Project: Where Am I!?

Abstract

Localization is one key topic of general mobile robot navigation task. In this project, we would like to simulate a mobile robot identifying its location in the virtual environment and then traveling to a designated target location. The simulation was conducted in Gazebo and RViz environment and the robot was implemented with the Adaptive Monte Carlo Localization method for navigation. In the end, the robot could successfully reach the target location in the pre-determined environment.

Introduction

Navigation is one of the challenging topics of modern mobile robots. To successfully navigate through space to the desired target location, a robot must process the following capabilities. First, perception. Since the world is not perfect, there are always uncertainty for real world tasks. A robot must be able to perceive and process sensor data to gather information about the world surrounding itself. Beyond that, secondly, the robot would need to use the information to understand its own position with respect to the environment (localization). Third, the robot will then need to adjust its action based on its pose and the environment information. With all these three functions, the robot could ultimately travel to the target destination.

Localization is a crucial part in robot navigation problems. In general, it has been categorized into three variations: Position tracking (also called local localization), global localization, and the kidnapped robot problem. In the local localization problem, the robot knows its initial pose and will need to determine its position while travel around the environment. In contrast, in the global localization problem, the robot's initial pose is unknown, and it must determine its pose relative to the ground trust map before it could successfully travel to the target. Finally, the kidnapped robot problem, which is more challenging compared to the global localization problem, the robot might be "kidnapped" at any time during the tasks and be moved to any place in the environment. Besides these differences, the environment could be either static or dynamic during the robot's action. The later is much closer to the real-world scenario but is also more difficult and beyond the scope in this project.

Background / Formulation

To solve localization problem, there are several algorithms have been developed in recent decades. Two most popular localization algorithms are the Extended Kalman Filter Localization Algorithm (EKF) and the Monte Carlo Localization Algorithm (MCL).

<u>Kalman Filter (KF) and Extended Kalman Filter</u>

The Kalman Filter (KF) is a very powerful algorithm, which takes account the uncertainty of each sensor's measurement but does not require a lot of data to make an accurate estimation. In the beginning of the algorithm, the Kalman filter make an initial guess of the state variables, which could be position, speed of the robot. Then the algorithm continues and repeats the following two steps: measurement update and state prediction. In measurement update, the collected sensor data and prior states were combined to generate the new states based on their Gaussian distribution. The new states estimate is more confident than both the prior belief and the measurement since two Gaussian offer more data than one alone. Next, the robot executes an action command such as a move to a certain distance. The state variables are then updated with the believed motion, which is also described as Gaussian distribution, and are called as posterior state prediction. Then the KF is ready for the next iteration of the measurement update.

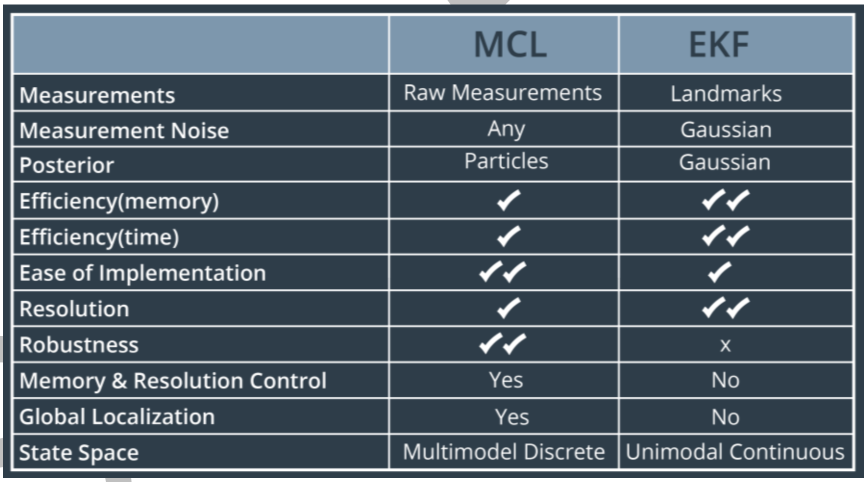
Standard Kalman filter (KF) algorithm can be applied to linear systems. However, when facing the non-linear real-world scenarios, the Extended Kalman Filter (EKF) is developed to apply linearization to a non-linear function using the Taylor series.

<u>Monte Carlo Localization Algorithm (MCL)</u>

In contrast to Kalman filter which using one main robot to perceive all sensor and action inputs, the Monte Carlo Location Algorithm (MCL) places many imaginary robots/particles in the environment to represent guesses of where the real robot might be located. Each particle has its position, orientation, and a weight component to represent the difference between the robot's actual and predicted pose. In the beginning of the MCL algorithm, the particles are spread uniformly throughout the entire space. As the algorithm progress, the particles' predicted pose and weight will be updated continuously each time the robot move or sense the environment. Then these particles are re-sampled according to the weight. High weight indicates more likely the particle represents the robot pose. Particles will low weight will be eliminated through the re-sample process. The number of particles is dynamically adjusted over a period and converse to the highest believed robot pose.

Comparison between KF/EKF and MCL

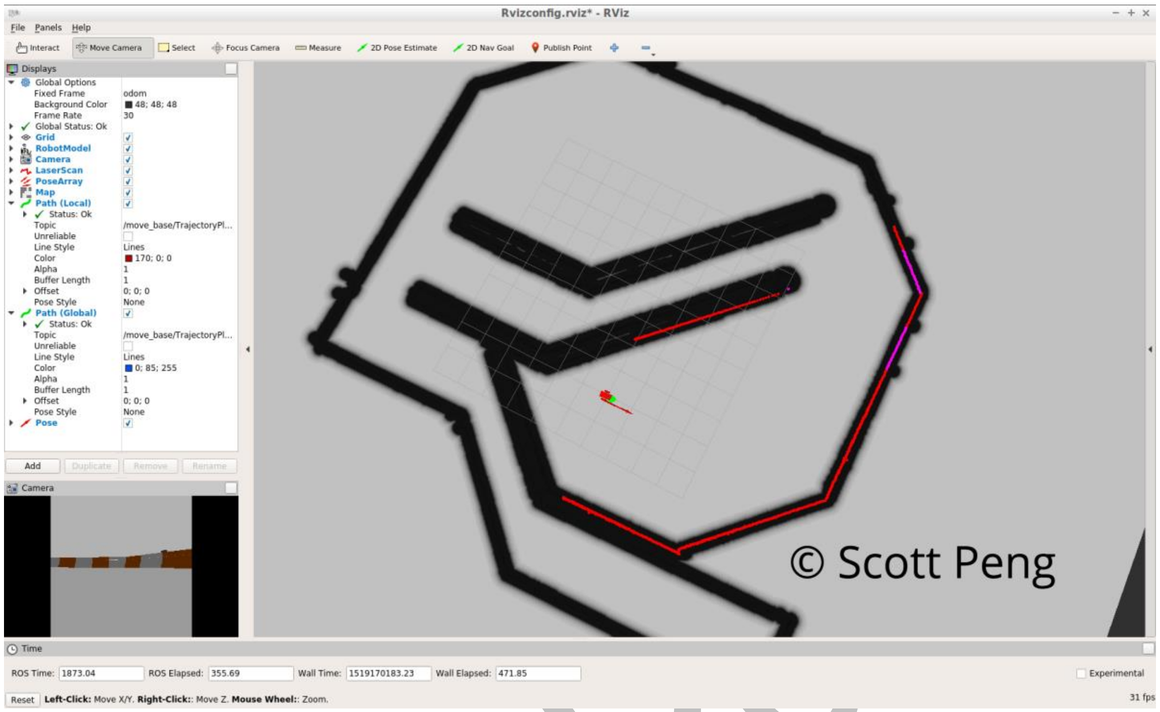
Although Kalman filter is an elegant and powerful localization algorithm, and can provide accurate estimate of the robot state, it has the fundamental assumption of linear and Gaussian distribution of noise, and ultimately represents posteriors as Gaussians. This nature restriction makes Kalman filter difficult and inapplicable to global localization problems, and so to many real-world problems. Comparing to EKF, MCL has the fundamental advantage in handling non-linear and non-Gaussian noise. For software standpoint, MCL is in general much simpler to implement and easier to control memory and resolution. In this project, we will be using MCL as our localization tool.
The table below provides a full comparison of the two algorithms.



	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

Simulation Environment

This simulation is conducted in a Gazebo environment with RViz for visualization of the mobile robot and virtual world. The following sections will describe simulation project formation, packaged used, the virtual world for simulation, mobile robot design and the parameters used to configure the robot performance.



```
Global Options
  Fixed Frame         odom
  Background Color    48; 48; 48
  Frame Rate          30
Global Status: Ok
Grid                  ✓
RobotModel            ✓
Camera                ✓
LaserScan             ✓
PoseArray             ✓
Map                   ✓
Path (Local)          ✓
  Status: Ok
  Topic               /move_base/TrajectoryPl...
  Unreliable
  Line Style          Lines
  Color               170; 0; 0
  Alpha               1
  Buffer Length       1
  Offset              0; 0; 0
  Pose Style          None
Path (Global)         ✓
  Status: Ok
  Topic               /move_base/TrajectoryPl...
  Unreliable
  Line Style          Lines
  Color               0; 85; 255
  Alpha               1
  Buffer Length       1
  Offset              0; 0; 0
  Pose Style          None
Pose                  ✓

ROS Time: 1873.04   ROS Elapsed: 355.69   Wall Time: 1519170183.23   Wall Elapsed: 471.85   Experimental
Left-Click: Move X/Y. Right-Click:: Move Z. Mouse Wheel:: Zoom.   31 fps
```

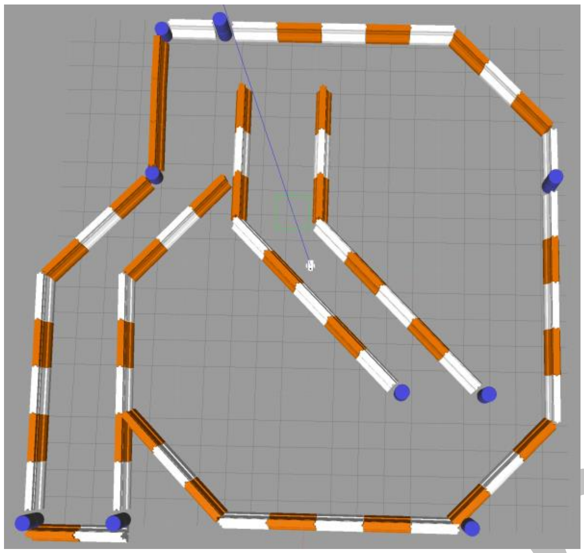
<u>Describe the package formation</u>

Project files are organized in the following folders:

Worlds – this folder contains the files that define the world. The udacity_world file is the blank world with ground plane, a light source, and a camera. The jackal_race world define the maze for navigation test.

URDF – this folder contains the robot description. *Udacity_bot.gazebo* defines the sensor plug-in, hokuyo leaser plug-in and the differential drive controller. *Udacity_bot* defines the physical features of the robot model.

Maps – this folder contains *Jackal_race.yaml* and *jackal_race.pgm* which define the map used in this project.



Meshes – contains the mesh files used in the project. *holuyo.dae* is placed here.

Launch – This folder contains the launch files and associated file. Launch files in ROS can be executed simultaneously to start separated nodes for different software tasks. In this project, there are three launch files are used:

1) Udacity_world.launch – launches the robot_description.launch file and jackal_race.world. It also launches RViz and spawns the robot in Gazebo.
2) Amcl.launch – launches the map server, places the map frame at the odometry frame, launches the acml localization package, the move_base package, and the tf package.
3) Robot_description.launch – launches joint_state publisher, which sends joint values, the robot_state publisher, which sends robot states to tf and the robot_description, which sends the robot URDF to the param server.

In addition to the launch files, RViz configuration file is also placed here for the software to be open

Config – this folder contains the parameters configuration files for simulation performance:

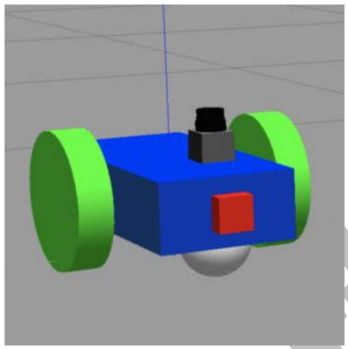
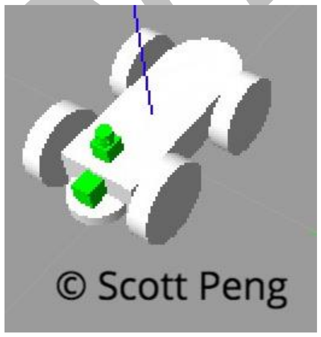
1) *amcl.yaml* – this contains acml filter parameters, laser model parameters, Odometry model parameters.
2) *base_local_planner_params.yaml* – this contains parameters for base_local_planner
3) *costmap_common_params.yaml* – this contains parameters for both global and

local cost maps

4) *costmap_params_template.yaml* – this is a template file containing all parameters for both global and local cost maps
5) *global_costmap_params.yaml* – this contains parameters for global cost maps
6) *local_costmap_params.yaml* – this contains parameters for local cost maps

<u>Robot model</u>

In this project, a standard template robot model (udacity_bot) was provided for the first trial, and then a second robot (my_bot) was created based on the first one. Both robots' shape are shown below:

Udacity_bot	My_bot
	

Each robot consists of a main body and revolute (continue) joints for right-hand and left-hand side wheels. To prevent rolling over or flip backward, the robot also has two casters attached to front and rear bottom.

My_bot robot was designed based on the udacity_bot. The base dimension remains the same. Two additional wheels are added, one to each side. Two semi-disc shape bumpers are added, one in the front and one in the back. The camera and laser sensor locations are slightly changed to accommodate the additional parts.

Model Configuration

Following sections are collections configuration parameters used in this project to adjuct the robot performance. Most of the parameters remain the same for both robot models.

base_local_planner_params:

controller_frequency	10.0
oscillation_reset_dist	0.1
escape_vel	-0.4

- controller_frequency was reduced to 10.0 from 20.0, and oscillation_reset_dist was set to double the size to reduce the computation effort.
- escape_vel was increased to allow the robot turn away more easier when it got stuck.

costmap_common_params

obstacle_range	3.0
raytrace_range	7.0

- obstacle_range and raytrace_range was set to 3.0 and 7.0, respectively, to increase the clearance between planned robot path to the obstacles (wall).

global_costmap_params

update_frequency	10.0
publish_frequency	10.0
width / height	30.0
resolution	0.02

- global costmap size are also reduced 30x30 to avoid extensive computation effort.
- Update and publish frequency was reduced to 10.0 Hz to match the hardware computation requirement.

local_costmap_params

update_frequency	10.0
publish_frequency	10.0
width / height	5.0
resolution	0.02

- local costmap size are also reduced and 5x5 to avoid extensive computation effort.
- Update and publish frequency was reduced to 10.0 Hz to match the hardware computation requirement.

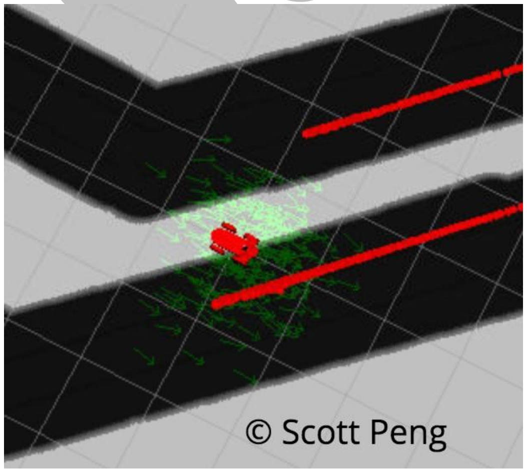
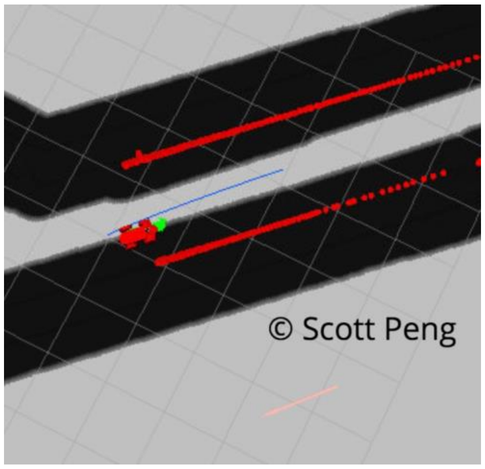
Amcl parameters

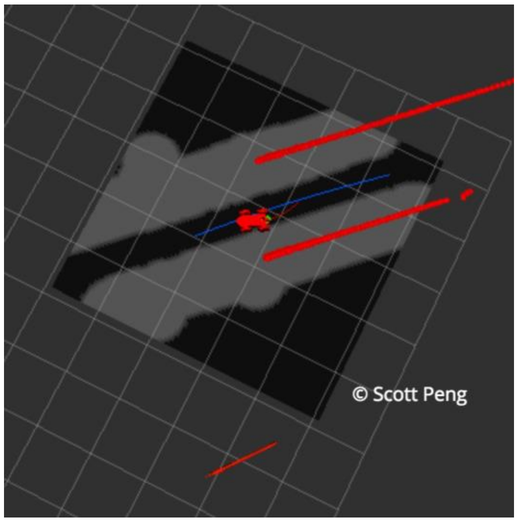
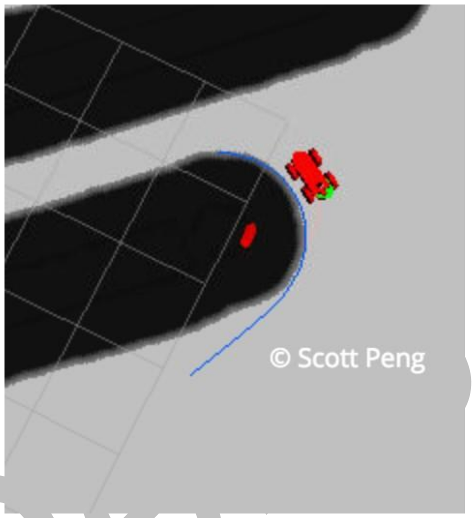
max_particles	200
min_particles	25
transformation_tolerance	0.2
recovery_alpha_slow	0.001
recovery_alpha_fast	0.1
use_map_topic	true
odom_model_type	“diff-corrected”

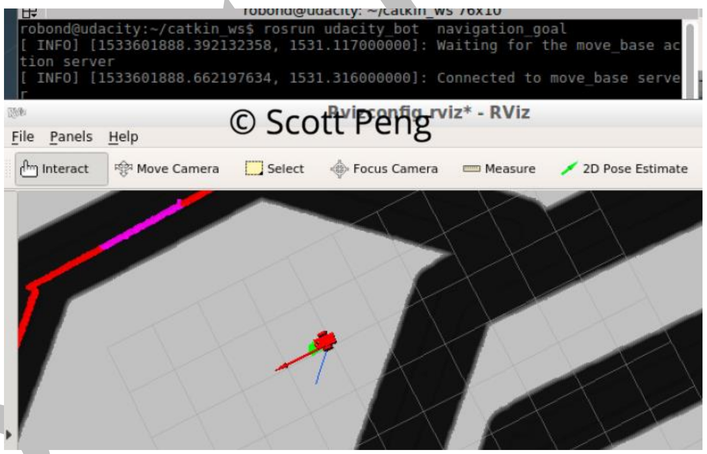
- The transformation tolerance is one important parameter to adjust to match the update frequencies.
- The number of particles was reduced to between 25 and 200 to avoid extensive computation effort.
- The recovery alpha slow and fast values were set based on the recommended values.

Results

After adjusting the configuration parameters, both robots could start movement and converge the particle amount quickly. At the end reach the target destination required by the testing program.

Before providing target location to move, the particles surrounding the robot spread widely.	Start moving. The particle converges fast.
	

Local costmap.	Planned path to navigate through the obstacle.
	

Reach goal in the navigation test.


```
robond@udacity: ~/catkin_ws 76x10
robond@udacity:~/catkin_ws$ rosrun udacity_bot  navigation_goal
[ INFO] [1533601888.392132358, 1531.117000000]: Waiting for the move_base ac
tion server
[ INFO] [1533601888.662197634, 1531.316000000]: Connected to move_base serve
[
```

Discussion

Both robots could reach the target location and the performance were very similar due to the similarity of their design. One important discovery during the project setup is about the balance between robot model design and the configuration. During the process of modifying the robot model, there are numerous trials to change my_bot's shapes and dimensions. Slight variances in the robot's inertial, joints' specification could affect the performance. Same do the placement of camera and laser sensor. It requires considerate patient for every design changes, and requires re-adjust the configuration settings. Especially on the distance to obstacle, escape_vel, etc.…

The original design of my_bot was to add additional wheel to each side of the main body so that they can provide supports to the body weight, and the casters could then be removed. However, the current differential drive control package could not support multiple wheels on each side without modifying the source code. For temporary solution, the robot model was changed back to original design. Only modified the visual part in the urdf file to show ideal robot shape. The robot collision, inertial, etc.… remain the same.

Another thing noticed is that, during testing with random assigned target, the robot rotated several turns before finding out the path and starting to move. Potential reason is that the robot might not be able to solve the local localization problem with the available information, and causes it to give up, spine around and try again. It only happens in a few target locations.

In addition, further adjustment in the current AMCL will be needed when applying to the advanced kidnapped robot problem, in which the robot is placed into random location at any time during the task. AMCL has the ability to dynamically adjust the number of particles in the filter when the environment changes. This enables the robot to make a tradeoff between processing speed and localization accuracy. However, the algorithm has to re-distribute particles and re-initiate the process as the robot was translated to a different location. Since currently the robot has no idea to detect it has been kidnapped, and the reminding particles might likely not be in the current robot location, the algorithm as current will not work well without modification.

AMCL used in this topic provides a very powerful technique to produce autonomous guidance and produce accurate collision free paths with local uncertainties. For real world application, the robot could have additional communication capability using GPS or WIFI to access network and to update the map data. It would be very useful to overcome more complicated tasks such as the kidnapped robot problem.

Conclusion / Future Work

In this project, we successfully implemented a mobile robot simulation environment, tested mobile robot with navigation package based on Monte Carlo Localization Algorithm (MCL/AMCL), and then fine-tuned many configuration settings in the software packages to improve the performance. For next step, there are several options we can try:

- Continue to improve the robot model to have differential drive controller that can control multiple joints on both sides.
- Re-adjust configuration setting with different hardware. We have been lower down the process speed (update frequency to 10) to ensure the result is accurate due to the hardware restriction. In addition, we also have lowered down map size were set to 30 x 30 and 10 x 10 for global and local, respectively, to reduce the computational effort. If we can deploy the package on a better suited hardware, we should be able to improve the accuracy while maintaining reasonable process time.
- Adding additional sensor/camera. Current, only one camera and one laser sensor are used. There would be great potential to improve the performance if additional sensors were used, either to reduce dead-spot of the obstacles or to increase the resolution and confident level of the data received.
- Continue to adjust parameters further to ensure the robot planning fast, smooth and could reach the target in more straightforward manner.
- Study more able the simulation settings to allow more flexible simulation tests. Such as placing the robot in random location when every time launches the package.

This is a successful first step and great foundation for robot navigation. There are endless possibilities for the future work in this playground.