Bo-Ray Yeh, Rajiv Bharadwaj
Prof. Dmitry Berenson, GSI: Anthony Liang
EECS 498 Final Project
18 December 2020

Localization

One of the main challenges for a robot while performing tasks is to position itself in its surroundings accurately. Although it can theoretically localize itself given an initial position and subsequent transformation, real world movement inaccuracies cannot be included in these calculations without the use of external sensing techniques.

This is also an important and difficult problem because it involves actions concerning the real world and estimating the environment around the robot. External sensor measurements are never 100% accurate. Even if certain sensors tend to reach this level of accuracy, they lack in terms of their rate of update. Take the example of a car navigating using a GPS. Although the GPS is very accurate, there may be certain environments where the beacon may lose connection with the satellite (under tunnels). In such cases, the car must still detect its position and hence relies on rotary and inertial sensors to minimize the drift.

To tackle this problem, various algorithms were developed that focus on computing a probabilistic estimation of the robot's state at a given time, and update it based on the actions taken and sensor measurements perceived.

FILTERING TECHNIQUES

Various filtering algorithms exist that solve this problem by computing probabilistic estimates regarding the state of a robot at a given time. In this project, we take a look at two algorithms, the Kalman Filter and Particle Filter and compare the performance of the two in several situations simulated with a PR2 robot in an OpenRAVE simulation environment.

Kalman Filter. The Kalman Filter works on the basis of a prediction and correction cycle. It uses a linear motion and sensor model to estimate a gaussian probability distribution of the current state of the robot based on the previous state distribution and sensor measurement.

Particle Filter. The particle filter samples from the previous state distribution multiple times and applies the actions taken by the robot to estimate the current distribution. An important distinction with respect to the Kalman filter is that this algorithm uses the implicit distribution to perform its estimations instead of fitting it to a Gaussian distribution.

IMPLEMENTATION

Based on the advantages and disadvantages of the two filters, we devised a set of paths and environments that can challenge the two algorithms and help us to quantitatively compare the two algorithms. The Kalman Filter algorithm only deals with Gaussian estimations. However, many aspects of the real world aren't normally distributed. Thus, an important test was to compare the performance in a non-gaussian based environment.

We modelled the robot's state, motion, and sensor model through the following equations:

State:

$$x_t = \begin{pmatrix} x \\ y \end{pmatrix}$$

Motion Model:

$$x_t = A_t x_{t-1} + B_t u_t + \epsilon_t$$

where,

$$A_t = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$B_t = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\epsilon_t = \text{motion noise at } t$$

Sensor Model:

$$z_t = C_t x_t + \delta_t$$

where,

$$C_t = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\delta_t = \text{sensor noise at } t$$

The actions taken by the robot are thus described in terms of steps taken in units of distance.

The sensor we are simulating is a global positioning system which returns the robot's position in the xy-plane with noise.

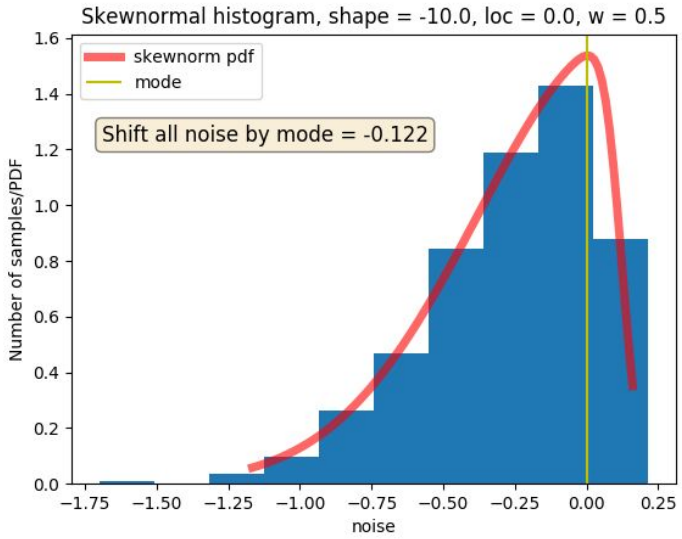
Our tests are comprised of three main steps:

1. Path Generation
2. Addition of Motion and Sensor Noise
3. Running Filtering Algorithms

Path Generation. To generate states ρ_i for each paths, we use an 8-connected A* path planning algorithm supplied with various waypoints to navigate through, and save the path generated in a pickle file. Using the models of the robot described above, we backtrace and compute the actions taken by the robot to achieve the path and add it to the generated pickle file.

Addition of Motion and Sensor Noise. To simulate real world inaccuracies, we take the path planned by A* to generate actions and ground truth state by applying the motion model we assumed previously.

In terms of noise models, we select two cases with Multivariate Normal Distribution and one with Skew-normal. In the skew-normal case, we specify a bias for the random variable by the mode of the distribution to make more sense to a real world's sensor or an actuator's random noise.



Visualization of the Skew-Normal Noise Distribution

The following equations show how to generate actions, ground truth states and measurements:

$$u_t = B^{-1}(\rho_t - A_t\rho_{t-1})$$

$$x_t = A_t x_{t-1} + Bu_t + \varepsilon_t$$

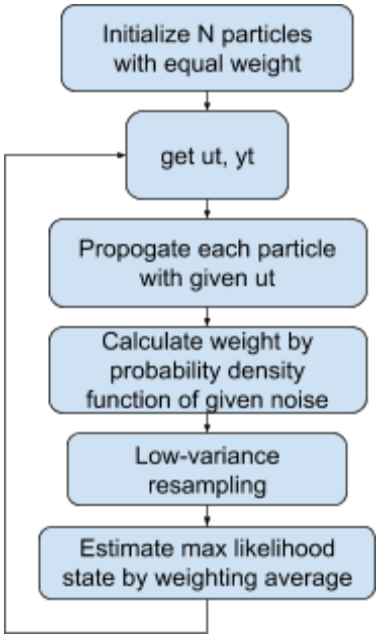
$$y_t = C_t x_t + \delta_t$$

where u : action, x : ground truth state, y *: measurement,*

Note: x_0 is a given starting point

Running Filtering Algorithms. We supply the generated pickle file to a simulator class that also takes a "filter" argument for the test being performed. The sensor measurements and actions taken are used to generate the actual path taken by the robot with the given scenario, and plots it onto the OpenRAVE environment. In addition to this, we also plot the ground truth states and estimations for comparison, and compute the error by taking the norm of the error vector between the ground truth states and estimated states.

1. Kalman Filter: Implemented exactly same as Homework 5
2. Particle Filter:



Appendix A contains information about running the demo script

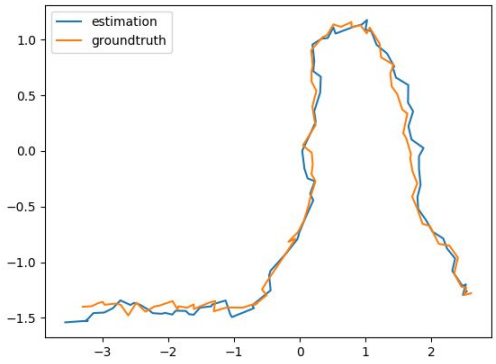
RESULTS

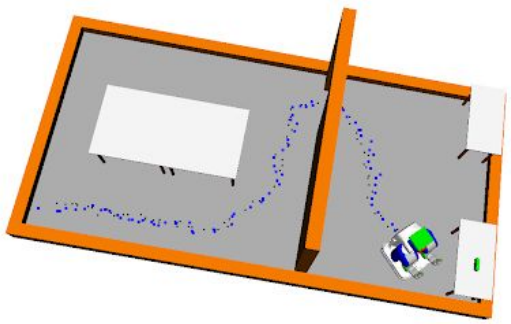
Case 1.

Noise model parameters:

	Noise type	Parameters
Motion noise	Normal Distribution	mean = 0, cov = $\begin{bmatrix} 0.003 & 0.001 \\ 0.001 & 0.003 \end{bmatrix}$
Sensor noise	Normal Distribution	mean = 0, cov = $\begin{bmatrix} 0.02 & 0.01 \\ 0.01 & 0.001 \end{bmatrix}$

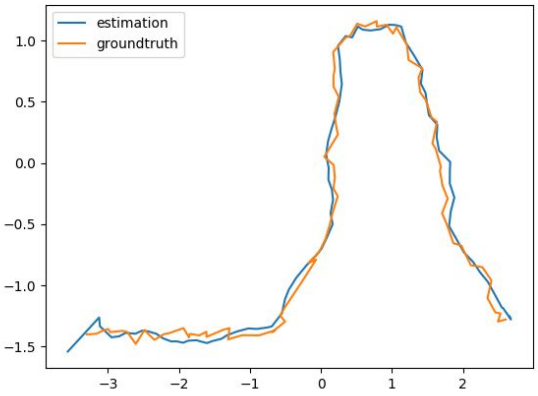
- Result by Kalman Filter:

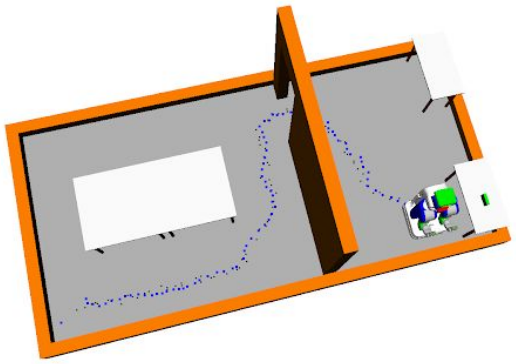




with total norm error = 6.619 and execution time = 0.449 second

- Result by Particle Filter with 2000 particles





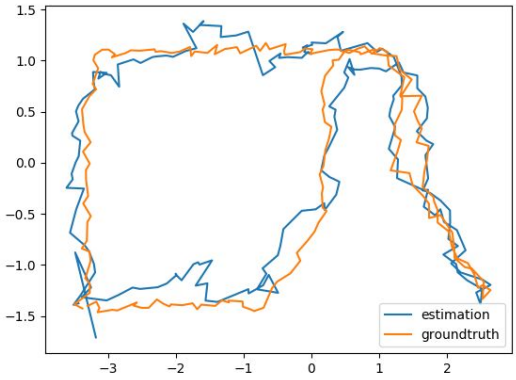
with total norm error = 6.102 and execution time = 457.345 second

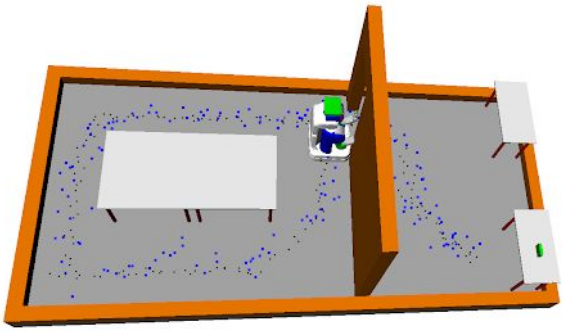
In this case, it's obvious that Kalman Filter is superior to Particle Filter by much less computational cost and outputs reasonable estimations. Particle Filter only decreases total norm error by 0.5 but takes 100 times longer to compute. In this case, it shows that even with small noise, Particle Filter still needs a lot of computation to estimate. Clearly, Kalman Filter is better in efficiency compared to Particle Filter.

Case 2.
Noise model parameters:

	Noise type	Parameters
Motion noise	Normal Distribution	mean = 0, cov = $\begin{bmatrix} 0.003 & 0.001 \\ 0.001 & 0.003 \end{bmatrix}$
Sensor noise	Normal Distribution	mean = 0, cov = $\begin{bmatrix} 1e-2 & -9.99e-3 \\ -9.99e-3 & 1e-2 \end{bmatrix}$

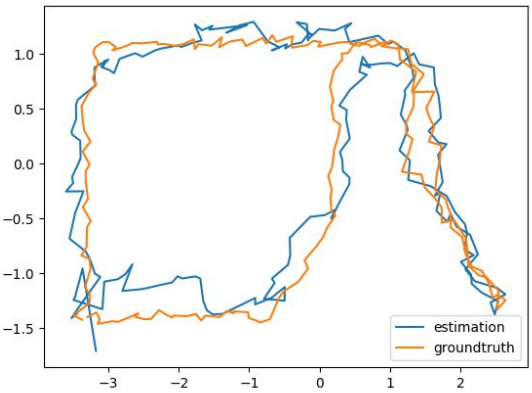
- Result by Kalman Filter:

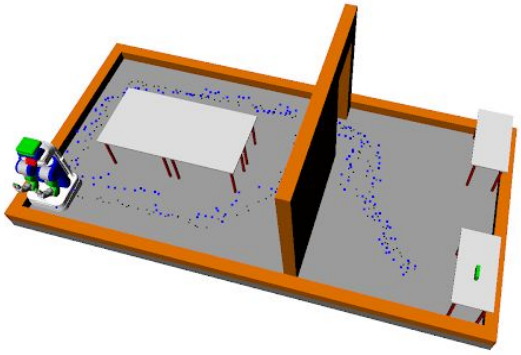




with total norm error = 36.898 and execution time = 0.627 second

- Result from Particle Filter with 5000 particles





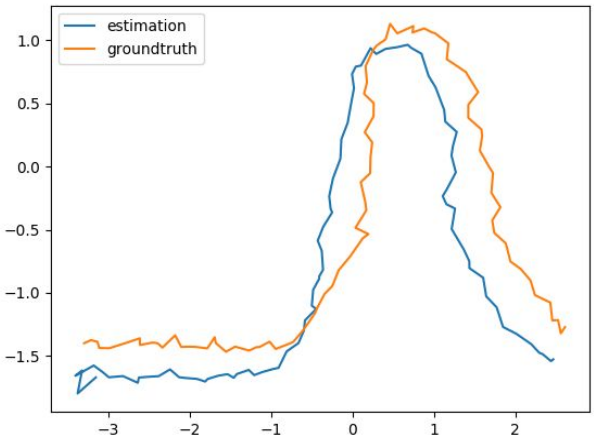
with total norm error = 36.263 and execution time = 1569.248 second

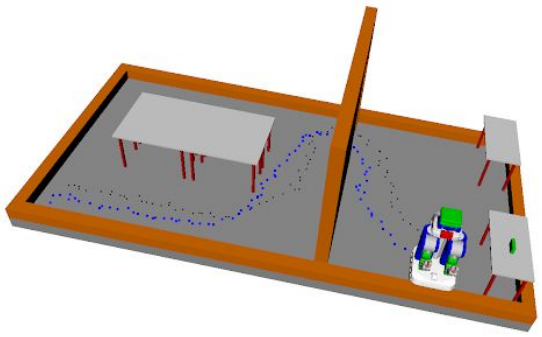
Even after injecting more noisy Gaussian noise, Kalman Filter still outputs a reasonable path. We also needed to increase the number of particles for Particle Filter to 5000 to get a similar accuracy compared to the Kalman Filter, and because of that, it becomes more computationally intense.

Case 3.
Noise model parameters:

	Noise type	Parameters
Motion noise	Normal Distribution	mean = 0, cov = $\begin{bmatrix} 0.003 & 0.001 \\ 0.001 & 0.003 \end{bmatrix}$
Sensor noise	Apply Skew-Normal to each dimension of measurement	shape = -10., scale = 0.5, location = 0.0 with bias -0.122 (mode by this configuration)

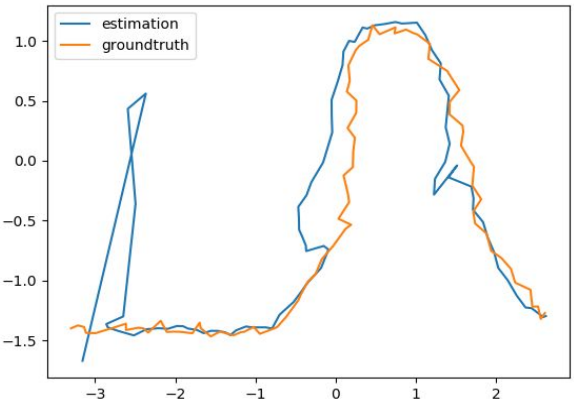
- Result by Kalman Filter

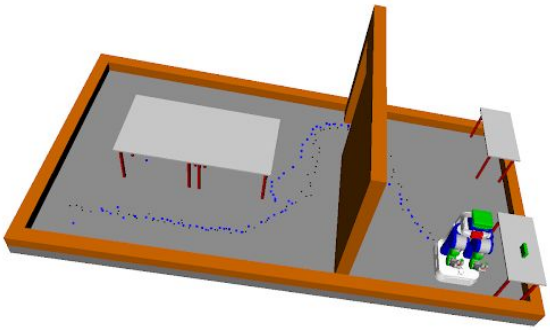




with total norm error = 32.417 and execution time = 0.391 second

- Result by Particle Filter with 10000 particles





with total norm error = 20.758 and execution time = 937.566 second

By Skew-Normal noise, Kalman Filter's estimation completely shifted and output relatively large total norm error. The Particle Filter, however, We can observe that it outputs a large error in the beginning but gradually starts to capture the random distribution at the end. Particle Filter is better in capability to adapt different kinds of random distribution.

CONCLUSION

We implemented a Kalman Filter and Particle Filter to estimate the location of a robot by filtering the measurement taken at each step. The Kalman Filter works accurately and is much more efficient as long as the random distribution of the system is normally distributed. The Particle Filter is slow and computationally intense as the number of particles increases in order to capture the random distribution of motion and the sensor.

However, it performs well with arbitrary noise models as long as we can derive an equation to estimate the probability of measurement with given states. On the other hand the Kalman Filter performs well only for normally distributed motion, sensor measurements and noise with zero mean.

During tuning the parameters, we find that the performance of the Particle Filter is not consistent, it is likely to fluctuate in terms of accuracy even with the same number of particles used on the same dataset. Therefore, it is hard to say how many particles would be required to make Particle Filter perform well enough.

This is consistent with the theoretical operation of the two algorithms we learned in class.

APPENDIX A: Guide to the Codebase

The code can be found at https://github.com/rbharadwaj9/eecs498-localization
It contains a README.md file which has an explanation of what the different files do.

In this section, we discuss the demo.py file which is a demonstration of the path taken by the robot in "data/env2_circle_back.pickle". The output from the simulator is provided in the terminal, OpenRAVE viewer, and a matplotlib viewer.

Terminal Output. Outputs error and time taken to run each simulation. Also mentions whether the estimated path might be in collision. Additionally, there are breakpoints set to play the trajectory estimated from each simulation which require keyboard input from the user.

OpenRAVE Viewer Output. The ground truth is plotted in black. Estimations from the Particle Filter and Kalman Filter are points in blue and green respectively. Points in collision for the Particle and Kalman Filter are in red and purple respectively.
This is also where the estimated path can be played.

Matplotlib Output. There are three figures plotted. The first figure shows the same output from the OpenRAVE environment. However, it's shown in 2 dimensions to make it visually comparable. The second figure shows just the Kalman Filter vs Ground Estimation. The third figure shows the Particle Filter vs Ground Estimation. The color code is the same as the OpenRAVE one.