Where Am I

Pramod Kumar

Abstract—Confinement is the test of deciding the robot's posture in a mapped domain. This is finished by executing a probabilistic calculation to channel boisterous sensor estimations and track the robot's position and introduction. This paper centres around restricting a robot in a known mapped condition utilizing Adaptive Monte Carlo Localization or Particle Filters strategy and send it to a objective state.ROS, Gazebo and RViz were utilized as the instruments of the exchange to recreate the earth and programming two robots for performing confinement.

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

1 Introduction

THE "Where m I" project is based on localization. Robot need the localization for probabilistic estimation of the current position and orientation in world scenario to take effective decision to take action to control it. If robot doesn't know its location then it is unable to take effective decision. There are three types of localization maps.

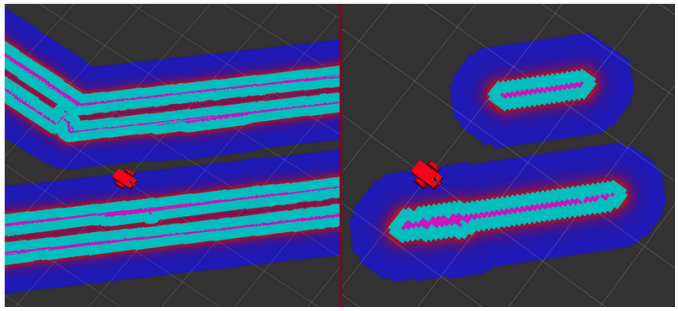

Fig. 1. Robot Localization.

1.1 Local Localization:

It is simplest localization. It is used to track the position of robot. Robot knows its initial position and its estimates the robot position and challenge entails estimation when it is out in the environment. This problem is not trivial as there is always some uncertainty in robot motion.However, the uncertainty is limited to regions surrounding the robot.

1.2 Global Localization:

This is a more complicated localization problem. In this case, the robots initial pose is unknown and the robot must determine its pose relative to the ground truth map. The amount of uncertainty is much higher.

1.3 The kidnapped robot problem:

This is the most challenging localization problem. This is just like the global localization problem, except that the robot may be kidnapped at any time and moved to a new location on the map.

2 Background

For the localization issue, an extensive variety of algorithms are accessible going from Monte Carlo Localization, Extended Kalman Filter to Markov lastly Grid Localization. The Monte Carlo Localization algorithm or MCL, is the most famous limitation calculations in robotics. After MCL is integrated, the robot will explore inside its known map and gather tangible data utilizing RGB camera and run discoverer sensors. MCL will utilize these sensor estimations to monitor the robot's posture. MCL is regularly alluded to as Particle Filter Localization, since it utilizes particles to limit the robot. These particles are virtual elements that resembles the robot. Each particle has a position and orientation and it represent a guess where the robot might be located. These particles are re-inspected each time the robot moves and faculties its condition. For this project, an altered adaptation of this calculation known as 'Adaptive Monte Carlo Localization' was utilized in light of the fact that this modified algorithm progressively alters the quantity of particles over a timeframe, as the robot explores around the map, thus making the procedure more practical.

A portion of the current difficulties of limitation includes:

a)It doesn't work well in an unmapped domain.

b) In the multi robot localization problem, which includes a group of robots which at the same time try to decide their poses in a known environment, conditions are made in the pose assessments of individual robots that present significant difficulties for the structure of the estimator. [1]

2.1 Kalman Filters

The Kalman filter is an estimation algorithm that is extremely unmistakable in controls. It is utilized to assess the estimation of a variable progressively as the information is being gathered. This variable can represent to the position or speed of the robot, or on the other hand even the temperature of a procedure. The reason that the Kalman filter is so net commendable is since it can take information with a lot of vulnerability or noise in the estimation and give an extremely precise estimate of the real qualities in a brief timeframe. In contrast to other estimation algorithm, it doesn't rely upon a lot of information to come in request

to calculate a precise estimate. The Kalman filter was concocted amid the Apollo program. It was utilized to enable Apollo to enter the orbit of the moon. Since its success with the Apollo program, Kalman filter has turned out to be one of the most pragmatic algorithm in the field of control engineering.

The Kalman filter works consistently between two stages. The Kalman filter delivers a estimate of the condition of the framework as a normal of the framework's anticipated state what's more, of the new estimation utilizing a weighted normal. The motivation behind the weights is that qualities with better (i.e., smaller) evaluated vulnerability are "trusted" more. The weights are calculated from the covariance, a measure of the estimated uncertainty of the prediction of the systems state. The aftereffect of the weighted normal is another state estimate that lies between the anticipated and estimated state, what's more, has a superior evaluated uncertainty than either alone. This procedure is rehashed at each time step, with the new estimate and its covariance informing the expectation utilized in the accompanying cycle. This implies the Kalman filter works recursively and requires just the last "best guess", instead of the whole history, of a framework's state to calculate another state. The linear Kalman filter accept that the output is corresponding to the information and thus it very well may be just connected to straight frameworks. This impediment is overwhelmed with Extended Kalman channels, which can be connected to non-linear systems, which is more relevant in robot technology as true frameworks are more frequently non-linear than linear. Likewise Linear Kalman filter accept that both the earlier and the posterior follows a unimodal Gaussian distribution. In any case, in real world, that is only from time to time the case. Extended Kalman filter conquer this issue by linear approximation of the back dissemination after a non-linear transformation.

2.2 Particle Filters

The Monte Carlo Localization or Particle Filters utilizes virtual particles to appraise a robot's posture. With MCL, particles are at first spread consistently and haphazardly all through the whole map. Much the same as the robot, every particle has a xy coordinate and an orientation vector. So each of these particles represent to the theory of where the robot may be. Notwithstanding the 3d vector, particles are assigned a weight. The weight of a particle is the distinction between the robot's actual posture and the particle's anticipated posture. The significance of a particle is subject to its weight. The greater the particle, more exact it is. Particles with more weights will probably survive amid the re-sampling process. After the re-sampling process, particles with noteworthy weights will probably survive though others will probably pass on. At last after a few emphasess of the calculation and after various phases of re-sampling, particles will merge and estimate the robot's posture. The MCL calculation evaluates the back dissemination of a robot's position and orientation dependent on tactile data. This procedure is known as Bayes Filter. Utilizing a Bayes filtering approach, the condition of a dynamical framework can be assessed from the sensor estimations.

2.3 Comparison / Contrast

There are sure huge advantages of Monte Carlo Localization over Extended Kalman Filter algorithm. Right off the bat, MCL is easy to code. Furthermore, MCL represents non-Gaussian distribution and can rough some other practical important appropriation. This implies MCL is unhindered by a linear Gaussian state based presumption as on account of EKF. This enables MCL to demonstrate a significantly more prominent assortment of conditions uncommonly since this present reality can't be continuously displayed by Gaussian disseminations. Thirdly, in MCL, the computational memory and the goals of the arrangement can be controlled by changing the quantity of particles dispersed consistently and arbitrarily all through the map. The general contrast between MCL and EKF is depicted in the table underneath. So MCL is for the most part more advantageous than EKF and henceforth MCL will be executed with the end goal of this project.

	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. 2. EKF Vs MCL comparision.

3 Simulations

The simulations were done in a ROS environment utilizing Gazebo and RViz. The navigation stack [4] can be visualized as follows:

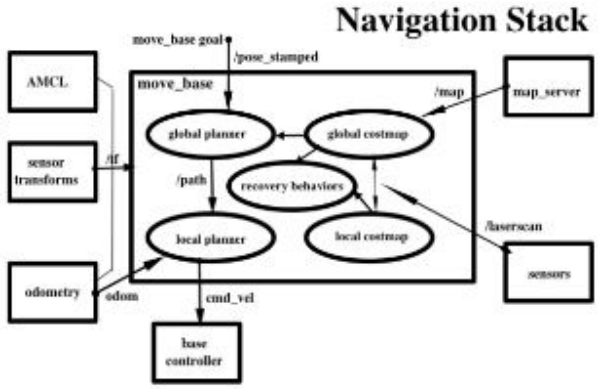

Fig. 3. Navigation Stack.

The entire simulation process was completed on both UdacityBot and PramodBot. The algorithm and map perception process were done in RViz. We can likewise observe the robot in action inside the Gazebo simulation environment.

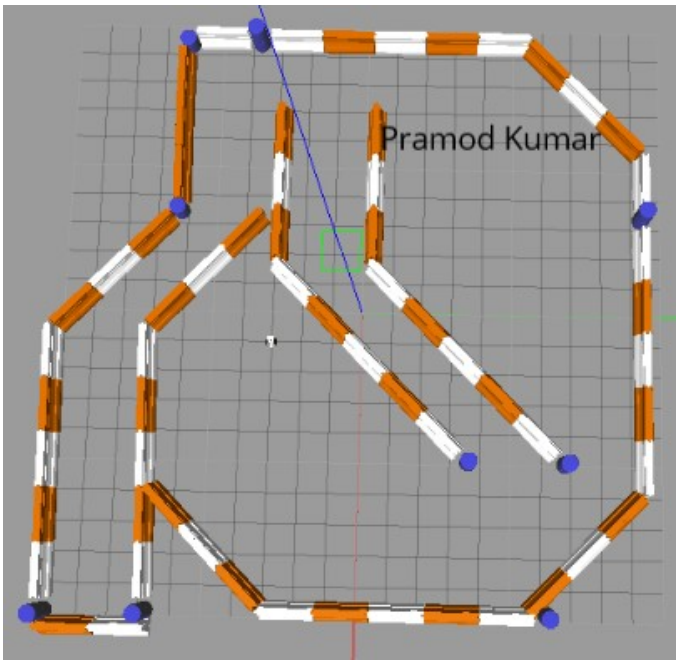

Fig. 4. Robot in Gazebo environment.

At first, for both the bots, particles are extremely scattered showing extraordinary vulnerability in the robot position. Now, sensors have not yet given any data in regards to the area. Fig 5 demonstrates the extraordinary vulnerability of the robot's position represented by the particles signified by red arrows.

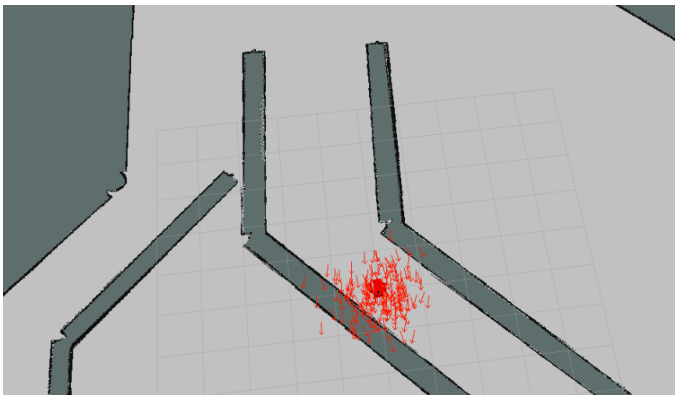

Fig. 5. High uncertainty in robot position.

After the reenactment is begun, the localization process begins from taking the sensor estimations and step by step make strides. After the algorithm converges, the particles successfully delineates the posture of the robot in the map, along these lines making the robot effectively explore through the labyrinth and reach the objective state.

3.1 Achievements

For this task, two robots were sent in the simulation environment. The benchmark model or UdacityBot and the specially designed model or PramodBot.

UdacityBot: The UdacityBot at first began towards the north of the delineate from its neighborhood costmap, it computed the path to the objective from its beginning position to be shorter than some other ways. Yet, it before long found the nearness of a deterrent furthermore, the inconceivability of achieving the objective through that path. At that point it pivoted and achieved the objective through the second most briefest path. Fig. 6 delineates UdacityBot at the objective position.

PramodBot: The PramodBot likewise displayed comparable conduct and began towards north of the map, before finding the difficulty of that route and after that it achieved the objective position through pretty much a similar route UdacityBot has taken. Fig. 7 delineates PramodBot at the objective position.

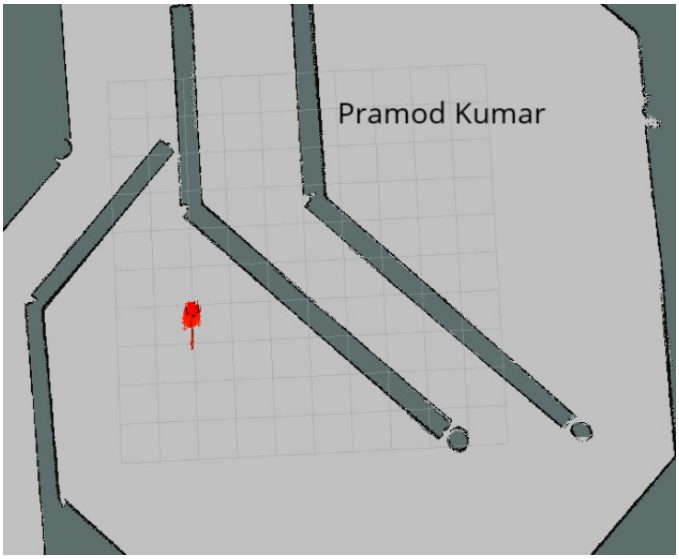

Fig. 6. UdacityBot at the goal position.

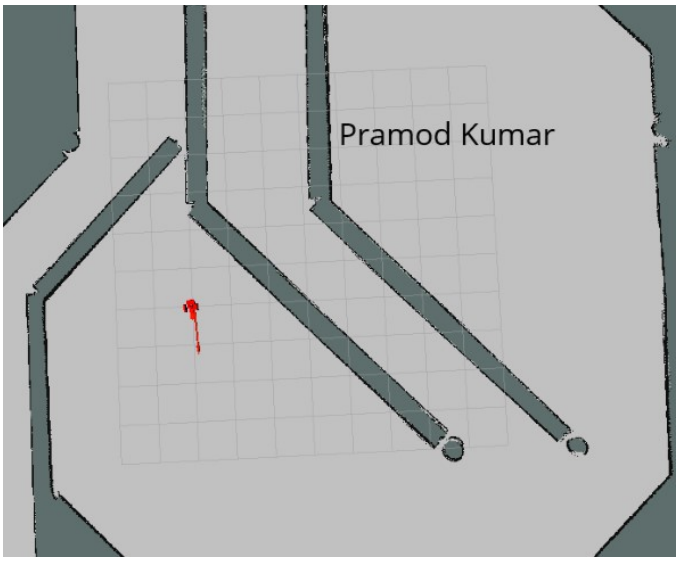

Fig. 7. PramodBot at the goal position.

3.2 Benchmark Model

3.2.1 Model design

The robot's structure contemplations incorporated the extent of the robot and the format of the sensors. They are

examined underneath.

a)Maps: The Clearpath jackalrace.yaml and jackalrace.pgm were used to create the maps.

b)Meshes: The laser scanner which was used in the robot for detecting obstacles is the Hokuyo scanner. The mesh hokuyo.dae was used to render it.

c)Launch files: Three launch files were used. They are as follows:

1) robot_description.launch: This launch file defines the joint_state_publisher which sends fake joint values, robot state publisher which sends robot states to tf and robot description which defines and sends the URDF to the parameter server.
2) amcl.launch: The amcl package relies entirely on the robots odometry and the laser scan data. This file launches the AMCL localization server, the map server, the odometry frame, the move base server and the trajectory planner server.
3) udacity_world.launch: This is the primary launch file which contains the robot_description.launch file, the gazebo world and the AMCL localization server. It also spawns the robot and launches RViz. Fig. 8 depicts the connection graph between the discussed nodes. Fig. 9 depicts the UdacityBot at the goal location and the AMCL particles as seen in RViz.

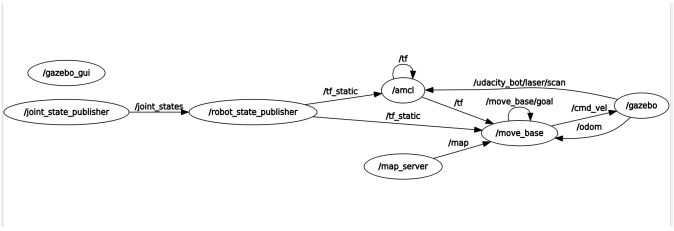

Fig. 8. Node relations.

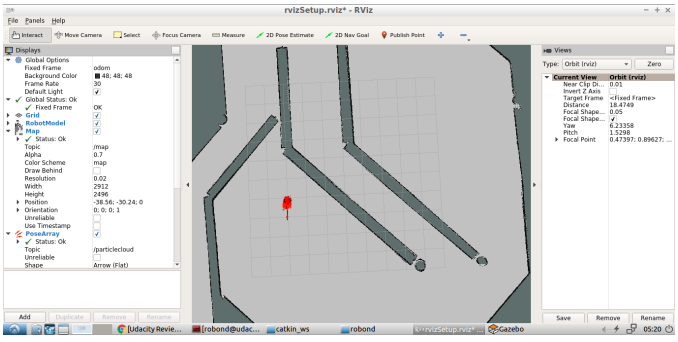

Fig. 9. RViz interface with UdacityBot at goal position.

d)Worlds: This project was done in two worlds.

1) Udacity world: This is the original blank world where the robots were created and prototyped. This defines the ground plane, the light source and the world camera.
2) jackal_race world: This world defines the maze.

e)URDF: The URDF files defines the shape and size of the robot. Two files were used as URDFs:

1) udacity_bot.xacro: Provides the shape and size of the robot in macro format. For the UdacityBot, a fixed base is used. A single link, with the name defined as chassis, encompassed the base as well as the caster wheels. Each link has specific elements, such as the inertial or the collision elements. The chassis is a cuboidal (or box), whereas the casters are spherical as denoted by their geometry tags. Each link (or joint) has an origin (or pose) defined as well. Every element of that link or joint will have its own origin, which will be relative to the links frame of reference. For this base, as the casters are included as part of the link (for stability purposes), there is no need for any additional links to define the casters, and therefore no joints to connect them. The casters do, however, have friction coefficients defined for them, and are set to 0, to allow for free motion while moving. Two wheels were attached to the robot. Each wheel is represented as a link and is connected to the base link (the chassis) with a joint. For each wheel, a collision, inertial and visual elements are present. 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 its 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. For the UdacityBot, two sensors were used. A camera and a Laser range-finder (hokuyo sensor).
2) udacity_bot.gazebo: This file was included as the URDF file is unable to make the robot take pictures with the camera or detect obstacle with the Laser range-finder. This file contains 3 plugins, one each for the camera sensor, the hokuyo sensor and the wheel joints. It also implements a differential drive controller. Fig. 10 depicts the navigation goal messages.

3.2.2 *Packages Used*

A ros package called udacity_bot was designed for this project. The structure of this package is shown below.

- config
- images
- launch
- maps
- meshes
- src
- urdf
- worlds

This package, along with the AMCL and the navigation stack packages were crucial for the success of the mobile robot in performing a localization task. Table 1 describes UdacityBot setup instructions.

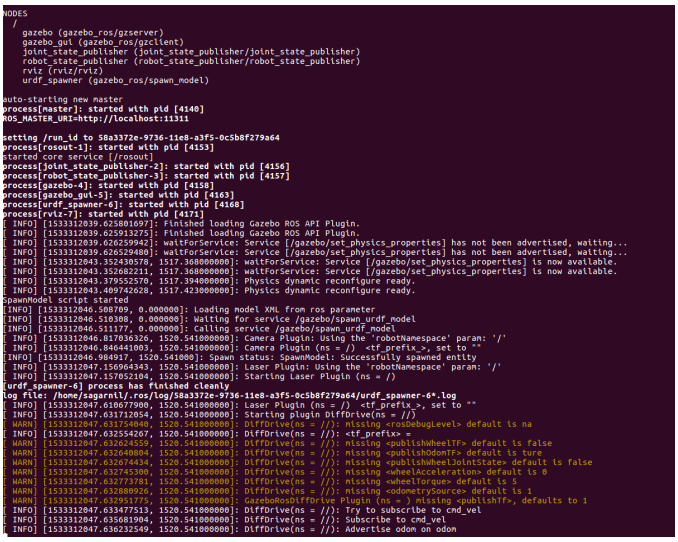

Fig. 10. Navigation goal messages.

TABLE 1
Udacity Bot setup instructions

Udacity Bot Body		
Part	Geometry	Size
Chassis	Cube	0.4 x 0.2 x 0.1
Back and Front Casters	Sphere	0.0499 (radius)
Left and Right wheels	Cylinders	0.1 (radius), 0.05 (length)
Camera Sensor	Link Origin Shape-size Joint Origin Parent Link Child Link	[0, 0, 0, 0, 0, 0] Box - 0.05 x 0.05 x 0.05 [0.2, 0, 0, 0, 0, 0] chassis camera
Hokuyo Sensor	Link Origin Shape-Size Joint Origin Parent Link Child Link	[0, 0, 0, 0, 0, 0] Box - 0.1 x 0.1 x 0.1 [0.15, 0, 0.1, 0, 0, 0] chassis hokuyo

3.2.3 Parameters

Exploring, adding and tuning parameters for the AMCL and move base packages were some of the main goals of this project. For effective parameter tuning, ROS basic navigation tuning guide [6] and Kaiyu Zhengs navigation tuning guide [7] were heavily used. The parameters were iteratively tuned to see what works best for the UdacityBot. The AMCL parameters were tuned as follows:

The min and max particles parameters were set to 25 and 200 in order to prevent over-usage of CPU. Increasing the max particles did not improve the robots initial ability to find itself with certainty. The transform tolerance was one of the main parameters to tune. 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. The maximum amount of delay or latency allowed between transforms is defined by the transform tolerance parameter. It was finally set to 1.25 for the cost-maps and 0.2 for the amcl package.

The laser model parameters like laser max beams, laser z hit and laser z rand, were kept as default as the obstacles were clearly detected in the local cost-maps with them as is. Fig. 11 depicts this observation. The odom model type was kept as diff-corrected as this mobile robot followed a differential drive. There are additional parameters that are specific to this type.

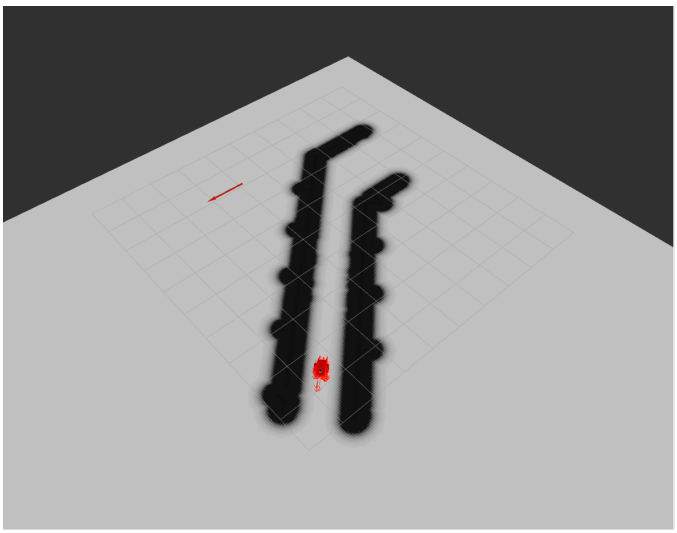

Fig. 11. Local cost-map with AMCL particles converged.

the odom alphas (1 through 4). These parameters define how much noise is expected from the robots movements/ motions as it navigates inside the map. They were kept at 0.005, 0.005, 0.010 and 0.005 respectively obtained from trial-error method. Table 2 depicts the AMCL parameters used.

TABLE 2
AMCL parameters

AMCL Parameters	
Parameter	Value
odom frame id	odom
odom model type	diff-corrected
transform tolerance	0.2
min particles	25
max particles	200
initial_pose_x	0.0
initial pose y	0.0
initial pose a	0.0
laser z hit	0.95
laser z short	0.1
laser z max	0.05
laser z rand	0.5
laser sigma hit	0.2
laser lambda short	0.1
laser model type	likelihood field
laser likelihood max dist	2.0
odom alpha1	0.005
odom alpha2	0.005
odom alpha3	0.010
odom alpha4	0.005

The move base parameters were tuned as follows:

The obstacle range parameter was modified to have a greater value of 1.5. This parameter depicts the default maximum distance from the robot (in meters) at which an obstacle will be added to the cost-map. The raytrace

range parameter was modified to a higher value of 4.0. This parameter is used to clear and update the free space in the cost-map as the robot moves.

Two parameters in the global and local cost-maps were also changed. They are:

1. update frequency: This value was set to 10.0. This is the frequency in Hz for the map to be updated.

2. publish frequency: This value was also set to 10.0. This is the frequency at which the map will be published on the display.

The yaw goal tolerance was updated to a value of 0.1. This parameter depicts the tolerance in radians for the controller in yaw/rotation when achieving its goal. The xy goal tolerance was updated to a value of 0.2. This is the tolerance in meters for the controller in the x-y distance when achieving the goal. Both these parameters were doubled to allow for additional flexibility in trajectory planning. The transform tolerance was set to 1.25. Table 3 depicts the move base parameters used.

TABLE 3
move base parameters

move base parameters	
Parameter	Value
yaw goal tolerance	0.1
xy goal tolerance	0.2
obstacle range	1.5
raytrace range	4.0
inflation radius	0.65
robot radius	0.3
update frequency	10.0
publish frequency	10.0

3.3 Personal Model - PramodBot

3.3.1 Model design

Model design: The PramodBot has a similar structure as UdacityBot, but it has a two chassis, square base and is bigger in size. The laser sensor has also moved to the middle of the robot.

Maps: The Clearpath jackal race.yaml and jackal race.pgm packages were used to create the map, similar to the UdacityBot.

Meshes: A hokuyo scanner was simulated as the laser scanner. The hokuyo.dae mesh was used to render it.

Launch: Three launch files were used. They are: 1. robot_description_pramod.launch: This launch file defines the joint_state_publisher which sends fake joint values, robot_state_publisher which sends robot states to tf and robot description which defines and sends the URDF to the parameter server. 2. amcl_pramod.launch: The amcl package relies entirely on the robots odometry and the laser scan data. This file launches the AMCL localization server, the map server, the odometry frame, the move base server and the trajectory planner server. 3. udacity_world_pramod.launch: This is the primary launch file which contains the robot _escription_pramod.launch file, the gazebo world and the AMCL localization server. It also spawns the robot and launches RViz. Fig. 8 depicts the connection graph between the discussed nodes. Fig. 9 depicts the UdacityBot at the goal location and the AMCL particles as seen in RViz.

Worlds: The PramodBot uses the same worlds as the UdacityBot.

URDF: The URDF files defines the shape and size of the robot. Two files were used to define the basic robot description and the gazebo view. 1. pramod bot.gazebo: This file defines the differential drive controller, the camera and the camera controller, the controller for Gazebo and Hokuyo laser scanner. 2. pramod bot.xacro: Defines the robots shape in macro format. pramod bot follows a similar structure as udacity bot except its square in shape and bigger. Table 4 defines the parameters used in pramod bot.xacro.

TABLE 4
PramodBot setup instructions

PramodBot Body		
Part	Geometry	Size
Chassis	Cube	0.4 x 0.2 x 0.1
Back and Front Casters	Sphere	0.0499 (radius)
Left and Right wheels	Cylinders	0.1 (radius), 0.05 (length)
Camera Sensor	Link Origin Shape-size Joint Origin Parent Link Child Link	[0, 0, 0, 0, 0, 0] Box - 0.05 x 0.05 x 0.05 [0.025, 0, 0.025, 0, 0, 0] chassis camera
Hokuyo Sensor	Link Origin Shape-Size Joint Origin Parent Link Child Link	[0, 0, 0, 0, 0, 0] Box - 0.1 x 0.1 x 0.1 [-0.05, 0, 0.1, 0, 0, 0] chassis hokuyo

3.3.2 Packages Used

Same as UdacityBot.

3.3.3 Parameters

The AMCL parameters were kept the same as UdacityBot, but significant changes were made in the move base parameters. The move base parameters were tuned as follows: The obstacle range parameter was modified to have a greater value of 5.0. This parameter depicts the default maximum distance from the robot (in meters) at which anobstacle will be added to the cost-map. This was done in a trial-and-error method. The basic intuition behind increasing this parameter was as the PramodBot was larger in size, it moved slower. So additional time was wasted if it was unable to see an obstacle in moderate range and followed the path towards it only to find the region bounded by an obstacle. Hence, in order to increase its sight with respect to the cost-map, a higher value of this parameter was used. The raytrace range parameter was modified to a higher value of 4.0. This parameter is used to clear and update the free space in the cost-map as the robot moves.

Two parameters in the global and local cost-maps were also changed. They are:

1. update frequency: This value was set to 10.0. This is the frequency in Hz for the map to be updated.

2. publish frequency: This value was also set to 5.0. This is the frequency at which the map will be published on the display

The yaw goal tolerance was updated to a value of 0.1. This parameter depicts the tolerance in radians for the

controller in yaw/rotation when achieving its goal. The xy goal tolerance was updated to a value of 0.05. This is the tolerance in meters for the controller in the x-y distance when achieving the goal. Both these parameters were doubled to allow for additional flexibility in trajectory planning. The transform tolerance was set to 0.2. Table 5 depicts the move base parameters used.

TABLE 5
move base parameters for PramodBot

move base parameters	
Parameter	Value
yaw goal tolerance	0.01
xy goal tolerance	0.05
obstacle range	1.5
raytrace range	4.0
inflation radius	0.6
robot radius	0.3
update frequency	10.0
publish frequency	5.0

4 Results

4.1 Localization Results

4.1.1 Benchmark - UdacityBot:

The time taken for the particle filters to converge was around 5-8 seconds. The UdacityBot reaches the goal within approximately two minutes. So the localization results are pretty decent considering the time taken for the localization and reaching the goal. However, it does not follow a smooth path for reaching the goal. Initially, the robot heads towards the north as it was unable to add the obstacle over there in its local cost-map and hence it followed the shortest path to the goal. But soon, it discovered the presence of the obstacle and it changed its strategy to the next shortest route to reach the goal, i.e. head south-east, turn around where the obstacle ends and reach the goal.

4.1.2 Student - PramodBot:

The time taken for the particle filters to converge was around 35-45 seconds. The PramodBot reaches the goal within approximately 20-30 minutes. So here, a deterioration of the results is observed. This can be attributed to the heavier mass of the PramodBot, even though no wheel slippage was kept for both the robots in their respective URDFs.

4.2 Technical Comparison

PramodBot was significantly heavier than the UdacityBot. Pramodot has a longer shape and also the Laser finder was moved in the middle.

5 Discussion

UdacityBot and PramodBot performed almost same. This might be attributed to the fact that PramodBot is considerably heavier than UdacityBot because it have two chassis. This significantly changed PramodBots speed and hence the time it takes to reach the goal also increased but instead of this it is almost same as UdacityBot. PramodBot, which occupies more space than UdacityBot also had a problem navigating with a higher inflation radius as it thought the obstacles to be thicker than they really are and hence deducing that it has not enough space to navigate.So, when the inflation radius was decreased, the performance of PramodBot improved considerably. The obstacle range was also another important parameter to tune. PramodBot, being slower in speed naturally wastes a lot of time if it goes to the wrong direction some times only to find an obstacle there. So in order to be able to successfully add an obstacle to its cost-map from a significantly larger distance, this parameter was increased which resulted in significant improvement of PramodBots performance.

5.1 Topics

- performed same as UdacityBot.
- One might infer the better performance of UdacityBot due to its smaller mass.
- In the Kidnapped-robot problem, one has to successfully account for the scenario that at any point intime, the robot might be kidnapped and placed in a totally different position of the world.
- In any scenario, where the world map is known, but the position of the robot is not, localization can be successfully used.
- MCL/AMCL can work well in any industry domain where the robots path is guided by clear obstacles and where the robot is supposed to reach the goal state from anywhere in the map. The ground also needs to be flat and obstacle free, particularly in the cases where the laser range-finder is at a higher position and cannot detect objects on the ground.

6 Conclusion / Future work

Both robots satisfied the two conditions, i.e. Both were successfully localized with the help of AMCL algorithm and both reached the goal state within a decent timelimit. UdacityBot performed better than PramodBot which can be attributed to the heavier mass of PramodBot. This time difference was reduced significantly by increasing the obstacle range and inflation radius parameters. The main issue with both the robots was erroneous navigation as both of the robots took a sub-optimal route due to the lack of knowledge of an obstacle beforehand. However, as the primary focus of this project was localization, this problem was ignored due to time constraint.

Even though both the robots reached the goal eventually, as they failed to take the optimal route to the goal location, this implies that neither of these robots can be deployed in commercial products.

Placement of the laser scanner can also play a vital role in the robots navigation skill. Placing the scanner too high from the ground could result in the robot missing obstacles in the ground and then get stuck on it causing a total sensor malfunction. On the other hand, placing the scanner too low may prevent the robot from perceiving better as it gets more viewing range in this situation. So this would be another important modification to improve the robots. Future work would involve making both the robots commercially viable by working on making/tuning a better navigation planner.

Also, the present model deployed only has one robot in a world at a time. In future, this project could be expanded to include multiple robots in the same world, each with the same goal or a different one.

6.1 Hardware Deployment

1) This present model was done in a simulation environment in a local computer in VM hosting Ubuntu 16.04 on a core-i7 machine. In order to deploy it in a hardware system like NVIDIA Jetson TX2, the TX2 prototype board camera could be connected into the model with suitable drivers. Laser range-finder hardwares would also need to be integrated in order the hardware version to successfully operate and detect obstacles. It would also need GIO connections for driving the wheels and be implemented on a suitable platform modeling the robot which was simulated.
2) The JETSON TX2 has sufficient has adequate processing power both in CPU and GPU memory to successfully host this model.

REFERENCES

[1] L. Lamport, *LATEX: a document preparation system: user's guide and reference manual*. Addison-wesley, 1994.