CompRobo - Mobile Robots

Nathan Lintz and Diana Vermilya

October 2014

1 Project Goal

The goal of this project was to develop a Bayesian particle filter for robot localization. Using the Neato robots and Professor Ruvolo's particle filter starter code [1], we implemented functions which targeted three main aspects of particle filtering creating a particle cloud representing hypotheses of the robot's position, updating the likelihoods of these particles using LIDAR data, and resampling these particles to update our hypothesis of the robot's position. For our particle creation function, our goal was to generate a good initial distribution for the particle positions and to properly update the particles positions as the robot moved. For our likelihood function, our goal was to create a method which updated the weights of each particle which could account for each value in the LIDAR's scan. Finally, for our resampling function our goal was to find a way to aggressively remove unlikely particles while avoiding sampling impoverishment[2].

2 How We Solved The Problem

We solved the particle filter problem by breaking it down into 5 discrete steps. First, we had to figure out how to load a map file into the occupancy grid supplied by the particle filter code. This required us to understand how the map_server service provided by ROS worked. Once we figured out how to load a map file, we were ready to begin constructing our particle filter. The first step of developing our particle filter was actually initializing our particles. We were given an initial guess of the robots position by the initial pose topic. Using this guess, we assigned 300 particles to the filter where each particles x, y, and theta was equal to the robot's initial position plus noise which we modeled using a Gaussian PDF.

We updated the position of the individual particles by applying the change in odometry to the x, y, and θ values of the particles. Calculating the change in θ was straightforward because the updated θ was simply the sum of the previous θ and the $\delta\ \theta$ of the robots odometry message. Updating the position (x, y) for each particle required a transform because change in position is relative to original heading.

After updating the particle's position's with odometry, we needed to figure out how to use the LIDAR data to improve our guesses of the robot's position using LIDAR data. We decided to use each LIDAR scan to improve our guess of the robots position. For each LIDAR reading, we transformed the reading relative to each particles position. We then took the PDF of a Gaussian centered around 0 for the value of the transformed LIDAR scan to the nearest closest object. This gave us the likelihood of a single laser scan being correct. Finally, we multiplied each of these PDF samples and cubed the result as per professor Ruvolo's suggestion to weight each particle.

The final step of our particle filter implementation was resampling particles. We randomly sampled our particles according to their weights to create a new set of particles representing the robot's position. By resampling particles, we made our particle filter more accurate each time it moved. Indeed, each generation of particles keeps the strongest of the previous particles while culling the weakest particles.

[1] $https://github.com/paulruvolo/comprobo2014/tree/master/src/my_pf/$

[2] http://robotics.stackexchange.com/questions/479/particle-filters-how-to-do-resampling

3 Design Decisions

One major design decision was to abstract all constants out into attributes of their corresponding class. This was encouraged by the code we were working with an made tweaking parameters easier.

In several places within our implementation, we needed to obtain a spread of data centered around a certain value. We chose to use Gaussian because we assumed the robot was susceptible to white noise. Based on the central limit distribution, white noise is best modeled using a Gaussian.

In order to avoid losing diversity in our particle cloud due to resampling, we needed to add noise to the system. We chose to add noise to the update_particle_with_odom function because we know that the odometry is inherently noisy.

4 Code Structure

There are three interesting ways in which we structured our code. First, we created external modules which helped us to test our design decisions. If you look in the scripts directory, you will see two scripts we used to test our code, PF_Optimization_Test.py and generateImageFromLidar.py. The first script computes the mean squared error between the robot's actual position and the position generated by the particle filter. This code helped us to quantitatively evaluate our likelihood functions. The second script generated an image from our LIDAR scan. We used this code to create a unit test for the geometry transforms we implemented in our LIDAR update function. In our LIDAR update function, we take each particle, transform each LIDAR reading relative to that particle's position and check to see how likely that LIDAR reading is. Given the strange trigonometry of the Neato namely its origin being 0 degrees instead of 90 degrees, we weren't confident that we were transforming LIDAR readings properly. By creating a way to visualize LIDAR scans, we could test that our understanding of the Neato's trigonometry was correct. Thus, the image generation script provided us with good test coverage for our LIDAR update function. The second interesting way we structured our code was that we broke down our code in the particle filter into useful helper functions. For example, when we ran into an issue with the built in normal PDF function being too slow. Instead of approximating the PDF in each function that used it, we created a static method which computed Gaussian PDFs. This made our code more modular and reusable. We based a lot of our helper function design on the elegant structure provided for us in the pf_level1 class provided by professor Ruvolo. In his code, he paid careful attention to separating concerns for the robot which we tried to emulate in our own helpers. The final interesting code choice we made was building useful launch files out of simpler launch modules. For example, we created a launch file to test our code on the Neato playground which included calls to a launch file which initiated gazebo, a launch file which initiated RVIS, and a launch file which initiated tele-op. By building more complex launch files out of simpler ones, we were able to optimize our workflow and avoid having to run multiple launch files in different terminal windows.

5 Challenges

First, we struggled a bit with finding relative metrics to compare different equations for the likelihood function. It was difficult to see how much of an effect different equations had on the behavior of the filter overall. Writing test cases that had good coverage on our implementation took a new level of creativity. For example, we ended up using a separate piece of code that plotted laser data to an image, in order to confirm that the trigonometry was accurate in translating laser data to a surrounding particle. Finally, the trigonometry itself was definitely a challenge, especially in the update_particle_with_odom function.

6 Future Improvements

We would really have liked to build a script to automate parameter sweeps for tuning the algorithm. Such a script would have started the launch file, collected data representing an amalgamated error, and terminated the processes for each entry in an array of parameter values. This sweep would have allowed us to tune parameters more scientifically. Another area for future improvement would be implementing a ray tracing model for calculating the likelihood function in updating the Particle Filter with laser data.

7 Lessons

The most important lesson we learned from this project was the importance of unit test coverage. Each function that you write should have a corresponding test to verify its correctness. In cases when we didn't have good test coverage we ran into issues verifying which functions were correct. This is compounded by the fact that most functions build on the other functions so errors in lower level code percolates to the higher level behaviors we are trying to program. Thus, it was very important for us to develop comprehensive tests for each unit of our code which we feel like we did pretty well but could improve on in the next project.