Udacity Robotics Nanodegree Term 2

Where Am I?

Robert Aleck, 19th June 2019

Abstract

In this paper, the author describes the steps taken to tune a number of ROS packages to correctly localize a robot in a simulated environment, and navigate inside a provided map to a given target. Simulation visualisation is provided by RViz and Gazebo.

Introduction

The required robot consists of at least 2 actuators, a camera, and a laser sensor, although the camera is not used for localisation or perception tasks. It was required to use the Adaptive Monte Carlo Localization package (https://wiki.ros.org/amcl), and the standard ROS Differential Drive Controller (https://wiki.ros.org/diff_drive_controller), along with a simulated hokuyo laser sensor (http://gazebosim.org/tutorials?tut=ros_gzplugins#Laser).

The robot was placed in to a simulated maze at an arbitrary location. A global map was provided; the robot had to localize within that global map, and complete navigation to an arbitrary point (x,y,yaw) within the map, avoiding obstacles.

It was not required to test the robot against the “kidnapped robot” problem [1].

Background

Localization background

Localization in robotics is “the process of determining where a mobile robot is located with respect to its environment” [2], including both location and orientation. Localization is a prerequisite for many successful mobile robotics systems, and so creating and maintaining an accurate model of the world, and the robot’s location and orientation within that, is a foundational component of many autonomous mobile robotics systems, and is critical to path planning.

Localization can be broken down in to three types, with increasing complexity:

- Position tracking – determining the pose of a robot, given a starting pose, and incremental sensor readings (e.g. odometry) relating to changes in that pose.
- Global localization – no initial pose is given, and the robot must determine its pose entirely using sensor readings.
- “kidnapped robot” – where “a well-localized robot is tele-ported to some other place without being told” [1] – requiring the robot to be able to recover from an absolute failure of localization.

In each scenario, the robot may be given a base (or reference) map, or have to determine one from scratch.

Mapping

Generating a map of the environment is one of the necessary conditions for truly autonomous systems [1]. Maps can either be [3]:

- *metric maps*, which aim to encode the world in terms of fixed locations and the geometric or spatial relations between them e.g. a feature map (as we might see at the entrance to a funfair), or occupancy grid (probabilistic representation of call occupancy), the latter of which being extremely cost useful for path planning, but at a relatively high computational cost.
- *topological maps*, which describe adjacency relationships between known objects. This method is useful in highly controlled environments, such as automated warehouses.

While topological maps provider higher accuracy in localization, metric maps provide more robust path planning.

Localization techniques

In its simplest form, localization will incrementally accrue odometry (such as wheel motion, or data from a Doppler velocity log [4]) with the robot's targeted motion (dead reckoning) and known start pose to estimate the current pose, however, as the calculation is incremental, measurement errors rapidly accrue leading to estimated pose drift.

Other hardware can be added, such as an Inertial Measurement Unit or IMI to reduce drift, but higher accuracy requires more complex techniques such as Simultaneous Localization and Mapping (SLAM) or Visual Odometry (VO).

In Visual Odometry, video data is analysed and used to estimate the camera's relative motion. Although subject to accumulated estimate error, VO provides significant improvements over wheel-based odometry, as demonstrated by its use on the Mars rover [5].

SLAM is the process of a robot localizing concurrent to incremental construction of a worldmap to allow the robot to develop a globally consistent estimate of pose with respect to environment, the latter providing benefits in tasks like path planning, where priority can be given to previously unsearched areas, or to previously traversed paths therefore assumed to be "safe". SLAM is typically

4 common techniques are used for localization with SLAM:

- Extended Kalman Filters (EKF)
- Markov Localization
- Grid Localization
- Monte Carlo Localization (MCL)

With Markov Localization, the robot maintains a probability distribution over all possible location states. Given a prior assumption that the environment is static, as the robot moves and acquires further sensor readings, the probability distribution is updated [6].

In Grid localization, a known map over all possible pose spaces is provided, and then broken in to a finite grid. Each cell in the grid is given a probabilistic estimate representing the likelihood that it is the current pose of the robot, calculated using a Bayesian update algorithm. [7]

Extended Kalman Filters

Kalman Filters allow a series of time series sensor readings taken to be adjusted/weighted according to the calculated reliability of the reading with respect to noise, and combined with other sensor readings to produce a joint probability distribution which is more accurate than any single probability set. Kalman Filters work only on linear systems.

The Extended Kalman Filter linearizes the estimation through a Taylor series expansion around the mean and covariance, allowing some non-linear systems to be estimated. Kalman Filters are highly computationally efficient, but propagation of covariance when linearizing an inherently non-linear model can cause them to significantly reduce in accuracy.

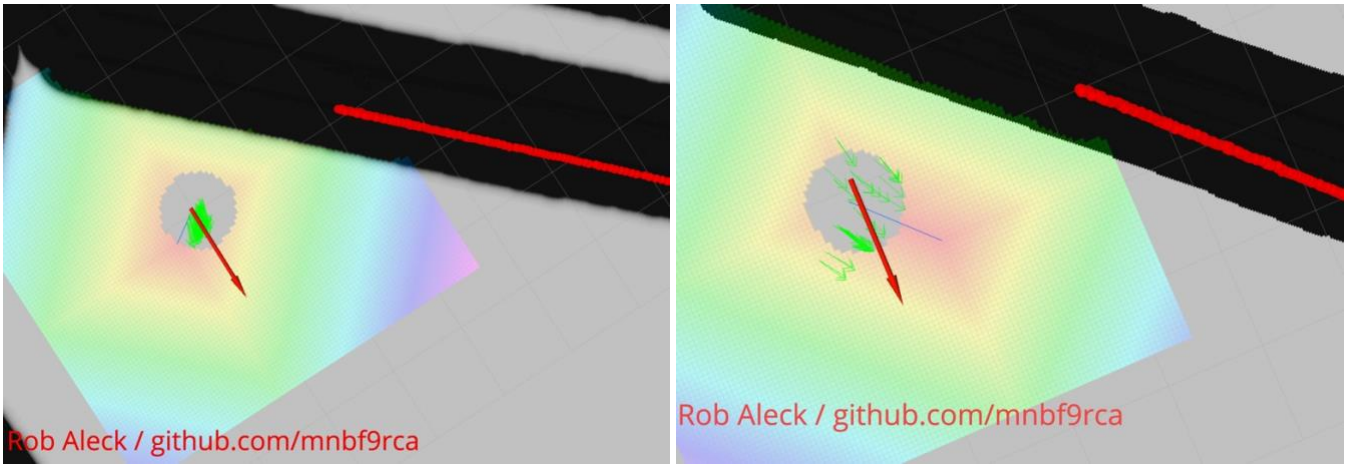
Other Kalman Filter-based models exist.

Monte Carlo Localization

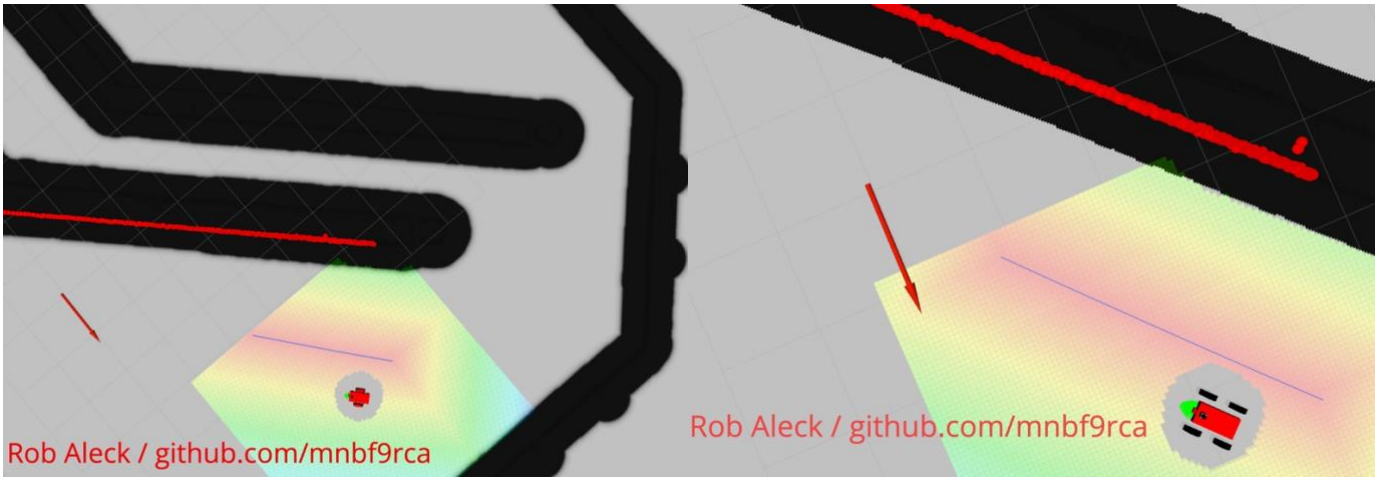
Although computationally heavier than EKF, MCL (also referred to as a “particle filter”) is applicable to both linear and non-linear systems (making no assumption about the underlying probability distribution). A set of particles are distributed randomly (and usually uniformly) over the entire pose space. The robot moves, and updates the calculated probability distribution across the particles based on expected movement. Sensor readings are then recursively applied through Bayesian estimation to update the probability distribution across the particles with the observed data. Particles are then resampled, removing a number of points at random, weighted towards those which have the lowest probability of being correct. MCL generally requires a pre-existing ground map, and with a sufficiently tuned particle decay, MCL will allow relocalization in the “kidnapped robot” scenario.

Results

The robot successfully localized within the map space for both the original UdacityBot (left) and my customised version (a 4 wheel robot using the libgazebo_ros_planar_move plugin).



During motion, the system maintained a highly localized state (original bot on left, modified on right):



Model Configuration

To meet the project rubic, the robot was modified to remove the "free moving balls" underneath, and to add a second set of wheels. An attempt was made to use the libgazebo_ros_skid_steer_drive plugin, but lack of documentation meant this wasn't possible. Instead, the simpler libgazebo_ros_planar_move plugin was used. Only the following files were modified to create the second robot, rob_bot:

src/rob_bot/config/base_local_planner_params.yaml

src/rob_bot/config/base_local_planner_params.yaml

src/rob_bot/urdf/rob_bot.gazebo

src/rob_bot/urdf/rob_bot.xacro

Parameter tuning

amcl.launch

Initial experiments showed that the update angle and distance were far too large for the small, slow robot, so these were reduced:

```
<param name="update_min_a" value="pi/1000"/>
<param name="update_min_d" value="0.01"/>
```

Additionally, the system seemed to quickly localize with just 20-100 particles. Reducing this from the default of 1000 significantly reduced compute resource consumption:

```
<param name="min_particles" value="20"/>
<param name="max_particles" value="100"/>
```

Early on, some noise was noticed close to the robot. A minimum range for laser data was added:

```
<param name="laser_min_range" value="0.5"/>
```

Values were added for exponential decay for slow and fast average weight filters, enabling further recovery through random pose insertion:

```
<param name="recovery_alpha_slow" value="0.001"/>
<param name="recovery_alpha_fast" value="0.1"/>
```

This file did not need to be further changed for the adjusted robot.

base_local_planner_params.yaml

Publishing the cost map allowed investigation of cost computation:

```
publish_cost_grid_pc: true
```

It was after the cost grid was visualized that it became apparent that the global goal was significantly impacting local path planning, with the robot consistently hitting the walls. In response, other settings in costmap_common_params.yaml and local_costmap_params.yaml were adjusted as outlined below.

The robot tends to vary quite significantly from the planned path. This is slightly mitigated by increasing the pdist_scale and gdist_scale values to increase the weight of local path and goal planning:

```
pdist_scale: 1.1
gdist_scale: 0.5
```

The modified robot had trouble navigating the corridors with these settings, constantly bouncing from one side to the other. The following adjusted settings smoothed local path planning and made the robot significantly more likely to follow the local cost map:

```
pdist_scale: 0.2
gdist_scale: 1.5
```

costmap_common_params.yaml

The obstacle_range and raytrace_range were increased to a more reasonable distance, and the cost_scaling_factor increased to reduce the importance of distant objects over local. The transform_tolerance was increased to 3 through trial and error on the local machine. An inflation_radius of greater than 0.4 resulted in the robot consistently getting stuck; lower than 0.3 resulted in collisions.

```
obstacle_range: 4.5
raytrace_range: 4.5
cost_scaling_factor: 25
transform_tolerance: 3
```

inflation_radius: 0.4

With the second robot, cost_scaling_factor was adjusted downwards to try and prevent the robot “bouncing” from the walls:

cost_scaling_factor: 2

local_costmap_params.yaml

update_frequency and publish_frequency were reduced to improve availability of data for planning. The local cost map was also shrunk from 20x20 to 5x5 to prevent global goals outside the current corridor from overriding local planning goals. XY and yaw tolerance was added to improve final result localization.

```
update_frequency: 5.0
publish_frequency: 2.0
width: 5.0
height: 5.0
xy_goal_tolerance: 0.03
yaw_goal_tolerance: 0.01
```

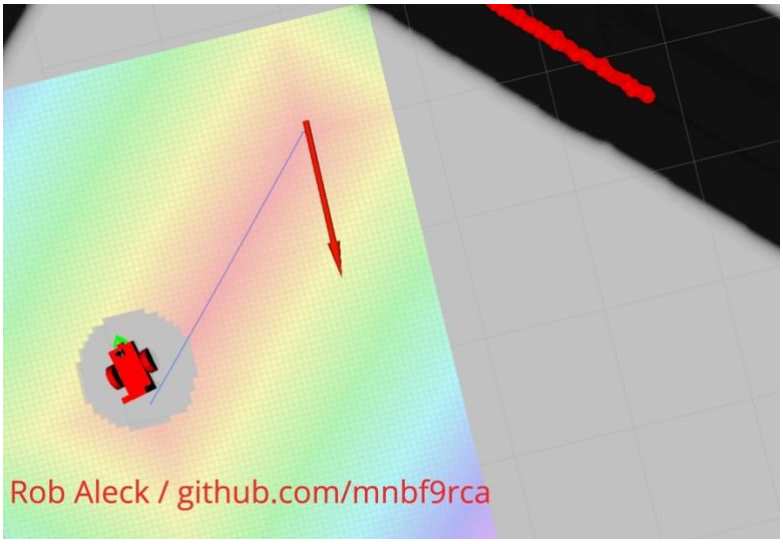
global_costmap_params.yaml

update_frequency and publish_frequency were reduced to improve planning.

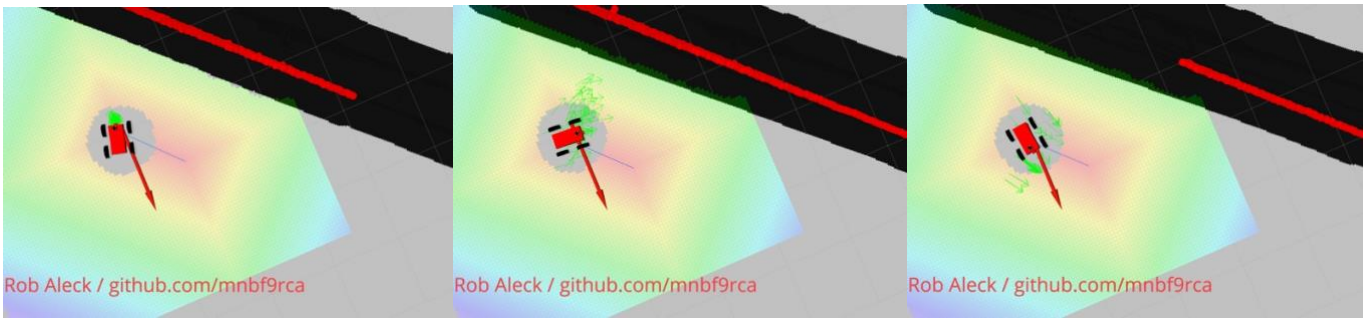
```
update_frequency: 10.0
publish_frequency: 5.0
```

Discussion

Overall, the robot achieved a satisfactory localization result, despite relatively poor path-following behavior – as can be seen below, the local cost model clearly shows the most efficient path to the navigation goal, however, the robot has taken a longer, more circuitous path to reach the goal.



The second robot was highly localized for most of the journey from start to goal, however, when it reached the goal, it rotated in place, which caused a loss of localization:



Overall, however, the authors are satisfied that the robot meets the design criteria.

Use of MCL

In principle, MCL (and in particular adaptive MCL) can recover from the kidnapped robot problem, as long as there are sufficient particles available on which to base relocalisation. In practice, this means increasing the number of particles when location confidence degrades, and then dropping them as confidence increases.

MCL and AMCL are useful in real world scenarios where there is a high degree of uncertainty about the world but a basemap is known. Examples may include:

- Industrial robotics, such as Amazon's "drive" robots, moving items around a warehouse
- General localization on a street map against features such as buildings, cars etc.
- Home robotics – such as the iRobot Roomba

Future Work

The model could be extended with additional sensors to determine whether this could increase accuracy (at the cost of additional compute). Work might also be undertaken to model other shapes (wider base, taller robot, etc.) and evaluate the impact of changes in collision zones on path planning. Finally, the robot might be remodeled as an non-holonomic robot, allowing greater range of movement.

References

[1] S. Thrun, D. Fox, W. Burgard and F. Dellaert, "Robust Monte Carlo localization for mobile robots," *Artificial Intelligence,* vol. 128, no. 1-2, pp. 99-141, 2001.

[2] S. Huang and G. Dissanayake, "Robot Localization: An Introduction," in *Wiley Encyclopedia of Electrical and Electronics Engineering*, J. G. Webster, Ed., John Wiley & Sons, Inc, 2016, pp. 1-10.

[3] K. Yousif, A. Bab-Hadiashar and H. Hoseinnezhad, “An Overview to Visual Odometry and Visual SLAM: Applications to Mobile Robotics,” *Intelligent Industrial Systems,* vol. 1, no. 4, p. 289–311, 2015.

[4] E. Westman and M. Kaess, “Underwater AprilTag SLAM and calibration for high precision robot localization,” Carnegie Mellon University, Pittsburgh, PA, 2018.

[5] D. Helmick, Y. Cheng, D. Clouse, L. Matthies and S. Roumeliotis, “Path following using visual odometry for a Mars rover in high-slip environments,” in *2004 IEEE Aerospace Conference Proceedings (IEEE Cat. No.04TH8720)*, Big Sky, MT, USA, 2004.

[6] D. Fox, W. Burgard and S. Thrun, “Markov Localization for Reliable Robot Navigation and People Detection,” in *Sensor Based Intelligent Robots. Lecture Notes in Computer Science, vol 1724*, Berlin, Heidelberg, Springer, 1999.

[7] Z. (. Li, “Robitics: Science and Systems - Localization: fundamentals & grid localization,” 2018. [Online]. Available: http://wcms.inf.ed.ac.uk/ipab/rss/lecture-notes-2018-2019/5%20RSS%20Localization_%20fundamentals%20-%20grid%20localization.pdf. [Accessed 18 06 2019].