Robot Localization Using ROS AMCL Package

Mustafa Eldeeb

Abstract—In this project, the aim is to utilize ROS packages to accurately localize a mobile robot inside a provided map inside a provided map in the Gazebo and RViz simulation environments. The overview of the tasks can are as follows, building two mobile robots for simulated tasks, creating a ROS package that utilizes AMCL package, exploring specific parameters.

Index Terms—Robot, IEEEtran, Udacity, LaTeX, Localization Kalman Filter, Particle Filter, Adaptive Monte Carlo Localization.

1 Introduction

ONEof the most integral, yet challenging components of a mobile robot is the ability to navigate through space. In order to successfully navigate, a robot must accomplish 4 sequential tasks. It must first perceive the world around it using sensors and then interpret sensor data to draw meaningful information. Second, the robot must be able to efficiently localize itself, or determine its position and pose within a mapped environment. The third and fourth tasks are cognition and and motor control – the robot must decide its next action and ultimately actuate movement in the desired direction.

Localization has received much attention over the past decade, and will be the subject of this paper. There are 3 types of localization problems: local localization, in which the robot knows its initial pose and must determine its position as it moves around the environment; global localization , in which the robots initial pose is unknown, and the robot must determine its pose relative to the ground truth map, and the ”kidnapped robot problem”, which is similar to global localization, except that the robot may be ”kidnapped” and set to a new location on the map at any point.

Noise from the sensors and the actuators makes it harder to estimate the robot position. There is a need for the location algorithms, to decrease the uncertainty caused by the noise. The most common algorithms are the Extended Kalman Filter Localization Algorithm (EKF) and the Monte Carlo Localization Algorithm (MCL).In this project , through the use of MCL as well as sensor fusion, by which inputs from multiple sensors are taken into account, sensor noise will be minimized to solve the localization problem

2 Background

The problem is to localize the robot, as mentioned earlier. The map is known; however, the sensors are not precise. There is noise to affect the measurements coming from the sensors, and the controls are not perfect to move the vehicle to the desired location. Therefore, we need to apply some techniques to use noise measurements. [**?**]

2.1 Kalman Filters

Kalman Filters use noisy measurements and the actuation commands to estimate the location of the robot. There are two steps in Kalman Filters. They are predict and update. Predict step forecast the position after actuation command is applied, whereas the update step uses the measurements to update the location. Kalman filters need to have Gaussian distributions to work. Therefore only linear operations are allowed on the probability. Since the robot model is not entirely linear, there is a need for transforming nonlinear equations to linear equations. Taylor Series is the method for linear approximation. With linearization, the Kalman Filter become Extended Kalman Filter.

2.2 Particle Filters

Particle Filter, also known as Monte Carlo Localization, starts with randomly generating particles. Each particle represents a possible location for the robot. The steps are similar to the Kalman Filter. After the motion, the position of each particle updates, and the uncertainty increases. After the measurements come from the sensors, the algorithm performs another update and the uncertainty decreases.

2.3 Comparison / Contrast

Particle Filter can work with any distribution whereas Kalman Filter only works with a Gaussian distribution. Particle Filter takes raw measurements; however, Kalman Filter requires landmarks. The posterior is particles for Particle Filter and Gaussian for Kalman Filter. Kalman Filter is more efficient and provides more resolution; on the other hand Particle Filter is easier to implement and more robust. Only Particle Filter has Memory and Resolution control and can provide a solution for Glocal Localization. Particle Filter has Multimodal Discrete state space, and Kalman Filter has unimodal continuous

3 Simulations

The simulation environment consists of ROS (Kinetic), Gazebo and RViz. The ROS packages are AMCL, move base, map server, and navigation. There are two different robot models to test on the simulation.

3.1 Achievements

The benchmark model and our model were able to successfully reach the goal position. In both cases, the terminal output confirmed that the robot had reached it’s goal.

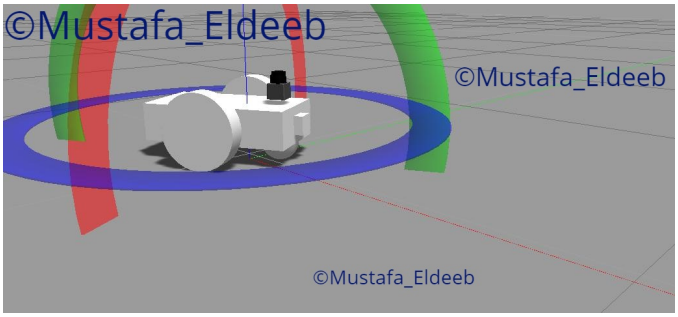

Fig. 1. Udacity bot in gazebo.

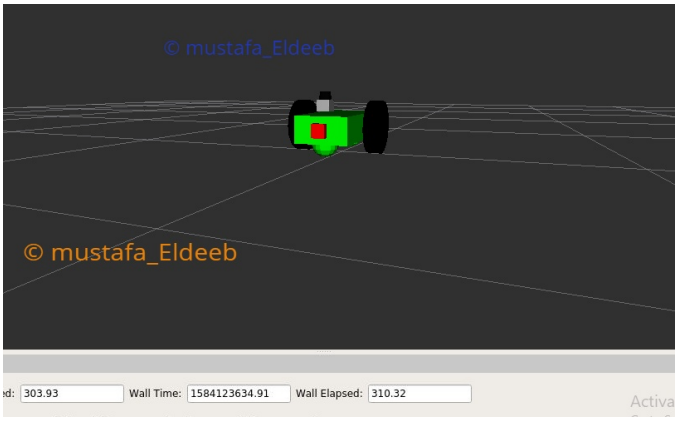

Fig. 2. Udacity bot in RViz.

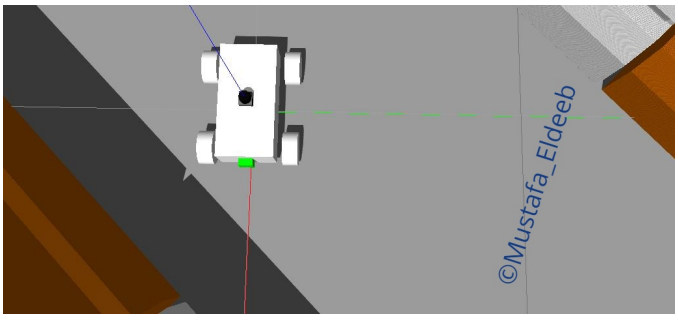

Fig. 3. Deeb bot in gazebo.

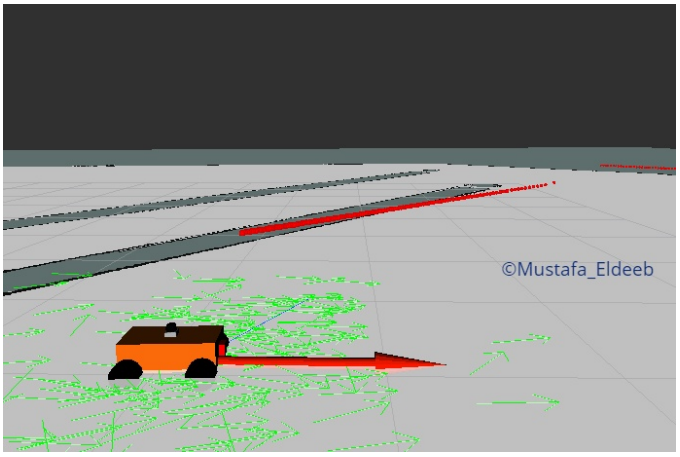

Fig. 4. Deeb bot in RViz.

During the course of navigation, both robots were fairly accurate in following the global path, and there were no major deviations.

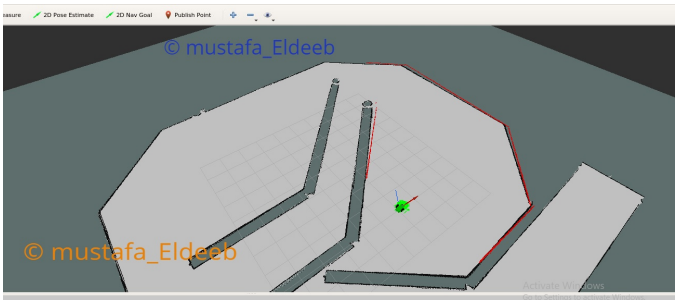

Fig. 5. Udacity bot riched goal.

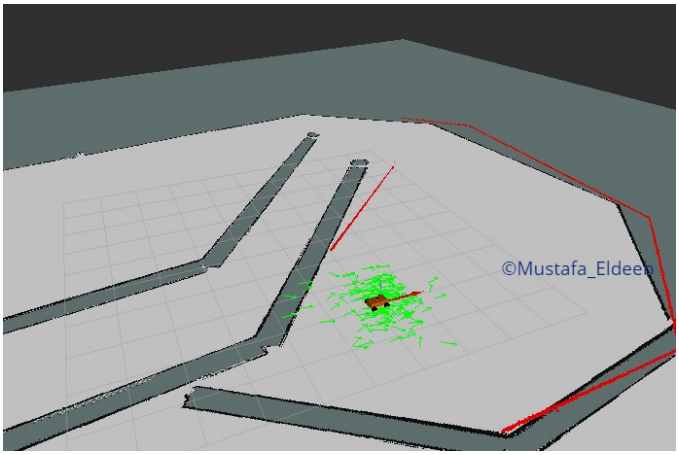

Fig. 6. Deeb bot riched goal.

3.2 Benchmark Model

3.2.1 Model design

The main part of the Udacity bot is the chassis. It has a box shape with dimensions 0.4 x 0.2 x 0.1. It has two casters, one at -0.15 0 -0.05 and the other at 0.15 0 -0.05. There are two wheels connected to the chassis at 0 -0.15 0 and 0 +0.15 0. The laser sensor located at position relative to the chassis 0.15 0 0.085. It has a cubic shape with length 0.1. The camera sensor is connected to the chassis at 0.2 0 0, and it has a cubic shape with length 0.05.Gazebo plugin is Differential Drive Controller. The difference between the Deeb bot and the Udacity bot is the chassis has a box shape with dimensions .4 .2 .125.There are four wheels connected to the chassis at 0.15 0.15 0 and 0.15 -0.15 0 and -0.15 0.15 0 and -0.15 -0.15 0. With wheel radius of 0.06. There is no caster wheels. The laser sensor located at position relative to the chassis 0 0 0.125.Gazebo plugin is skid steer drive controller.

3.2.2 Packages Used

- ros-kinetic-navigation
- ros-kinetic-map-server
- ros-kinetic-move-base
- ros-kinetic-amcl

3.2.3 Parameters

Parameters used to fine tune ROS navigation stack and localization performance are divided into 5 yaml formatted configuration files:

1) **amcl.yaml** The amcl package 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. **Overall filter parameters: min particles:** Minimum allowed number of particles. **max particles:** Maximum allowed number of particles. Particle representations can approximate a wide array of distributions,but the number of particles needed to attain a desired accuracy can be large. Min/Max number needs to be adjusted to get the required accuracy without exceeding the available computing capability in the robot system. High number of particles will cause the system to miss publishing cycles. **transform tolerance:** Time with which to post-date the transform that is published, to indicate that this transform is valid into the future. this parameter is also dependent on your system specifications. This helps decide the longevity of the transform(s) being published for localization purposes. A good value should only be to account for any lags in the system. **initial pose x (y) (a):** Initial pose mean (x,y,yaw), used to initialize filter with Gaussian distribution. **Laser model parameters:** laser model type: likelihood field model is widely used. While it lacks a strict probabilistic interpretation, it tends to be more reliable, especially in environments with small obstacles. Also, the likelihood field model is more computational efficient than the beam model, which requires ray-tracing at run time (the likelihood field model pre-computes the field at startup, so that the run time work is simply a table lookup). Whichever mixture weights are in use should sum to 1, the likelihood field model uses only 2: z hit and z rand [4] **laser z hit:** Mixture weight for the z hit part of the model. **laser z rand:** Mixture weight for the z rand part of the model. **laser max beams:**How many evenly-spaced beams in each scan to be used when updating the filter. **Odometry model parameters:** odom frame id:Which frame to use for odometry. odom model type:Which model to use, either ”diff”, ”omni”, ”diff-corrected” or ”omni- corrected”.If odom model type is ”diff” then sample motion model odometry algorithm is used; this model uses the noise parameters odom alpha 1 through odom alpha4 **odom alpha1..4:** Parameters odom alpha 1 through odom alpha4 specifies the expected noise in odometry’s rotation estimate. used for ”diff” and ”omni” odom model **odom alpha5:**Translation-related noise parameter (only used if model is ”omni”) captures the tendency of the robot to translate (without rotating) perpendicular to the observed direction of travel.

   - The number of particles chosen to be between 15 and 200 to suit the computational power of the system
   - The initial pose was set to (0, 0, 0).
   - we set laser model type to be likelihood-field.
   - we set laser-z-hit to be 0.997.
   - we set laser-z-rand to be 0.003.
   - we set laser-z-rand to be 0.003.
   - we set laser-max-beams to be 90.

   The odom parameters were set by trial and error to be number (0.2). These parameters specify the expected noise in odometry from different components of the robot’s motion. Since in this simulation environment the expected noise was very low, this approach seemed to work well.

```
# Overall filter parameters
base_frame_id:    robot_footprint
global_frame_id:  map
min_particles:    15
max_particles:    250
initial_pose_x:   0.0
initial_pose_y:   0.0
initial_pose_a:   0.0

# Laser model parameters
laser_model_type: likelihood_field
laser_z_hit:      0.997
laser_z_rand:     0.003
laser_max_beams:  90

# Odometry model parameters
odom_frame_id:    odom
odom_model_type:  diff #-corrected
odom_alpha1:      0.2
odom_alpha2:      0.2
odom_alpha3:      0.2
odom_alpha4:      0.2
```

Fig. 7. Amcl param.

2) **base local planner params.yaml:**move base package creates and calculates a path or a trajectory to the goal position, and navigates the robot along that path. The set of parameters in this configuration file customize this particular behavior. **holonomic robot:** Determines whether velocity commands are generated for a holonomic or non- holonomic robot. For holonomic robots, strafing velocity commands may be issued to the base. For non-holonomic robots, no strafing velocity commands will be issued. **meter scoring:** Whether the gdist scale and pdist scale parameters should assume that goal distance and path distance are expressed in units of meters or cells. yaw goal tolerance: The tolerance in radians for the controller in yaw/rotation when achieving its goal. **xy goal tolerance:** The tolerance in meters for the controller in the x and y distance when achieving a goal. **occdist scale:** The weighting for how much the controller should attempt

to avoid obstacles. **sim time:** The amount of time to forward-simulate trajectories in seconds. **pdist scale:**The weighting for how much the controller should stay close to the path it was given, maximal possible value is 5.0.

```
controller_frequency: 10.0

TrajectoryPlannerROS:
 holonomic_robot:       false   #Since differential drive robots are nonholonomic,
                                #the holonomic robot parameter was left to be false.
 meter_scoring:         true
 yaw_goal_tolerance:    0.10 #The tolerancein radians for the controller in yaw/rotation
                             # when achieving its goal.


 xy_goal_tolerance:     0.10 #The tolerance in meters for the controller in the x
                              #and y distance when  achieving  a  goal.

 occdist_scale:         0.05
 sim_time:              1.0
 pdist_scale:           0.6
```

Fig. 8. base local planner params.

3) **costmap common params.yaml** this file is for the list of parameters common to both types of **costmaps;** it defines which sensor is the source for observations. thats the laser sensor in this project case. Aside from that, there are a few parameters that are also required as listed below. **map type:** What map type to use. ”voxel” or ”costmap” are the supported types, with the difference between them being a 3D-view of the world vs. a 2D-view of the world. **obstacle range:** The default maximum distance from the robot at which an obstacle will be inserted into the cost map in meters. This can be over-ridden on a per-sensor basis. **raytrace range:** The default range in meters at which to raytrace out obstacles from the map using sensor data. This can be over-ridden on a per-sensor basis. **transform tolerance:** Specifies the delay in transform (tf) data that is tolerable in seconds. This parameter serves as a safeguard to losing a link in the tf tree while still allowing an amount of latency the user is comfortable with to exist in the system. For example, a transform being 0.2 seconds out-of-date may be tolerable, but atransform being 8 seconds out of date is not. If the tf transform between the coordinate frames specified by the global frame and robot base frame parameters is transform tolerance seconds older than ros::Time::now(), then the navigation stack will stop the robot. In . footprint: Specification for the footprint of the robot. **inflation radius:**. **observation sources:** A list of observation source names separated by spaces. We make a safety distance of 0.6 m to be considered between detected walls and robot path. **laser scan sensor:** Name of the laser scanner observation source.

```
map_type:             costmap
obstacle_range:       2.5
raytrace_range:       3
transform_tolerance:  0.2
footprint:            [[0.2, 0.2], [0.2, -0.2], [-0.2, -0.2], [-0.2,0.2]]
#robot_radius:        0.0
inflation_radius:     0.6
observation_sources:  laser_scan_sensor
laser_scan_sensor: {sensor_frame: hokuyo, data_type: LaserScan, topic: /
udacity_bot/laser/scan, marking: true, clearing: true}
```

Fig. 9. costmap common params.

4) **global costmap params.yaml** This file consists of parameters that specify the behavior associated with local (or global) costmap. The local costmap relies on odom as a global frame since it updates as the robot moves forward. Since the costmap updates itself at specific intervals, and aims to cover a specific region around the robot it requires its own updating and publishing frequencies, as well as dimensions for the costmap. a similar set of parameters is used for the local costmap.global frame: The global frame for the costmap to operate in. **robot base frame:** The name of the frame for the base link of the robot. **update frequency:** The frequency in Hz for the map to be updated. In both robots frequency of higher than 10 was causing errors of missed cycles. **publish frequency:** The frequency in Hz for the map to be publish display information. In both robots frequency of higher than 10 was causing errors of missed cycles. **width:** The width of the map in meters. **height:** The height of the map in meters. **resolution:** The resolution of the map in meters/cell. **static map:** The costmap has the option of being initialized from a user-generated static map. If this option is selected, the costmap makes a service call to the map server to obtain this map. **rolling window:** Whether or not to use a rolling window version of the costmap. If the static map parameter is set to true, this parameter must be set to false.

```
global_costmap:
   global_frame:       map
   robot_base_frame:   robot_footprint
   update_frequency:   10.0
   publish_frequency:  10.0
   width:              50.0
   height:             50.0
   resolution:         0.02
   static_map:         true
   rolling_window:     false
```

Fig. 10. global costmap params.

5) **local costmap params.yaml** Same list of global costmap parameters as described above are used for the local costmap.

```
local_costmap:
   global_frame:       odom
   robot_base_frame:   robot_footprint
   update_frequency:   10.0
   publish_frequency:  10.0
   width:              4.0
   height:             4.0
   resolution:         0.02
   static_map:         false
   rolling_window:     true
```

Fig. 11. local costmap params.

4 Results

This two images will present the results of the two simulated robots:

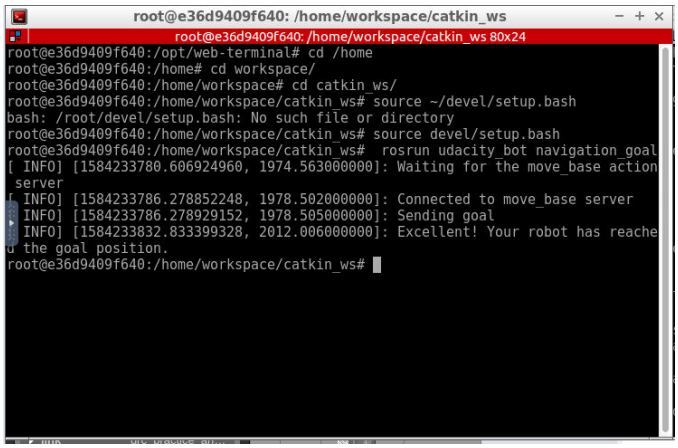

Fig. 12. udacity bot time to goal

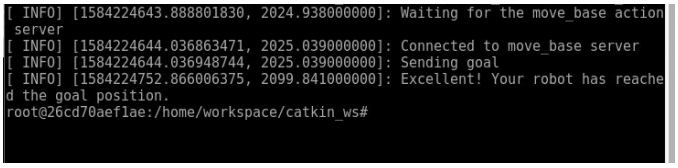

Fig. 13. Deeb bot time to goal.

5 Discussion

5.1 Topics

- Which robot performed better? Considering the time required to reach goal and the trajectory taken by both robots, benchmark robot performance was better.
- Why it performed better? as deeb bot need to be tuned and redesigned to be compatible with the skid steer drive controller plugin
- How would you approach the 'Kidnapped Robot' problem? In 'kidnapped robot' problem a well- localized robot is tele-ported to some other place without being told. This problem differs from the global localization problem in that the robot might firmly believe itself to be somewhere else at the time of the kidnapping. The kidnapped robot problem is often used to test a robots ability to recover from catastrophic localization failures. The regular MCL algorithm with small number of particles is unfit for the kidnapped robot problem, since there might be no surviving samples nearby the robots new pose after it has been kidnapped. The solution is either by increasing the number of samples which will require higher computing power from the robot system or using a modified more effective algorithm such as mixed-MCL
- What types of scenario could localization be performed? performing inspection tasks in industrial area, house cleaning robot, garden mowing robot, and retail warehouse robots.
- Where would you use MCL/AMCL in an industry domain? Mobile base robots in industrial manufacturing where manipulators are able to move performing all sorts of tasks with high accuracy, such as welding, painting, cutting.

6 Conclusion / Future work

Both robots were able to reach the required goals using two different designs, parameters can be further improved in both cases to get better results or make the robot ready for certain tasks/types of obstacles. Next section will explain few suggested improvements

6.1 Modifications for Improvement

The robot perfor- mance can be improved by applying some of the following steps.

- **Base Dimension**: Robot base design can be further improved to simulate a more realistic design; mesh files can be used to show the actual shape of the hardware that will be used, base can be decided based on number and locations of the required sensors plus the targeted tasks that will be done by the robot
- **Sensor Location :** Based on expected tasks that will be done by robot and the hight/range of obstacles, sensors locations can be further improved
- **Sensor Layout:** Having two LiDARs with 180 degrees of visibility each can cover 360 degrees around the robot which will improve over all localization accuracy. Another way is to have one 360 degrees LiDAR in the middle of the robot base to cover the full range
- **Sensor Amount:** Additional types/number of sensors can also improve the robot performance and make it more robust to handle kidnapped robot situations.

6.2 Hardware Deployment

An exciting area for future work could be in hardware deployment of these models. The Jetson TX2 board with ROS installed would be a great option to explore.