16-831 Project 3: Robot Localization

Alex Brinkman, Shivam Gautam, Tushar Agrawal

October 26, 2016

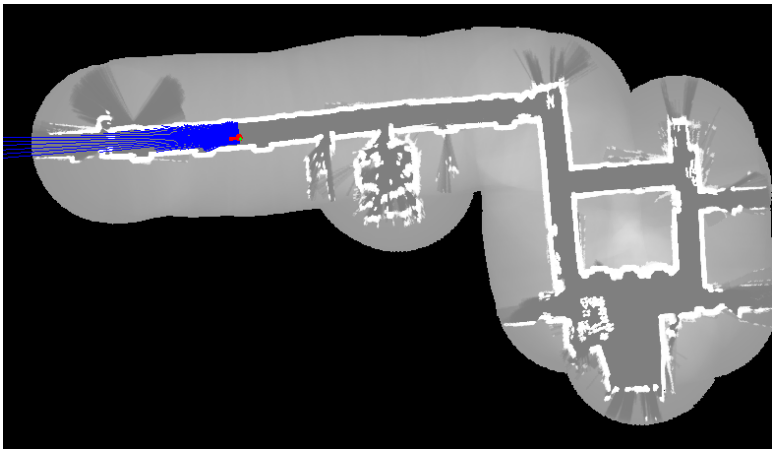

Figure 1: Visualization of the Particle Filter after Convergence

1 Approach

We started by first developing the modules in isolation to facilitate future integration. We started by independently developing the sensor model, motion model, ray tracing, and graphics engine, and log-streaming modules first. We performed unit testing on each to verify they were working properly, relying heavily on the graphics to validate the complex modules. Once the modules were build up, we integrated the modules and debugged the interfacing issues and added importance resampling. Since the modules were tested independently, this was relatively painless. Finally, we refined the pipeline to improve runtime and tuned the particle filter parameters to find a good trade-off between convergence speed and correctness. Figure 1 depicts the final outcome of the particle filter.

2 Implementation

We implemented the particle filter in C++ and used a private git repo in our workflow. All modules were implemented as stand alone classes and invoked in executable scripts. The final integration took the individual classes and encapsulated the modules while exposing configuration parameters to ease tuning. We used GNUplot-iostream to visualize 1D data like the sensor model and the SFML library to visualize the map, particles, and laser readings.

2.1 Sensor Model

The sensor model consists of a superposition of an exponential decay function, Gaussian distribution, uniform distribution, and max value. The Gaussian distribution mean is placed at the expected range measurement

with a tunable variance. Since the we need to sum the log-likelihood of each of the 180 range readings, the uniform distribution of the model is 1 so for any range measurement this provides a non-negative log-likelihood. The decay function plays a small role in the overall shape of the function, only affecting the first meter or so. The Gaussian was tuned first by using reasonable values for gain factor and standard deviation. We found that once we set the initial values to something reasonable, we did not have to tune this model much to get the performance we desired. Figure 2 shows the final sensor model used in our tuned model and Figure 3 shows the final parameters.

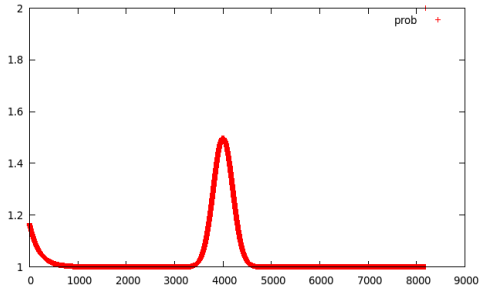

Figure 2: Sensor Model for reference reading for 4000cm, X-axis is cm

uniformParam	1
decayScale	0.167
decayRate	0.002
maxParam	0.333
rangeSTD	66.667
gaussianGain	250

Figure 3: Final Sensor Model Parameters

2.2 Motion Model

Using the odometry readings from the log, the particle position was updated from the measurements augmented by some Gaussian noise.

alpha1	0.005
alpha2	0.005
alpha3	0.005
alpha4	0.005

Figure 4: Final Motion Model Parameters

2.3 Ray Tracing

Ray tracing was needed to compute the predicted range readings for a given particle. For each angle -90 to 90 degrees about the heading angle, a ray was cast out from the particle through the map grid and terminated when it hit the edge of the map. Figure 5 Shows a comparison between predicted ranges compared to the replayed laser readings. This visualization was part of the verification testing of this module.

Since ray casting was required for each particle, 180 degrees, up to a max distance of 800 grid elements for each iteration, it was the part we were most concerned about affecting runtime. We decided to implement a caching scheme that would memorize ray traced results and greatly reduce repeated computation. We discretized the number of angles to 360 degrees for each of the points on the grid. We could allow the system

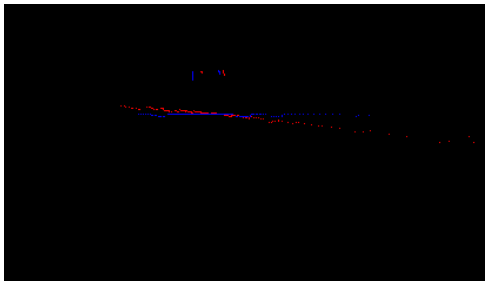

Figure 5: Ray Tracing(Red) Validation against Logged Data(Blue)

to lazily compute the ray traced table but ended up implementing the precomputation for every point in free space before running the localization algorithm.

2.4 Importance Resampling

After a laser reading update, each particle was assigned a weight according to how well it fit the sensor reading. Our implementation used low variance resampling. We used the algorithm laid out by Cyrill Stachniss from his lectures[1]. This technique forces a set of samples to be sampled randomly but at small, regular intervals through the particle weights. This ensures that some particles of low weight still survive and ensures the filter will not converge when the weights are uniform i.e. when no information is gained from the laser reading. This has the added benefit of running in linear time.

3 Results

Our particle filter algorithm converges to an accurate solution when using 3000 particles and the filter parameters described above on all provided data logs. Including visualizations, we get approximately 17 updates per seconds running on a single core of a 4.0 Ghz Intel i7-4790K processor. This performance was deemed acceptable even without parallelization.

4 Future Work

There are several obvious areas to improve our particle filter. The first is parallelization. We could approximately achieve a speed up of approximately Nx using N threads given the particle propagations computations are inherently parallel operations.

More here pending results of extracredit part

5 References

[1] Cyrill Stachniss, https://www.youtube.com/watch?v=eAqAFSrTGGY