Localization: Where Am I?

Quah Yu Soon

Abstract—This documentation discusses about Kalman Filter and Monte Carlo Localization which are two most common localization algorithms that could effectively tackle local and global localization problem. An experiment that implemented Adaptive MCL on two different robot models would be carried out for localization performance evaluation. The experiment was performed in a simulated environment in Gazebo, with the aid of ROS Navigation Stack which provides capability to navigate robot model towards goal position in automatic behaviour. Tuned parameters in amcl and move_base nodes would also be discussed.

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

1 Introduction

Localization is about identifying robot pose relative to the ground truth map. There are three types of localization problems, ordered from lowest to highest difficulty: Local Localization, Global Localization and Kidnapped Robot problem. In local localization problem which is also known as Position Tracking, initial pose of the robot is approximately known and any robot movements will be kept track of,but current pose of the robot can no longer be determined once it was kidnapped. Whereas in Global localization problem, initial pose is unknown, thus amount of uncertainty is much greater than that in Position Tracking. Kidnapped Robot problem is similar to Global Localization problem except that the robot may be kidnapped and moved to a new location at any time. Kalman filter is good in solving Position Tracking problem, whereas Monte Carlo localization which is also known as particle filter is good in solving in Global Localization problem.

In the experiment, there is a benchmark robot model provided by Udacity and it was named as udacity_bot. Another would be self-developed robot model which provides students an opportunity to demonstrate their skills in building a robot model. Both robot models would be tested in a simulated environment in Gazebo as shown in Figure 1. Each robot model would be navigated towards goal position by following the global path computed by move_base node and localized itself along the path using amcl node. Several parameters in amcl and move_base nodes were tuned to achieve best localization result while experiment was performed.

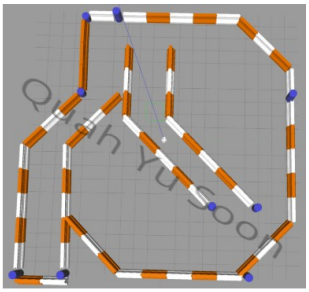

Fig. 1. Simulated map in Gazebo.

2 Background

Robots could perceive surrounding environment through a variety of sensors. However, each sensor contains noise and error. Therefore, good localization algorithms take noisy measurements from at least two sensor devices, compare them ,filter the noise, remove uncertainties and ultimately provide good estimate of the robot's pose, for example, Kalman filter and Particle filter.

2.1 Kalman Filter

Kalman filter is an algorithm which can quickly produce accurate estimate of the true value of the variable being measured after just a few sensor measurements. There are two steps to perform in each iteration of Kalman Filter algorithm: measurement update and state prediction. After a measurement update, it outputs a unimodal N-dimensional Gaussian distribution with N numbers of mean and variance. This output will then be used as prior belief in state prediction and be applied to different formulas depending on how many dimension to generate another new posterior Gaussian.

However, Kalman filter has a limitation, it can only be applied to linear systems. The reason is when Gaussian distribution undergoes a nonlinear function,it would result in a non-Gaussian distribution which is much more computationally expensive to work with. Also due to the fact that real-worlds systems are more often non-linear than linear, a nonlinear version of Kalman Filter named as Extended Kalman filter was introduced. It linearizes the nonlinear function f(x) over a small section using Taylor Series and hence the resulting distribution is a Gaussian.

2.2 Particle Filter

As the name implies, this algorithm uses particles to estimate robot pose. In beginning, particles will be generated and be spread randomly and uniformly throughout entire map. Each particle represents a guess of where the robot might be located. Whenever robot moves, importance weight is assigned to each particle, larger weight to the particles that likely represent the robot actual pose. Particle will then go through re-sampling process where particles

with large weight are more likely to survive and re-drawn in next iteration. After several iterations of the algorithm and after different stages of re-sampling,particles will converge and estimate robot current pose.

2.3 Comparison / Contrast

TABLE 1
Comparisons between Kalman Filter and Particle Filter

Aspect	Kalman Filter	Particle Filter
Easier to Implement	No	Yes
Supported Distribution(s)	Gaussian	Any
Memory & Resolution Control	No	Yes
Posterior	Gaussian	Particles
Higher Efficiency (Memory&Time)	Yes	No
Localization problem	Local	Global
State Space	Unimodal Continuous	Multimodel Discrete

As summarized in Table 1, Particle filter has comparatively more advantages than Kalman filter. In the experiment, a variant of particle filter which is Adaptive Monte Carlo Localization would be implemented. It is computationally much more efficient than basic particle filter as it could dynamically adjusts the number of particles over a period of time while the robot navigates around in a map.

3 Simulations

Gazebo, a physics simulator was being used throughout the experiment. Robot model would each be spawned on a simulated map which has a width of 2912 pixels and a height of 2496 pixels. RViz was being used to visualize camera and laser data which were published over corresponding ROS topics. The simulation was integrated with ROS amcl and move_base packages for robot localization and navigation functionality respectively.

3.1 Benchmark Model

3.1.1 Model design

Benchmark model is named as udacity_bot. This model has a chassis encompassing a cuboid base and two spherical caster wheels for stability purpose. Two cylindrical wheels are connected to the chassis. A camera sensor represented by a cube with 0.05 meters each side is attached to the front of chassis. In addition, a hokuyo sensor is installed on the top of chassis. The overall look is shown in Figure 2 and detailed specifications of the robot are described in Table 2.

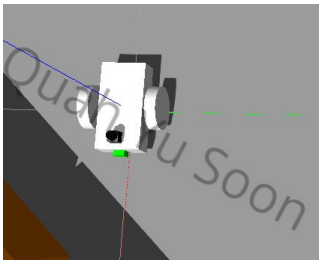

Fig. 2. udacity_bot.

TABLE 2
Specifications of udacity_bot

Link/Joint	Name	Specifications
link	robot_footprint	
joint	robot_footprint_joint	type: fixed origin: [0,0,0,0 0,0] parent: robot_footprint child: chassis
link	chassis	base geometry: box (0.4m x 0.2m x 0.1m) front caster origin: [0.15,0,-0.05,0 0,0] geometry: sphere (radius=0.05m) back caster origin: [-0.15,0,-0.05,0 0,0] geometry: sphere (radius=0.05m)
link	left_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
link	right_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
joint	left_wheel_hinge	type: continuous origin: [0,0.15,0,0 0,0] parent: chassis child: left_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
joint	right_wheel_hinge	type: continuous origin: [0,-0.15,0,0 0,0] parent: chassis child: right_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
link	camera	geometry: box (0.05m x 0.05m x 0.05m)
joint	camera_joint	type: fixed origin: [0.2,0,0,0 0,0] parent: chassis child: camera
link	hokuyo	geometry: mesh (filename=hokuyo.dae)
joint	hokuyo_joint	type: fixed origin: [0.15,0,0.1,0 0,0] parent: chassis child: hokuyo

3.1.2 ROS Packages/Gazebo Plugins Used

A few essential ROS packages and Gazebo plugins were needed to ensure the simulation works during experiment and are all listed in Table 3.

3.1.3 Parameters

Both amcl and move_base nodes contain set of parameters. Selected parameter values are listed in Table 4 after rounds of tunings in an attempt to achieve best experiment result.

3.2 Self-Developed Model

3.2.1 Model design

This model is named as custom_bot and is similar to benchmark model except that caster wheels are removed from

chassis and additional two cylindrical wheels make the robot a four-wheeled. The overall look is shown in Figure 3 and detailed specifications of custom_bot are described in Table 5.

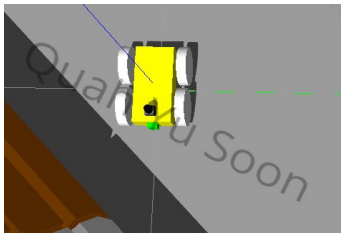

Fig. 3. custom_bot.

3.2.2 *ROS Packages/Gazebo Plugins Used*

All the packages and plugins for benchmark model would also be applied to this model except differential_drive_controller. As this model has four wheel joints, skid_steer_drive_controller would be used instead.

3.2.3 *Parameters*

All the tuned parameters for benchmark model would also be applied to this model without any changes.

4 RESULTS

The PoseArray, in RViz, depicts number of particles and is represented as arrows around the robot. As explained earlier, each arrow represents a guess of where the robot might be located. When arrows converge around one point,the robot pose is more certain. When the spread of arrows is large, it indicates high uncertainty of robot pose.

In the experiment, both robot models took around one minute to navigate from origin to a fixed goal position. As robots moved forward, arrows would quickly converge. However, arrows would begin to spread when robots stopped and steered.

4.1 Benchmark Model

Figure 4 shows initial PoseArray around robot at starting point. Figure 5 shows udacity_bot following global path (blue line) computed by move_base node. Arrows converged and centered around robot when robot was moving towards goal position as shown in Figure 6. Figure 7 shows final PoseArray when udacity_bot reached goal position.

4.2 Self-Developed Model

Figure 8 shows custom_bot following global path. Arrows converged and centered around robot when robot was moving towards goal position as shown in Figure 9. Figure 10 shows final PoseArray when custom_bot reached goal position.

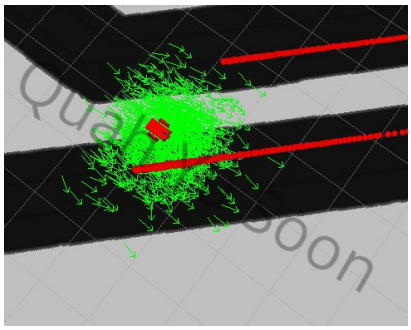

Fig. 4. PoseArray at starting point.

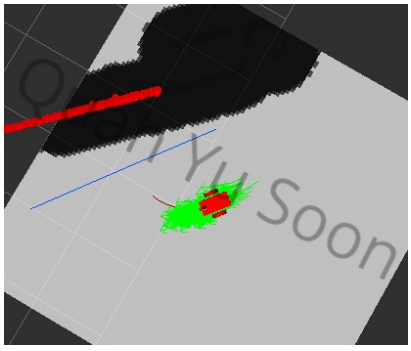

Fig. 5. udacity_bot moved by following global path.

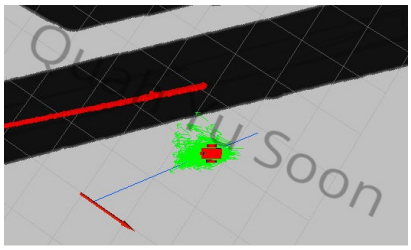

Fig. 6. PoseArray when udacity_bot moved forward.

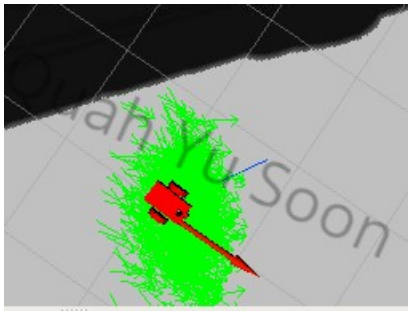

Fig. 7. PoseArray when udacity_bot reached goal position.

4.3 Technical Comparison

In following global path, custom_bot performed better as its navigation was always stick to the path. However, udacity_bot has better localization result because when it stopped at the goal, majority of the arrows remain centered around the robot and pointed in the same direction as goal. As for custom_bot, arrows were spread widely indicating high uncertainty of robot pose.

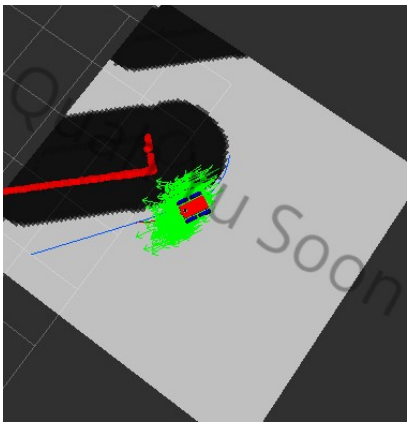

Fig. 8. custom_bot moved by following global path.

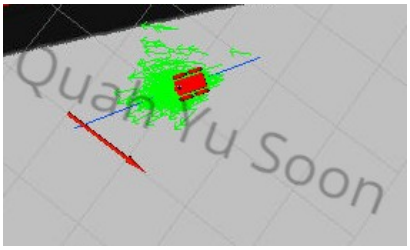

Fig. 9. PoseArray when custom_bot moved forward.

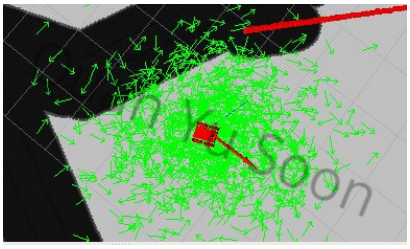

Fig. 10. PoseArray when custom_bot reached goal position.

5 Discussion

Performances of both robot models were considered good as arrows did quickly converge while robots were moving forward. One may believe udacity_bot localized itself better than custom_bot at the goal because different drive controllers were used. Robot paired with differential drive controller seemed to steer slower than the one paired with skid steer drive controller.

AMCL seemed to work well in solving kidnapped robot problem as such scenario was tested during experiment. Figure 11 shows arrows scattering around the map at the moment robot was kidnapped. Arrows began to converge after rotate recovery behavior started and robot continued to move as shown in Figure 12.

ACML could also be used on unmanned aerial vehicles to deliver food or goods.

6 Conclusion / Future work

Localization worked well on both robot models in overall and they were successfully navigated to the goal position by just taking around one minute.

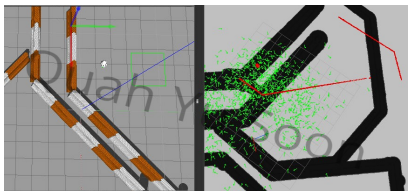

Fig. 11. PoseArray at the moment robot was kidnapped

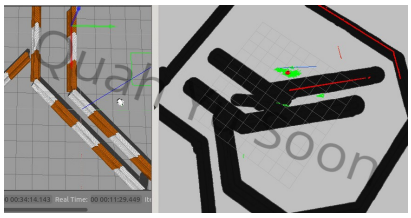

Fig. 12. PoseArray when robot recovered and continued to move.

In the implementation of AMCL, when robot pose is highly uncertain, the number of particles is increased in an attempt to achieve better accuracy and thus an increase in processing time would occur. When robot pose is well-determined, the number of particles is reduced and processing time is shortened.

There is room for improvement on the robot functionalities. For future works, robot arm and gripper could be installed on the robot to empower it with grabbing ability. Inertial Measurement Units (IMU) sensor could be a good choice to integrate into the robot as it is capable of providing feedback by detecting changes in an objects orientation (pitch, roll and yaw), velocity as well as gravitational forces.

TABLE 3
Packages/Plugins Used

Type	Name	Features
ROS package	gazebo_ros	Spawns any URDF models in Gazebo
ROS package	rviz	Visualizes data being published over ROS topics
ROS package	map_server	Offers map data as a ROS Service
ROS package	amcl	Takes in a laser-based map, laser scans, and transform messages, and outputs pose estimates
ROS package	move_base	Computes a local costmap and global costmap. As robot moves around, local costmap keeps getting updated allowing the package to define a continuous path for the robot to move along.
Gazebo plugin	differential_drive_controller	Controls wheel joints to move robot in a specific direction in simulated environment
Gazebo plugin	camera_controller	Provides image capturing functionality to camera object during simulation
Gazebo plugin	gazebo_ros_head_ hokuyo_controller	Provides hokuyo sensor laser scan functionality during simulation

TABLE 4
Tuned parameters in ROS nodes

Node	Parameter	Usage	Selected Value	Reason
amcl & move_ base	transform_ tolerance	Specifies the delay in transform (tf) data that is tolerable in seconds	0.2	Default value of this parameter
move_ base	update_ frequency of both local & global costmap	Specifies frequency in Hz for the map to be updated	5.0	Default value of this parameter
move_ base	publish_ frequency of both local & global costmap	Specifies frequency in Hz for the map to be published display information	5.0	To have same frequency as update_ frequency
move_ base	width of local costmap	Defines width of local costmap in meters	5.0	To consume less memory resources storing obtacles' information since local costmap was configured as rolling window
move_ base	height of local costmap	Defines height of local costmap in meters	5.0	To have a square local costmap
move_ base	obstacle_range	Defines maximum distance from the robot at which an obstacle will be inserted into the cost map in meters	2.5	Default value of this parameter
move_ base	raytrace_range	Defines range in meters at which to raytrace out obstacles from the map using sensor data	3.0	Default value of this parameter
move_ base	inflation_radius	Defines radius in meters to which the map inflates obstacle cost values	0.5	Close to default value of this parameter (0.55)
amcl	min_particles	Defines minimum allowed number of particles	100	Default value of this parameter
amcl	max_particles	Defines maximum allowed number of particles	1000	Avoids high maximum as it could result in heavier computational task
amcl	update_ min_d	Defines translational movement required in meters before performing a filter update	0.1	To trigger robot pose estimate in more frequent manner

TABLE 5
Specifications of custom_bot

Link/Joint	Name	Specifications
link	robot_footprint	
joint	robot_footprint_joint	type: fixed origin: [0,0,0,0 0,0] parent: robot_footprint child: chassis
link	chassis	base geometry: box (0.4m x 0.2m x 0.1m)
link	leftfront_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
link	rightfront_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
link	leftrear_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
link	rightrear_wheel	origin: [0, 0, 0, 0, 1.5707, 1.5707] geometry: cylinder (radius=0.1m;length=0.05m)
joint	leftfront_ wheel_hinge	type: continuous origin: [0.11,0.15,0,0 0,0] parent: chassis child: leftfront_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
joint	rightfront_ wheel_hinge	type: continuous origin: [0.11,-0.15,0,0 0,0] parent: chassis child: rightfront_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
joint	leftrear_ wheel_hinge	type: continuous origin: [-0.11,0.15,0,0 0,0] parent: chassis child: leftrear_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
joint	rightrear_ wheel_hinge	type: continuous origin: [-0.11,-0.15,0,0 0,0] parent: chassis child: rightrear_wheel axis: [0,1,0,0 0,0] limit: effort=10000;velocity=1000 dynamics: damping=1.0;friction=1.0
link	camera	geometry: box (0.05m x 0.05m x 0.05m)
joint	camera_joint	type: fixed origin: [0.2,0,0,0 0,0] parent: chassis child: camera
link	hokuyo	geometry: mesh (filename=hokuyo.dae)
joint	hokuyo_joint	type: fixed origin: [0.15,0,0.1,0 0,0] parent: chassis child: hokuyo