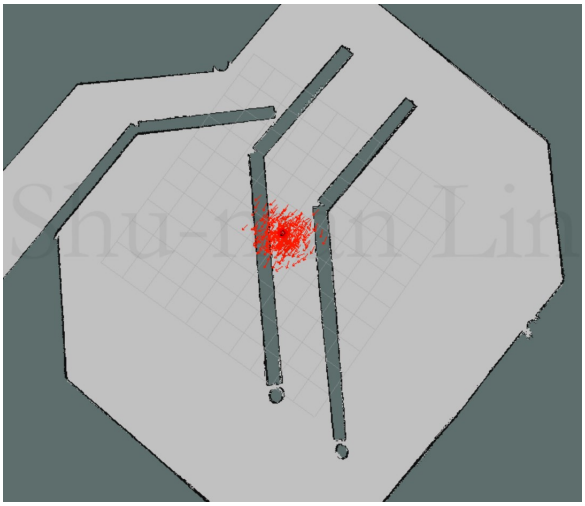

Fig. 9. Initial estimated pose of Mazebot in world map

Mazebot takes approximately 3 minutes to reach the goal location and about 3 minutes to slightly modify its position near the goal and orient itself in the correct direction. From start to finish, mazebot follows a path closely trailing alongside the estimated global path to the goal and maintains a particle cloud with an average approximation close to its trajectory. A video of the robot's autonomous drive to the goal is given in the following link: https://youtu.be/A_xRMnOvkCg. The final estimated pose of mazebot is shown in 10 and 11.

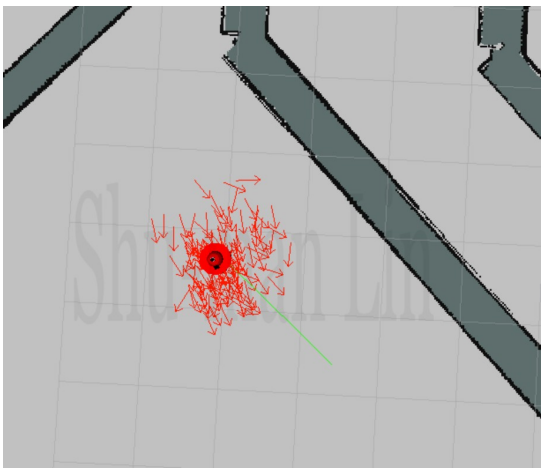

Fig. 10. Final estimated pose of Mazebot with final particle array

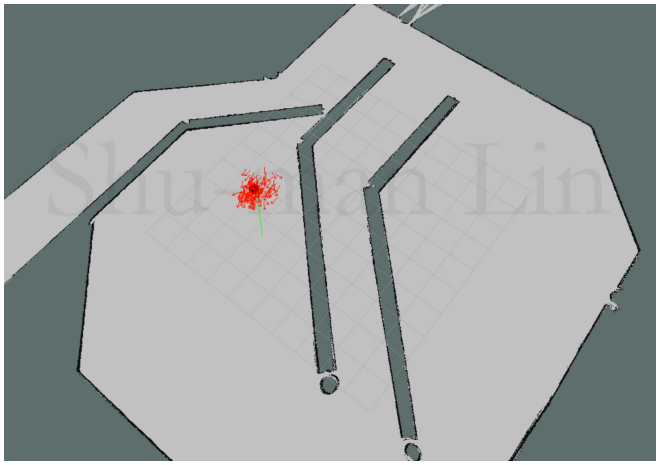

Fig. 11. Final estimated pose of Mazebot in world map

V. Discussion

Although the udacity_bot reaches the goal, the time that it takes to get there is extremely long compared to mazebot. It takes udacity_bot 5 times as long to reach the goal as it does mazebot. In addition, the end orientation time for both robots is fairly lengthy as well. The time in which it takes mazebot to precisely orient is the same time as it takes to reach the goal. For shorter goal times, goal tolerances could be relaxed to include a greater area of approximation as a final solution, but this is no more of a solution since it comes at the cost of degrading accuracy. Precision takes its time. Nonetheless, perhaps the udacity_bot model has something to learn from the mazebot model in that reducing sim_time and gdist_scale, decreasing obstacle_range and raytrace_range, and shrinking costmap size could, when taken together, lead to a better udacity_bot model by increasing accuracy in global path tracing and thereby decreasing time spent reaching the goal. In addition, other constraints to the slow speed of the robot may be removed if the physical parameters were modified slightly so that the casters don't drag on the ground.

On the other hand, the localization problem for both robots can be reduced from the global localization problem to the robot tracking problem due to close proximty of the initial particle array aggregated near the true position of the robot, thereby giving a close approximation of the robot starting position from the start. The AMCL parameters, in both cases, successfully localize each robot after an initial sensor reading and are able to maintain that fidelity to high position approximation as each robot moves towards the goal. Further, the AMCL parameters may be modified to accomodate solving the kidnapped robot problem by changing the recovery_alpha_slow and recovery_alpha_fast parameters to positive values that act to admit random samples in proportion to the quotient of long-term to short-term measurement likelihood when the short-term likelihood is larger than the long-term likelihood. The act of teleporting the robot to a new position with the robot carrying the history of localizing in another region is no longer a problem since sensing in the new location would cause a decay in measurement likelihood which would reintroduce random particles into the environment so that the localization process can be refreshed.

Inside of buildings, applications of AMCL or MCL in industry include its integration into autonomous vehicles for transport of goods within warehouses or in warehouse autonomous flight vehicles for merchandise tagging [10]. AMCL could be the core algorithm for hospital robots transporting supplies and service employed grocery store robots patroling the aisles where prior maps of the floor plan of buildings are recorded for reference position mapping. In the outdoors, tasks such as driving in a crowded urban environment could employ AMCL as a back-up localization tool where a prior mapping of the environment is used for global localization and position tracking on roads obscured by buildings where GPS becomes degraded [11]. Sidewalk food delivery carts could employ AMCL to get from food source to home sink by drawing city maps with key block-to-block street view features as location markers. Aerial traffic of multiple localized agents could be synchronized by combining their sample sets for coordinated, zero-collision package delivery as sensors track their position relative to ground landmarks. Essential to using the AMCL algorithm is preparing a good map that is sensor recognizable.

VI. Future Work

In this case study of udacity_bot, decreasing the processing time by lowering parameters that would consume more compute time such as sim_time, obstacle_range, raytrace_range, costmap size and gdist_scale variance would at first glance seem to lower the accuracy of the locally planned trajectory since less information is used to plan for the future, however, the uniform lowering of these parameters to a lower bound limit similar to the parameters chosen for mazebot may increase accuracy overall. This may be due to the fact that a wider gap between gdist and pdist scales may orient a more decisive local plan to follow the global path to the goal with low enough cost_scaling_factor to maintain a straight, middling path down corridors and a minimum costmap size and sim_time to plan around turning corners. Further, as was discussed previously, the boundaries of the casters may be causing unwanted friction with the ground making the robot run slow so a cut to the size of the radius of the casters may speed up rolling motion.

As for mazebot, any further reduction in processing time would severely handicap both accuracy and flexibility of the robot to navigate without obstacle collision to the goal. Certainly velocity and acceleration limits may be increased, but that is at the risk of robot motion instability and the robot toppling over. However, if the robot falls at certain angles, the robot has demonstrated that it can resurrect itself and carry on its way, but to maintain certainty of its ability to recover, it is best to steer the robot away from obstacles at all costs as is common practice for most creatures and vehicles that have an upright position, unless there is another way to mold the body of the robot that is more resilient to recovery when fallen. Is it possible to design a more robust robot that is a spherically symmetric limbed rolling robot with symmetric sensors so that if the robot falls over it is able to pickup and move with a different body and sensor orientation? Perhaps the closest answer to this question is given by the invention of the "platonic beast" [12] which is a robot with a symmetric polyhedron base such that each face of the base extends a leg so that it is robust to toppling and adopts a novel "rolling" gait. Certainly experimenting with spherical triangulations and placing a "rolling" leg at each vertex of the triangulation with a sensor at the centroid of each triangle while only activating the sensor positioned with respect to the top of the sphere to navigate is a curiosity that could be of interest to explore. Without deviating for the original make, another question is how small or large could this prototype be made and for what utility? It is up to future work to address these questions.

References

[1] S. Thrun, W. Burgard, D. Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press. 2005.

[2] S. Thrun, D. Fox, W. Burgard, F. Dellaert. *Robust Monte Carlo Localization for Mobile Robots*. Artificial Intelligence. online: http://robots.stanford.edu/papers/thrun.robust-mcl.pdf, 2001.

[3] S. Thrun. *Particle Filters in Robotics*. Uncertainty in AI. online: http://robots.stanford.edu/papers/thrun.pf-in-robotics-uai02.pdf, 2002.

[4] ROS wiki: navigation/Tutorials/RobotSetup. *Setup and Configuration of the Navigation Stack on a Robot*. ROS wiki: Open Source Robotics Foundation. online: http://wiki.ros.org/navigation/Tutorials/RobotSetup, 2015.

[5] ROSwiki: costmap_2d. *Package Summary*. ROS wiki: Open Source Robotics Foundation. online: http://wiki.ros.org/costmap_2d, 2018.

[6] ROS wiki: base_local_planner. *Package Summary*. ROS wiki: Open Source Robotics Foundation. online: http://wiki.ros.org/base_local_planner, 2018.

[7] ROS wiki: amcl. *Package Summary*. ROS wiki: Open Source Robotics Foundation. online: http://wiki.ros.org/amcl, 2017.

[8] K. Zheng. *ROS Navigation Tuning Guide*. online: http://kaiyuzheng.me/documents/navguide.pdf. Sept. 2, 2016.

[9] S. Gangl. *Robotic exploration for mapping and change detection*. Master tehsis at ikg and GIH. online: https://www.ikg.uni-hannover.de/fileadmin/ikg/staff/thesis/finished/documents/ma_gangl.pdf. Oct. 28, 2014.

[10] G. Vasiljevic, D. Miklic, I. Draganjac, Z. Kovacic, P. Lista. *High-accuracy vehicle localization for autonomous warehousing*. Robotics and Computer-Integrated Manufacturing Vol. 42. online: https://bib.irb.hr/datoteka/816062.warehousing_localization.pdf. Dec. 2016.

[11] Z. J. Chong, B. Qin, T. Bandyopadhyay, T. Wongpiromsarn, E. S. Rankin, M. H. Ang Jr., E. Frazzoli, D. Rus, D. Hsu, K. H. Low. *Autonomous Personal Vehicle for the FIrst- and Last- Mile Transportation Services*. Cybernetics and Intelligent Systems (CIS). IEEE 5th International Conference on 17-19 Sept. 2011. online: http://ares.lids.mit.edu/fm/documents/autonomous_firstlast.pdf.

[12] D. Pai, R. Barman, S. Ralph. *Platonic Beasts: Spherically Symmetric Multilimbed Robots*. Autonomous Robots Vol 2. Issue 3. online: https://pdfs.semanticscholar.org/8368/c54e03376ef317fdf718c25e0959de258ee7.pdf 1995.