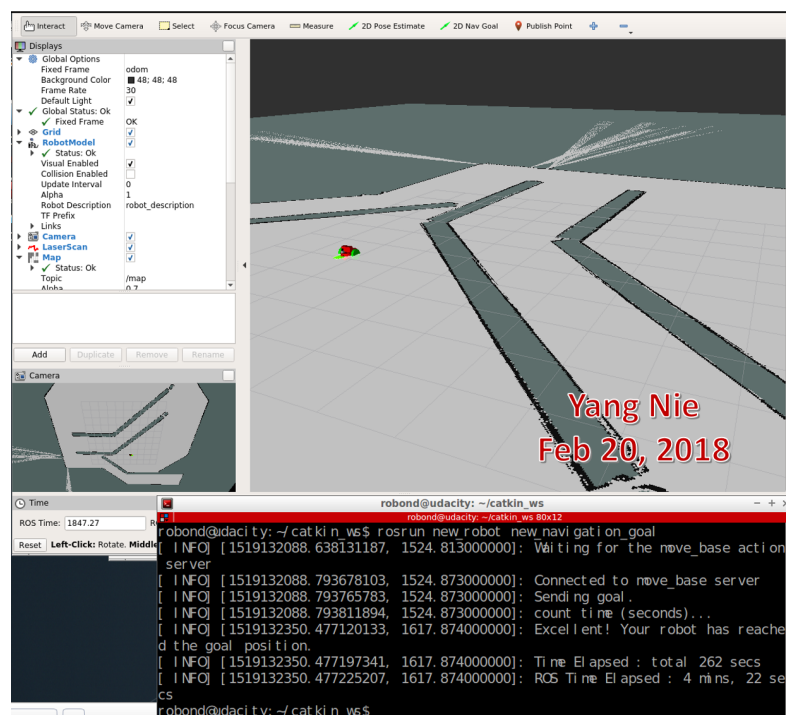# Udacity Robotics Software Engineer Project

Term2 Localization Lab

Abstract

The two robots were created in the project, both robots started from a initial starting point, then utilized ROS packages to accurately localize a mobile robot inside a provided map in the Gazebo and RViz simulation environments. The sensors and AMCL (Adaptive Monte Carlo Localization) algorithm were used to locate current position, and search the path to navigate automatically to a predefined target position. The results for both the Classroom robot and the developed robot will be compared in this article.
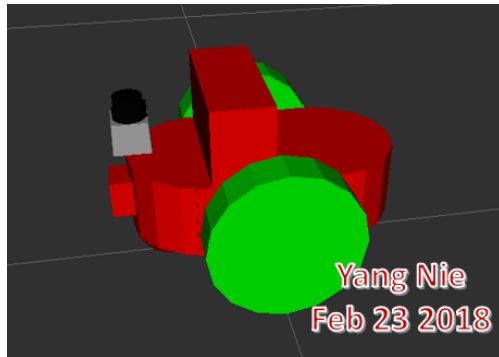
Introduction

The project focuses on the following several aspects of robotics:

- Building a mobile robot for simulated tasks.
- Creating a ROS package that launches a custom robot model in a Gazebo world and utilizes packages like AMCL and the Navigation Stack.
- Exploring, adding, and tuning specific parameters corresponding to each package to achieve the best possible localization results.

The project created two robots:

- One is benchmark robot (called: udacity_bot) given as part of the project,
- The second one (called: new_robot) was created by auther.

Both robots need to use sensors such as a camera or Lidar (Light Detection and Ranging) and AMCL algorithm package.

A predefined maze map was provided, and a C++ navigation goal program was coded to give a navigation goal position.

Background

Consider a robot with an internal map of its environment. When the robot moves around, it needs to know where it is within this map. Determining its location and rotation (more generally, the pose) by using its sensor observations is known as robot localization[4].

Localization involves one question: Where is the robot now? Or, robo-centrically, where am I, keeping in mind that "here" is relative to some landmark (usually the point of origin or the destination) and that you are never lost if you don't care where you are.

Although a simple question, answering it isn't easy, as the answer is different depending on the characteristics of your robot. Localization techniques that work fine for one robot in one environment may not work well or at all in another environment. For example, localizations which work well in an outdoors environment may be useless indoors.

All localization techniques generally provide two basic pieces of information:

- what is the current location of the robot in some environment?
- what is the robot's current orientation in that same environment?

The first could be in the form of Cartesian or Polar coordinates or geographic latitude and longitude. The latter could be a combination of roll, pitch and yaw or a compass heading[8].

The robot performance is related a running environment directly, it is so important which hardware and virtual machine configuration were used in this project.

Hardware:

```
Computer model: Surface Pro 4
```

```
Processor: Intel i7-6650U CPU @ 2.20GHz @2.21GHz
```

```
RAM: 16GB
```

```
Operation System: Window 10 Pro
```

Virual Machine:

```
VMware Workstation 12 Pro, version 12.5.6
```

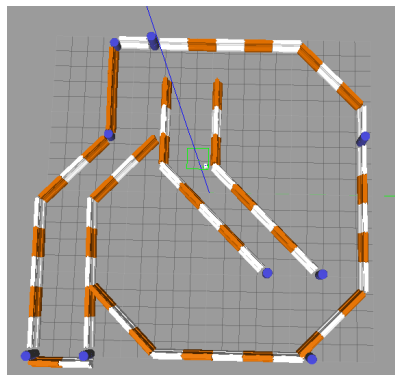```
Processor: 2
```

```
Memory: 8GB
```

```
Hard Disk: 40 GB
```

Software

1. Using an Udacity ROS (Kinetic) package to create a robot simulation environment on VMWare machine. This ROS includes Python (2.7), Gazebo (7.10.0) and RViz (1.12.15) packages.
2. Using URDF (Unified Robot Description Format) to create the robot model which includes pose, inertial, collision and visual data.

   Two sensors - a camera and a laser rangefinder (Hokuyo)[1] was added in this URDF model.
3. A map created by Clearpath Robotics[2] was used for both robots in the project.

4. AMCL (Adaptive Monte Carlo Localization) algorithm was used to dynamically adjust the number of particles over a period of time.
5. A C++ code was used to send a target position to move_base action server.

Kalman Filters and Monte Carlo Simulations are two most common algorithms for robot localization:

1. Kalman Filters and EKF
   The algorithm works in a two-step process. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. The algorithm is recursive. It can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.

Using a Kalman filter does not assume that the errors are Gaussian. However, the filter yields the exact conditional probability estimate in the special case that all errors are Gaussian-distributed.

Extensions and generalizations to the method have also been developed, such as the extended Kalman filter and the unscented Kalman filter which work on nonlinear systems. The underlying model is similar to a hidden Markov model except that the state space of the latent variables is continuous and all latent and observed variables have Gaussian distributions.[4].

2. Monte Carlo Simulations Monte Carlo simulations is an algorithm for robots to localize using a particle filter. Given a map of the environment, the algorithm estimates the position and orientation of a robot as it moves and senses the environment. The algorithm uses a particle filter to represent the distribution of likely states, with each particle representing a possible state, i.e., a hypothesis of where the robot is. The algorithm typically starts with a uniform random distribution of particles over the configuration space, meaning the robot has no information about where it is and assumes it is equally likely to be at any point in space. Whenever the robot moves, it shifts the particles to predict its new state after the movement. Whenever the robot senses something, the particles are resampled based on recursive Bayesian estimation, i.e., how well the actual sensed data correlate with the predicted state. Ultimately, the particles should converge towards the actual position of the robot.[5].

3. Compare Monte Carlo Simulations vs. Extend Kalman Filters

	MCL	EKF
Measurements	Raw Measurements	Landmarks
Measurement Noise	Any	Guassian
Posterior	Particles	Guassian
Efficiency(memory)	OK	Good
Effcency(time)	OK	Good
Ease of implementation	Good	OK
Resolution	OK	Good
Robustness	Good	Poor
Memory & Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multimodel Discrete	Unimodel Continuous

The Challenges[7]

The mobile robot localization problem is to determine the pose (direction and position) of the robot given the map of the environment, sensor data, a sensor error model, move- ment data, and a movement error model. It is a very basic problem of robotics since most of robot tasks require knowl- edge of the position of the robot. There are three types of lo- calization problems in increasing order of difficulty (Thrun, Burgard, and Fox 2005).

Local Position

Tracking The initial pose of the robot is assumed to be known in this type of problem. Since the uncertainties are confined to region near the actual pose, this is considered to be a local problem.

Global Localization

In contrast to local position tracking, global localization assumes no knowledge of initial pose. However, it subsumes the local problem since it uses knowl- edge gained during the process to keep tracking the position. The goal of the MCL algorithm introduced in this assignment is to perform global localization.
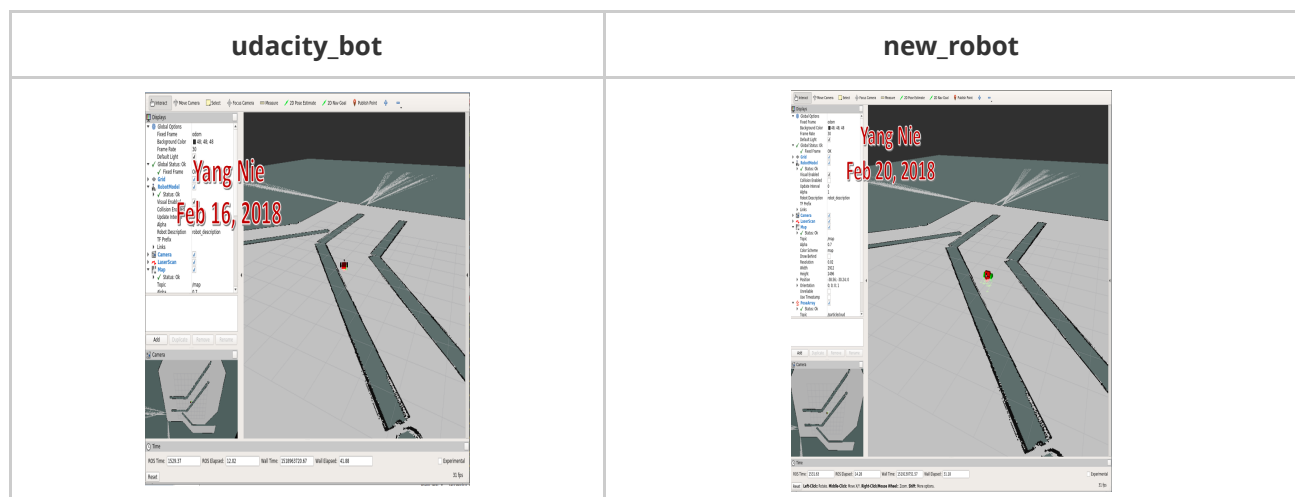
Kidnapped Robot Problem

The kidnapped robot problem arises from the movement of a successfully localized robot to a different unknown position in the environment to see if it can globally localize. Thus it is more difficult than global localization problem since the robot has a strong but wrong belief in where it is. The original MCL algorithm does not have the ability to recover from kidnapping. Such failure is also often referred to as catastrophic failure.
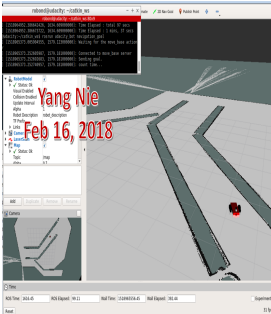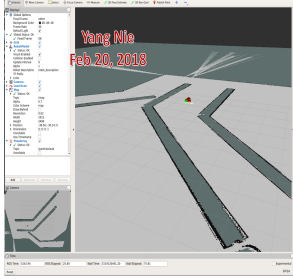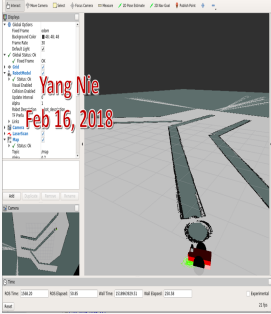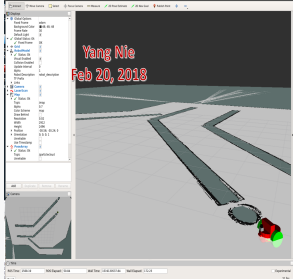
Results

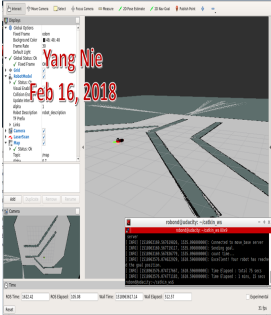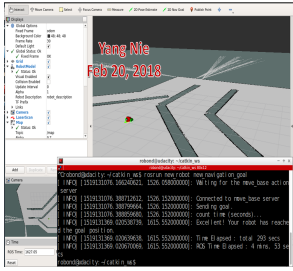## Testing scenario:

Both robots used the same map with same starting (0 0 -0.785) and target (0.995 -2.99 0) position.

udacity_bot	new_robot

Testing results

Both robots navigated in map well and could arrive to the taget position within reasonable time.

	udacity_bot	new_robot
Go straight		
Make a turn		
Arrived target		
Average Time	6 -7 munites	4 -5 munites

The navigation trajectory for both robots is a green line route, the robots arrived to the goal in the end. The problem is that robots need to go up then make a cycle turn first in the map (Figure 1) and this cycle turn routing wasted time. The better navigation approach is followed red line, it goes to the target position directly.

Anther problem is the robot stuck on the wall in several testings. The program needs to restart to solve this issue.
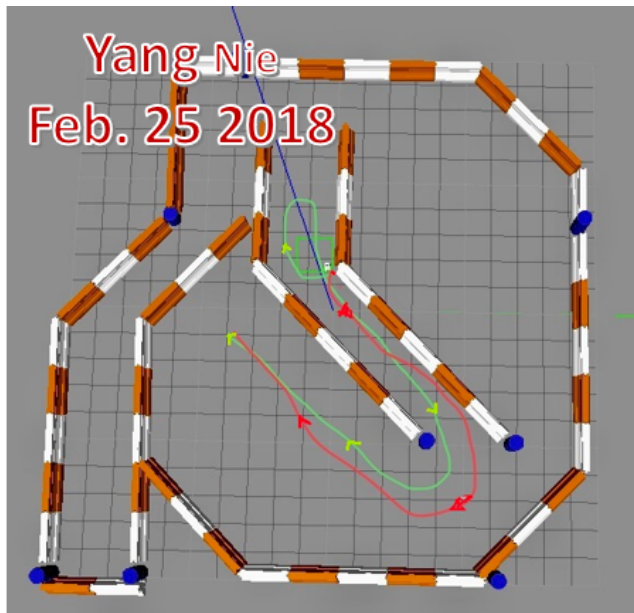
Figure 1.

Model Configuration

These parameters were adjusted in the project to improve the robot performance:

- /amcl/laser_model_type: likelihood_field_prob = (string, default: "likelihood_field") Which model to use, either beam, likelihood_field, or likelihood_field_prob (same as likelihood_field but incorporates the beamskip feature, if enabled)[5].

  Used likelihood_field_prob, make laser sensor has beamskip feature.

- /amcl/max_particles: 240 = (int, default: 5000) Maximum allowed number of particles.

- /amcl/min_particles: 30 = (int, default: 100) Minimum allowed number of particles[5].

  Adjusted these two value lower to reduce CPU usage and improve performance.

- /amcl/resample_interval: 1.0 = (int, default: 2) Number of filter updates required before resampling[5].

  Set a lower value to improve performance.

- /amcl/transform_tolerance: 3.2 = (int, default: 2) Number of filter updates required before resampling[5].

  Set the value higher to improve localization accuracy.

- /move_base/TrajectoryPlannerROS/sim_time: 3.0 = (double, default: 1.0) The amount of time to forward-simulate trajectories in seconds[5].

  Set higher value to speed up robot navigation.

- /move_base/TrajectoryPlannerROS/xy_goal_tolerance: 0.05 = (double, default: 0.10) The tolerance in meters for the controller in the x & y distance when achieving a goal[5].

  Reduce the value to increase the challenge to achiev a goal

- /move_base/controller_frequency: 5.0 = (double, default: 20.0) The frequency at which this controller will be called in Hz. Uses searchParam to read the parameter from parent namespaces if not set in the namespace of the controller. For use with move_base, this means that you only need to set its "controller_frequency" parameter and can safely leave this one unset[5].

  Set the lower value to eliminate the warning message "Control loop missed its desired rate of 20.0000Hz". This parameter doesn't impact robot performance, but it will reduce these unnecessary warning messages on the screen and in the log file.
- /move_base/global_costmap/raytrace_range: 9.0
- /move_base/local_costmap/raytrace_range: 9.0 = (double, default: 3.0) The maximum range in meters at which to raytrace out obstacles from the map using sensor data[5].

  Used higher value to increase sensor detecting obstacles distance
- /move_base/global_costmap/robot_radius: 0.19
- /move_base/local_costmap/robot_radius: 0.19 = (double, default: 0.46) The radius of the robot in meters, this parameter should only be set for circular robots, all others should use the "footprint" parameter[5].

  Set a lower value to match the project robot size.
- /move_base/global_costmap/transform_tolerance: 0.4
- /move_base/local_costmap/transform_tolerance: 0.4 = (double, default: 0.2) 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[5].

  Set a little higher value to increase the system tolerable.

Discussion

- Adjusting the parameter is a big challenge and time consuming job. Those parameters can be changed independently, but they are related eachother. It is impossible that one person tries all possible combination values for all parameters in limited time. A team work needs to assign for achieving the best result.
- AMCL would'n work well for the kidnapped robot problem, when this error: "Clearing costmap to unstuck robot" happened, the robot could be showed up at another new location, the AMCL algorithm couldn't figure out this position, the robot was stuck in the map .
- A moving robot with MCL/AMCl algorithm can be used warehouse industry to move and delivery good inside the warehouse. This job and working environment have clear start and end positions.

Future Work

- Both robots started forward to dead end direction, then turned back to reverse point (Figure 1). The further study needs to involve to find out this is an algorithm issue or parameter turning problem.
- Additional sensor can be added on back of the robot, so the robot can go back and forth without rotating to navigate to the target position.

- Adjusting and trying different parameters are very man power cost work, a database can be built to store these test and result data to help developing a new robot, and use Deep Learning technology to figure out and generate these parameters automatically.

Reference

[1] ClearPathRobotics,"Clearpath robotics home page."https://www.clearpathrobotics.com, 2018.

[2] Hokuyo, "Hokuyo laser scanner home page." https://www.hokuyo-aut.jp, 2018.

[3] Wikipedia, "Kalman filter" https://en.wikipedia.org/wiki/Kalman_filter 2018

[4] Wikipedia, "Monte Carlo localization" https://en.wikipedia.org/wiki/Monte_Carlo_localization 2018

[5] wiki.ROS.ORG, "Documentation" http://wiki.ros.org/ 2018

[6] R. Siegwart, "Mobile Robot Localization" http://www.cs.cmu.edu/~rasc/Download/AMRobots5.pdf 2002

[7] Zuozhi Yang and Todd W. Neller, "A Monte Carlo Localization Assignment Using a Neato Vacuum with ROS" https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/download/15025/13983 2017

[8] wikibooks.org , "Robotics/Navigation/Localization" https://en.m.wikibooks.org/wiki/Robotics/Navigation/Localization