Localization

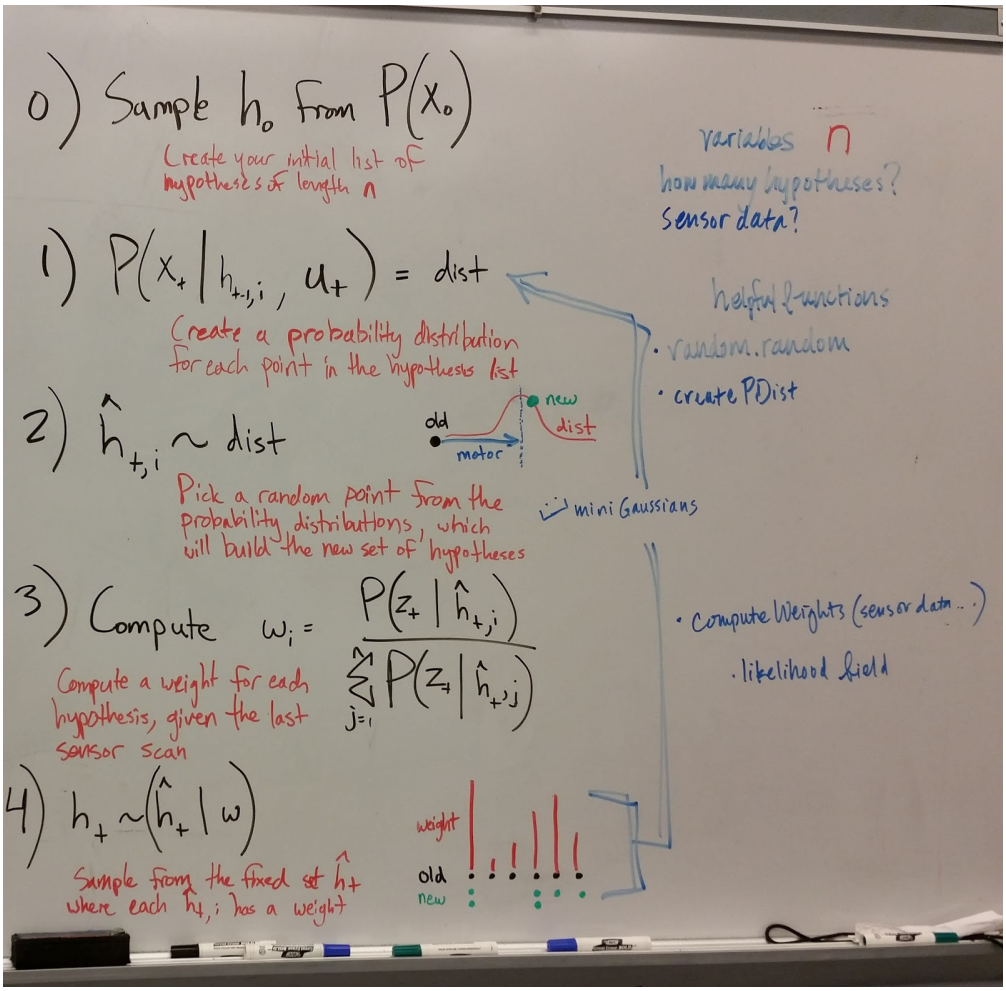
Emily, Franz, Cypress
Computational Robotics
Oct 14 2014

Project goal

We followed the goals set forth in the scaffolded project: localization.

Our solution

Ideally, we could make an initial guess of the probability of the robot being at any position on the map (our prior), and then update that guess with LIDAR and odometry data as the robot moves. It would be computationally costly to make a guess for every position on the map for any reasonable discretized resolution so we used a subset of the points on the map, which we initialized in various ways throughout our project.



^ A summary of the math involved with particle filters.

Important design decision

Since this project was scaffolded, we didn't encounter too many design decisions. We took advantage of the scaffolded code structure (i.e. the existing skeletons for the ParticleFilter, Particle, TransformHelpers, and OccupancyField classes; filling in the functions for updating particles with laser and/or odom) while working on the project. We obtained experience in reading, building off of, and optimizing already existing code.

Our code structure

We used the scaffolded project code structure.

Classes

`ParticleFilter` The algorithms that update our particle probabilities given our sensor data. Every time the robot moves by a certain distance threshold then ParticleFilter runs its filters using odometry and LIDAR data to narrow in on the correct point.
`Particle` Convenience class to keep track of a position on we the map and its probability
`TransformHelpers` Utilities to help us with some transform math
`OccupancyField` Helps us get the distance to the closest obstacle given any point on the map

Launch files

We used a launch file that started the map server, started rviz with the proper settings, and ran our particle filter script. At times we launched our script independently of rviz because we wanted to run our script multiple times without restarting rviz.

Maps

We placed our maps in our repo in `particle_filter/maps` and we used them when running the map server. Our launch file defaulted to a specific map, but we could also pass the path of a different map as an argument if needed.

Bag files

We collected some bags of the robot moving a short distance in the STaR center which we placed in our repo in the `bagfiles` directory. We used them to test our script after we got it to work in the simulator.

Challenges we faced

Computational time

Our initial version took 0.1 ms to compute the weight for each laser scan point. Using 30 position guesses and all 360 laser scan points took 30*360*0.1 ms, or about 1 second for each laser scan update. We used a timer to identify the most costly operation, which turned out to be `scipy.stats.norm` and replaced it with our own function that was 10x faster and did the same thing. Even with this optimization, the updates are still slow. For example, if we wanted to use 300 position guesses, the computation would take over 1 second for each laser scan update.
We generally compromised by picking a 1) number of particles and 2) number of laser points to check for each particle that in all took around 200 ms to calculate, which worked out because that was how fast the laser scans updated. As a baseline we could analyze 250 laser points for 30 hypothetical position guesses, or (equivalently) 25 laser points for 300 hypothetical position guesses.

How to combine the laser scan weights

Initially we calculated a likelihood of each laser scan point and then multiplied them together to get the total likelihood for the position guess. Problematically, the likelihoods were so small that we encountered floating point underflow (http://en.wikipedia.org/wiki/Arithmetic_underflow).

We tried fixing it by adding 1 to each likelihood, which prevented underflow by making the numbers converge on 1 instead of 0, but was not conceptually or mathematically sound. There are other ways to combine the likelihoods, which we discuss in the *improvements* section.

Ideally, we would also take into account the fact that laser scans are not independent. The total probability of seeing all of them is not simply their probabilities multiplied together, but instead is the probability of each one *given all the others*.

Converging to the wrong point

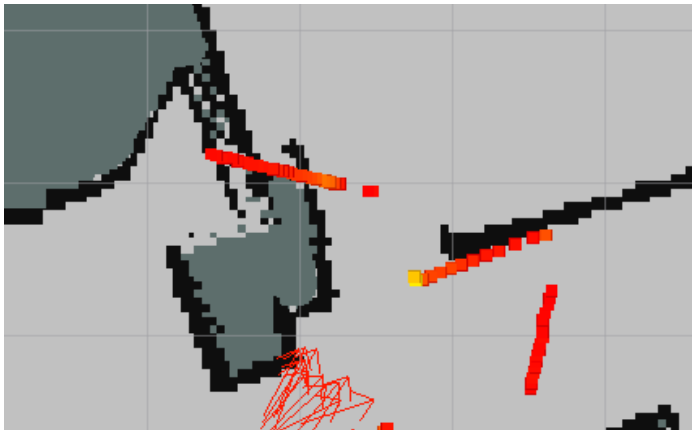
When we used our algorithm with data from the Neato and a map of the star center we created using hector slam it would often converge to the wrong point, then all the future guesses would be very close to that point. It would get stuck in a rut. Some ways to fix this include:

- Re-randomize the points across the map if the probability of the best of them falls below a certain threshold.
- Increase the jitter on the points when resampling.
- Improve the initial guesses.
- Improve the quality of the map.
- Change the algorithm to be less aggressive by adding in a term to account for outlier points and extraneous map data.
- Change the algorithm to be less aggressive by combining the laser scan likelihoods in a different manner.

Future improvements

Ray tracing

Our current implementation will examine a given laser point, match it up with pixels behind interfering walls, and call it a good match, which is bad. Ray tracing would have helped our robot calculate the probability of a hypothesis more accurately by identifying the occupied point on the map the laser would have hit. It would also would require a fair amount of work because the scaffolded project code was designed to use nearest neighbor.



Here's a real-world example. Our robot sees points that match up well with a chunk of wall that would not register given our hypothesis.

Automated testing

Automated testing of algorithms could be really helpful. We felt a little helpless at the end of the project because we had a lot of parameters of the algorithm that we could adjust but we were only evaluating the results by eye and it took a while to test various hypotheses.

Better visualizations of errors and results

For example, color coding each hypothesis in rviz to account for its weight, plotting the exact position of the robot alongside its hypothesized positions, or somehow visualizing the inner workings of our updates by incrementally plotting the weights and hypotheses as they are updated via laser and odom data.

Different probability accumulators

We talked with Paul about various ways to accumulate the probability of each laser scan from a hypothesis into a guess at the probability of that hypothesis.

Did alright	**Did poorly**	**Not fully explored**
Averaging the probabilities	Multiplying the probabilities (too extreme, converges too quickly)	Adding the logs of the probabilities
Adding a constant to the probability calculation for each laser scan before accumulating		Adding the cubes of the probabilities

The "adding a constant to the probability calculation" statement is a little unusual, and bears explaining. In the original particle filter that we wrote, we said that for each laser scan, the probability that the laser scan was correct could be stated as

$$probabilty\ of\ correct\ scan\ = gaussian(\ mu = 0,\ sigma = laser\ variance,\ x = nearest\ neighbor\ to\ scan\ on\ map\ )$$

We heard from Paul that the built-in particle filter added another gaussian in to account for the possibility that the laser data was bad, but we were confused at to why that probability would be a gaussian. It seemed to us that the likelihood of the given laser data was bad would be constant regardless of what distance the laser was reporting. We also realized that we shouldn't only account for the possibility that the

laser scan is bad, we should at the same time acknowledge that the map might be bad at that point. Our map of the STAR center was definitely not perfect. We tried rehashing the probability estimation like so:

$$probabilty\ of\ correct\ scan\ = gaussian(\ mu = 0,\ sigma = laser\ variance,\ x = nearest\ neighbor\ to\ scan\ on\ map\ ) + constant\ chance\ of\ error\ in\ map\ or\ laser\ data$$

The addition of this constant offset definitely made the code less aggressive. It made the low probability points more likely and kept them in the game, but didn't really help the algorithm resolve in the right place.

Valuable lessons for future robotic programming projects

- Make good visualizations of your results
- Use rosbag to test repeatably
- It's useful to work off of someone else's code (scaffolding)
- Roslaunch prevents you from getting a migraine trying to remember your commands
- It's nice to save an rviz configuration for testing
- We had data type conflicts that caused nasty errors, were hard to debug (numpy array vs list, etc.) Lesson: be careful with types
- Python can be too slow for localization algorithms
- We learned a lot about version control