# EECS 498 Final Report

Bingxin Zhang, Dongqing Xia

December 19, 2017

## 1 Introduction

In order to plan an action, one need to know or at least estimate its state. For example, a robot want to plan a route to goal destination, it needs to localize itself to a starting position (here, position is a state variable). If the variables in a state were discrete, one could model the system with a hidden Markov model. However, the state variables in reality are usually continuous (e.g. position, velocity and acceleration). To discretize state variables will not only increase the size of hidden Markov matrix, but also miss useful information in that continuous space. This report presents experiments on two filtering schemes which help to estimate dynamic continuous condition.

One is Kalman filter, invented by Rudolf E. Kalman. In regular Kalman filter, a system adopts Gauss-Markov model for assumption. Although the true state variables are always unknown and measurements are with white noise, Kalman gain is selected to update previous prediction and minimize the estimation error given the observed measurements in the particular time frame. It has numerous applications in guidance, navigation and control of vehicles and aircrafts. One common example is to estimate position of an inertial navigation system using its own accelerometer measurements and GPS reading. Extended Kalman filter is also introduced when one encounters nonlinear system. Generally, Kalman filter relies on Gaussian assumption, good initial estimates and reasonable transition model of the whole system.

However, life is not easy. One need to localize itself in a more complex environment than that we assume above. Nonlinear behaviors and nonlinear measurements can be hard to linearize. One might not have knowledge about the model of transition process. Noise can be more than white noise. More powerful nonlinear filters such as particle filter can be good candidates to provide good estimations. Particle filter adopts a genetic-like algorithm. It evaluate probabilities of particles (i.e states) after each step. When resampling, particles with larger probability will be selected more frequently, and particles will smaller ones might be abandoned. Finally we adopt a better cluster for the current state estimation. particle filter enable us to adapt to changes in the implicit probability distribution, without giving any fixed assumption about that. Currently particle filter finds its applications significantly in bioinformatics, robotics and other fields involving nonlinearity.

## 2 Implementation

### 2.1 Kalman Filter

#### 2.1.1 Kalman Filter (KF)

Consider a continuous stochastic system:

$$\dot{X} = FX + Bu + W \tag{1}$$

$$Y = HX + V \tag{2}$$

where $X$ represents the states, $u$ is the deterministic input, $w$ is the disturbances that affects the system dynamics. $P$ denotes co-variance of the state variables. Meanwhile, we have $Y$ as observed measurements and $Y$ might not be the same as state $X$. Matrix $H$ denotes the linear transformation from state variables to measurements, while $V$ represents measurement noise. Assume that the disturbance $W$ and $V$ are zeros-mean, Gaussian white noise. We also assume that the cross

correlation between $W$ and $V$ is zero. We should notice that, if $X \in R^{m\times m}$ and $Y \in R^{p\times p}$, the noise intensities in system dynamics $R_w \in R^{m\times m}$ and $R_v \in R^{p\times p}$.

We formulate the optimal estimation problem as finding the estimate $\hat{X}(t)$ that minimizes the mean square error $E\{(X(\text{t})\text{-}\hat{X}(\text{t}))(X(\text{t})\text{-}\hat{X}(t))\}$ given $Y(\tau)\text{:}0 \leq \tau \leq \text{t}$. The intuition is to find the expected value of $X$ subject to the "constraint" observations (i.e. measurements). We also assume the system states transit according to Markov assumption. Finally, we are going to find $\hat{X}(t_k) = E\{X(t_k)|Y(t_k)\}$. It can be viewed as solving a least squares problem and the optimal estimator should be in the following form:

$$\dot{\hat{X}} = F\hat{x} + Bu + K(Y - C\hat{X}) \tag{3}$$

where a optimal gain (i.e Kalman gain) should be:

$$K = -PC^T R_v^{-1} \tag{4}$$

As measurements are sampled at some discrete time, and estimations happen at discrete time, we need to design discrete time Kalman filter. Fortunately, same theory can be applied to the design process. Instead we consider a discrete-time Gauss-Markov model:

$$X_k = F_{k-1}X_{k-1} + B_{k-1}u_{k-1} + W_{k-1}, k = 1, 2, ... \tag{5}$$

$$Y_k = H_k X_k + V_k \tag{6}$$

where: $W_k$ and $V_k$ are independent, white, zero-mean, Gaussian processes. Worth noticing, $F_k$ and $B_k$ can be approximately obtained from discretizing the continuous time system dynamics.

$$X_k = X_{k-1} + \dot{X} \cdot \delta t \tag{7}$$

$$F_k = F_{k-1} + F \cdot \delta t \tag{8}$$

where F comes from Eq.(1), the states transition matrix in continuous system. The question essentially become: (i) How to obtain new state estimates given previous states $\hat{X}_{k|k-1} = E[X_k|Y_{k-1}]$; (ii) How to obtain new estimates given current observations $\hat{X}_{k|k} = E[X_k|Y_k]$

**Algorithm 1** Discrete Time Regular Kalman Filter

**procedure** FILTERING AT T=K

**Input**: setting $\hat{X}_{k-1|k-1}, P_{k-1|k-1}, Y_k$

**Time-Predict**

$\hat{X}_{k|k-1} = F_{k-1}\hat{X}_{k-1|k-1} + B_{k-1}u_{k-1}$

$P_{k|k-1} = F_{k-1}Pk-1|k-1F_{k-1}^T + R_{w,k-1}$

**Measurement-Update**

$K_k = P_{k|k-1}H_k^T(H_k P_{k|k-1}H_k^T + R_{v,k})^{-1}$

$\hat{X}_{k|k} = \hat{X}_{k|k-1} + K_k(Y_k - H_k\hat{X}_{k|k-1})$

$P_{k|k} = P_{k|k-1} - K_k H_k P_{k|k-1}$

**Output**: $\hat{X}_{k|k}$, $P_{k|k}$

In this algorithm, configurations about system dynamics such as $F_{k-1}, B_{k-1}, u_{k-1}, H_k, R_w, R_v$ as assumed to be known.

### 2.1.2 Extended Kalman Filter (EKF)

Consider a nonlinear system:

$$\hat{X} = f(X, u, W) \tag{9}$$

$$Y = g(\hat{X}, V) \tag{10}$$

where $X \in \mathcal{R}^m$, $u \in \mathcal{R}^n$, $Y \in \mathcal{R}^p$, $W$ and $V$ are Gaussian white noise processes with covariance matrices $R_W$ and $R_V$. A nonlinear observer for the system can be constructed by using the process:

$$\dot{\hat{X}} = f(\hat{X}, u, 0) + K(Y - g(\hat{X}, 0)) \tag{11}$$

We can extend design of Kalman gain to a nonlinear system. We linearize around current estimate $\hat{X}$:

$$\tilde{F} = \frac{\partial f}{\partial X}|_{(\hat{X},u,0)} \tag{12}$$

$$\tilde{H} = \frac{\partial g}{\partial X}|_{(\hat{X},0)} \tag{13}$$

Then, we can apply the same analysis to calculate K, P and continue filtering based on these parameters as what we do in regular mode. Same theory also applies to discrete nonlinear system.

**Algorithm 2** Discrete Time Extended Kalman Filter

**procedure** FILTERING AT T=K

**Input**: setting $\hat{X}_{k-1|k-1}, P_{k-1|k-1}, Y_k$

**Time-Predict**

$F_{k-1} = \frac{\partial f}{\partial X}|_{(\hat{X}_{k-1},u,0)}$

$\hat{X}_{k|k-1} = F_{k-1}\hat{X}_{k-1|k-1} + B_{k-1}u_{k-1}$

$P_{k|k-1} = F_{k-1}Pk-1|k-1F_{k-1}^T + R_{w,k-1}$

**Measurement-Update**

$H_k = \frac{\partial g}{\partial X}|_{(\hat{X}_{k|k-1},0)}$

$K_k = P_{k|k-1}H_k^T(H_kP_{k|k-1}H_k^T + R_{v,k})^{-1}$

$\hat{X}_{k|k} = \hat{X}_{k|k-1} + K_k(Y_k - H_k\hat{X}_{k|k-1})$

$P_{k|k} = P_{k|k-1} - K_kH_kP_{k|k-1}$

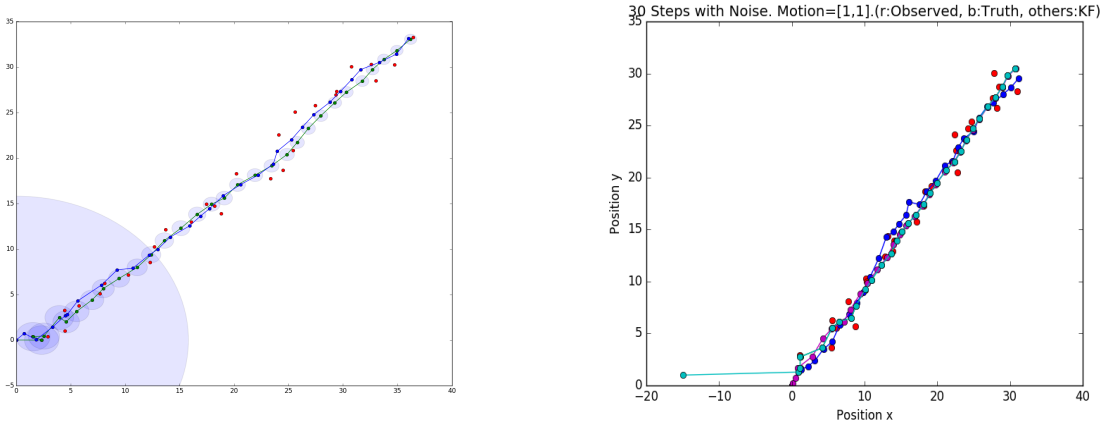
**Output**: $\hat{X}_{k|k}$, $P_{k|k}$

### 2.1.3 Kalman Filter Example

Let say we want to utilize GPS reading to estimate the position. X contains variable. (x,y) in 2d environment. Each time step, one can directly move by a motion u, but subjected to some noise (e.g. a robot can forward by one step but not exactly know the metrics). Sensor gives noisy observation on (x,y). We adopt easy linear model for testing the basic functionality of Kalman filter. $F$=$I_2$, $H$=$I_2$, $P0$=$I_2$*1000 (to model a not so sure about starting place scenario). Mean of motion is (1,1) and the motions are subject to a Gaussian noise with zero mean and variance of 0.1. In this way, we set $W$=$I_2 * 0.01$. And observation convariance matrix $V$=$I_2 * 1$ since measurements are assumed to be unbiased and subjected to Gaussiance noise with covariance to be 1. The estimated curve has slope close to 1, and the estimated points mostly get close to the truth.

Then we test functionality in other scenarios. When we have a wrong initial estimate, we can re-construct a estimate good estimate by setting initial $P0$ to be a matrix with large number on diagonal entries, and $V$ to be a matrix with relative small diagonal elements. Ituition is that we say we are not sure about the initial condition and we will trust our GPS because it has small variance in measurement data. The starting point of light blue curve is deviated, while the curve converges to the truth quickly. Another case is that we use same $W$ and same measurement noise, but model

the measurement noise with a wrong model which means $V$ is set to be $I_2 * 100$. Estimation is still not bad (purple points shown in **??**). By looking to equation 4, intuition is that we try not to believe what we get in measurement data (since $R_v^{-1}$ is small now, so update from measurements will have small weights), and we trust more in estimates from state transition model. Results are shown in in figure 1.




(a) Uncertainty decreases throughout the process.

(b) 3 different settings about Kalman filters.

Figure 1: Regular Kalman Filter Example. Robot utilizes GPS measurements to localize itself. Blue: True location, red: noisy observed location for measuring the distance to (0,0), green: original and true modelling filter output, same in (a) and (b). (b) Purple: true initial location, with larger $V = I_2 * 100$ though true noise doesn't increase; light blue: deviated initial position, but with smaller $V = I_2 * 0.1$.

Let say, robot is placed in a indoor environment. GPS is no long reachable. Now, robot can have a distance sensor which gives the noisy estimation on how far it's away from one beacon (e.g. a fixed location $(x_0, y_0)$) in a certain room. Now there is a nonlinear sensor model and we can locally linearize the sensor model as following:

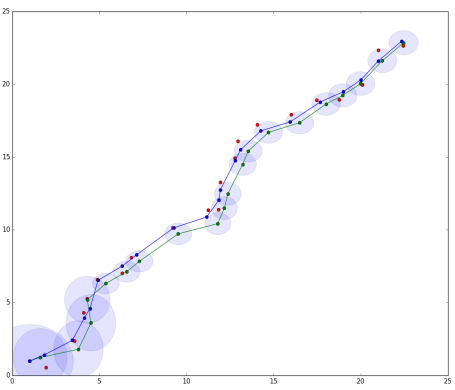
$$d = \sqrt{(x_k - x_0)^2 + (y_k - y_0)^2} \tag{14}$$

$$H = \frac{\partial g}{\partial X}|_{\hat{X}_{k|k-1}} = [\frac{x_k - x_0}{d}, \frac{y_k - y0}{d}] \tag{15}$$

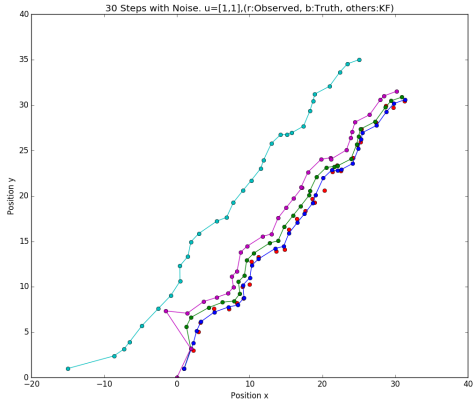
We set $P0 = I_2 * 10$ this time, making robot have more confidence on its initial position. Other configurations are kept the same with regular Kalman experiment. Results are showed in figure 2. The estimated curve has slope close to 1, and the estimated points mostly get close to the truth. Then we test functionality in other scenarios. Unlike the regular Kalman cases, when we have a wrong initial estimate, we set $V$ to be a matrix with relative large diagonal elements. By looking to equation 4, intuition is that we place smaller weight on measurement updates and we stubbornly trust the wrong estimates from transition model. Light blue points in figure **??** shows a deviated curve in this case. Another case is that we use same $W$ and same measurement noise, but model the measurement noise with a wrong model which means $V$ is set to be 0.1. Estimation is not bad (purple points shown in **??**). By looking to equation 4, intuition is that we try not to believe what we now place a large weight on measurement updates than it should be. But since we start from the correct initial place, the estimates from state transition model are close to true locations. It's reasonable to see purple points are slightly diverted from the true path in blue, and perform poorer than EKF with good modelling (in green).

## 2.2 Particle Filter

Particle Filter is a sampling method generally used in localization implementations. By sampling the whole environment, then re-sampling based on predicted possibilities, particle filter can converge



(a) Uncertainty decreases throughout the process.




(b) 3 different settings about Kalman filters.

Figure 2: Extend Kalman Filter Example. Distance measurements to a fixed point to localize itself. Blue: true location, red: noisy location for distance measurements, green points: original and true modelling filter output, same in (a) and (b). (b) Purple: true initial location, with larger $V$=0.1, though true noise doesn't decrease; light blue: deviated initial position, with smaller $V$= 100 though the true noise doesn't increase.

to a most likely location with high accuracy. But particle filter has possibility of losing the true location by sampling and the convergence speed is generally slow. The overall algorithm is shown in Figure 3.

In particle filter implementation, the start location can be whether known or unknown. Particle has its significant advantage in solving kidnapped robot(unknown location) problem. For both preconditions, our algorithm has different *Action*, *Sensor* and *Weight* functions. All other parts are same. We use the same *Action* and *Sensor* functions with the functions used in Kalman Filter so as to compare their accuracy. In kidnapped condition, we use different functions and calculation methods, which will be shown in Section 2.2.2. For initialization part, which is the first line in Figure 3, an example is shown in Figure 4. By "not knowing the initial position", We assume the whole environment has an uniform distribution and sample on valid positions (neither colliding with the environment nor being out of the room). This example denotes Kidnapped robot condition exactly.

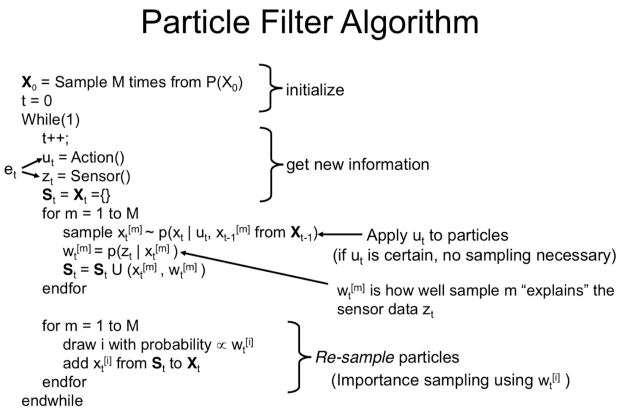



Figure 3: Particle filter algorithm. Information retrieved from *Kalman Filters and Particle Filters* in course 498-006: Introduction to Algorithmic Robot (2017 Fall).

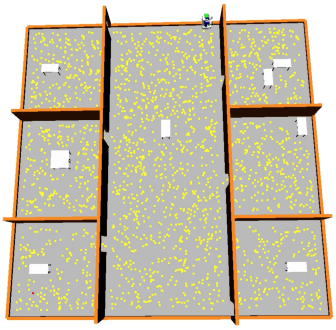


Figure 4: Particle Filter with unknown start Location. This example denotes initial sampling condition by assuming all valid places have equal probability.

### 2.2.1 Known Start Location

We use the same *action* and *sense* function for Kalman Filter and Particle Filter. When the robot know its initial location, goal of the filter is to find an accurate location estimate when the robot has stochastic actions. By the term "stochastic", we mean the action is within a certain bound, for example, the mean of a step is about 1. But the action is subjected to some random small noise. The *action* function contains both an A* path and randomly selected heading. The action with randomly selected heading is executed in about 15% of running time. 15% is a reasonable error, which can show the filter's functionality clearly. The function is shown in Listing 1

```
#location is current robot true location, heading is a chosen direction
def Action(location, heading, error = 0.08):
    #Next Location
    target_location = location + heading

    #possibility to have an error target location
    if (random.uniform(0, 1) < error):
        choose a neighbor point other than the target_location

    #Check whether the target location is valid.
    #If not valid, stay
    if (is_collision(target_location) or is_out_of_bound(target_location)):
        return true_location, True #origin location, hit_wall
    else:
        return target_location, False #next location, not_hit_wall
```

Listing 1: Particle Filter Action Function for Known Start Location Robots.

Every 30 iterations, we alternate the route. We choose to use A* path because robot is placed in a complex environment. Robot hardly get out of a room through a small door by just randomly selected headings and step forwards. We randomly generate a valid target point outside the initial small room, and utilize A* to search a path in 2d setting. This is a method to generate a reasonable path for simulation. And we will not send this path information to robot's estimation process so robot still knows little about its exact position.

The *sense* function is implemented by choosing random points around the true location within a noise range. This can make the sensed location not so far away from the true location but has some reasonable errors. The *weight* function is based on the sensed location. The function used is shown below.

$$w_t^{[m]} = \frac{1.0}{||z_t - x_t^{[m]}|| + C}$$

Here $z_t$ is the sensed location. $x_t^{[m]}$ is the sampling location. $C$ is a constant to avoid invalid denominator. When samples are quite close to what we observe, we consider them with a larger weight in location estimation, we consider them with more probability to be true location. When

samples are far away from observation, they get lower probability instead of a zero. In implementation, we use $C = 0.1$ to make the points which are close to the sensed position not getting a too large weight (sometimes the sensed location may have large error). We also don't want the result depends heavily on the sensed location.

### 2.2.2 Unknown Start Location(Kidnapped)

When the robot does not know its initial location, it thinks it can be anywhere in the environment. Example is shown in Figure 4.

The *action* function used for kidnapped robots is shown in Listing 2. To enhance the success possibility, we let the action to move totally based on the heading vector. If the heading is not valid, stay in the same location and randomly select another heading.

```
#location is current robot true location, heading is a chosen direction
def Action(location, heading):
    #Next Location
    target_location = location + heading

    #Check whether the target location is valid.
    #If not valid, stay
    if (is_collision(target_location) or is_out_of_bound(target_location)):
        return true_location, True #origin location, hit_wall
    else:
        return target_location, False #next location, not_hit_wall
```

Listing 2: Particle Filter Action Function for Kidnapped Robots.

The heading is chosen by two methods. The first one is to randomly pick a heading, walking in that heading until the robot hit the wall or get out of the bound. After a hit, choose a heading randomly again. But this method can not go through a door in a complex environment. So we use A* algorithm to find a way to a random selected target location in that environment. A* can go through a door without collision. However, collision information is also important for robot to filter out those non-colliding samples ( it is extremely useful when having a large amount of sample points). In this reasoning, we take turn to utilize these two methods to generate path. At the beginning, use the colliding-preferred policy (where robot goes towards the wall or obstacle). It helps reduce a lot of invalid samples.

The *sense* function is shown in Listing 3. The *sense* function is similar to sonar sensors. It is used when the total number of sample points is in a considerable range (here, typically less than 200). If robot owns too much sample points, the speed of the estimation algorithm will be too slow. This function can return distances in eight directions between current location and the environments obstacles (including walls, boundaries and objects). By multiplying a random number to the true distance at the end of *sense* algorithm, we stimulate the noisy sonar measurements.

```
#8-connect heading in 2-D
heading = [[1, 0], [0, 1], [-1, 0], [0, -1],
          [1, 1], [-1, 1], [1, -1], [-1, -1]]

#location is current robot true location, M is total points num
def Sense(location, M):
    distance = []
    #if M > MaxNum, we do not use this sense function. Use is_wall returned by
    Action function to evaluate.
    if M > 1000:
        return distance

    #Each step
    s = 0.5
    global heading
    for h in heading:
        step = [h[0] * s, h[1] * s, 0.0]
        start = 0.0
        # if not collide, continue.
        # Find the distance with the closest obstacle in the heading direction.
        while(not is_collision(location + start * step)):
```

```
            if (is_out_of_bound(location + start * step)):
                break
            start += 1.0
        distance.append(start * s)

    # eight directions, generate sensor error
    for i in range(8):
        # have 10 percent possibility of error.
        # (Possible through walls or out of bound)
        if random.uniform(0, 1) < 0.1:
            distance[i] *= random.uniform(1, 1.5)

    # Use the sensed distances in 8 directions to evaluate
    return distance
```

Listing 3: Particle Filter Sense Function for Kidnapped Robots.

After sensing, we need to calculate the weight of each sample points. The pseudocode of *weight* function is shown in Listing 4. The equation used for method 2 is shown below. $C$ is a scale number in case of the distance is 0. The number shown in *weight* function is the optimal value after tested. AS we mention above, method 1, which utilizes the collision information (actually, it means "just facing the wall and stay there"), can reduce significant amount of invalid samples. After that, method 2 can get much more accurate weights values within a reasonable computation time. Particles with higher differences with the true position sensed distances, get lower weight. They will then have lower possibility to be resampled and thus higher possibility to be abandoned. For smaller scalar constant $C = 0.00001$, we can get large weight for the correct location.

$$w_t^{[m]} = \frac{1.0}{||z_t - x_t^{[m]}|| + C}$$

Here $z_t$ is the eight directions' sensed distances between the robot and obstacles. $x_t^{[m]}$ is the sample point's sensed distances in eight directions.

```
#true_distances = sensed true location distances returned by Sense()
#xm_distances = sensed point Xm distances returned by Sense()
#is_wall = true location collision returned by Action()
#xm_is_wall = point Xm collision returned by Action()
def Weight(true_distances, xm_distances, is_wall, xm_is_wall):
    #Weight will be normalized after all points weight calculated.

    #method 2: using distances returned by Sense()
    if (len(xm_distances) > 0):
        #norm the distances difference and scale
        #larger distances diff, smaller diff.
        return 1.0 / (sum(abs(true_distances - xm_distances) ** 2) + 0.00001)

    #method 1: using is_wall returned by Action()
    if is_wall = xm_is_wall:
        #if same, has high possibility
        return 1.0
    else:
        #if wrong, nearly no possibility to be this point
        return 0.005
```

Listing 4: Particle Filter Weight Function for Kidnapped Robots.

Particle filter has a problem during stimulation. During the early stage, a lot of particles have similar weights. Good estimation can be neglected in *Resample* process due to randomness. The almost true estimations can't be re-construct afterwards. Initially, we found that around 87 percent of trials fail due to this reason. We solve this problem of generating particles around a point with some amount of neighbors. The number of particles to be added, is positively related to weight of that point. The higher weight the point has, the more neighbouring particles will be added. Even if the almost true estimation is lost, this policy gives us a chance to re-construct it and evaluate the samples in next iteration. Meanwhile, a point, or a small group of prints with higher probability

will always be chosen by the *Resampling* function. It makes probable estimates shrink to very few estimates (*diversity* decreases). These points again have possibility to be lost in the next iteration due to randomness. We choose to add neighbouring particles around that particular sample (and make it become a cluster again). We don't simply add the same point. Instead, we make it a *diverse* cluster. An example of this re-sampling method result is shown in Figure 5. In the figure, robot's estimation is close to true location. Green lines denotes distances returned by the Sense function. Yellow points are current possible points in clusters. More points in the cluster, higher the possibility of the mean location in that cluster. After applying this method, we get better success rate.

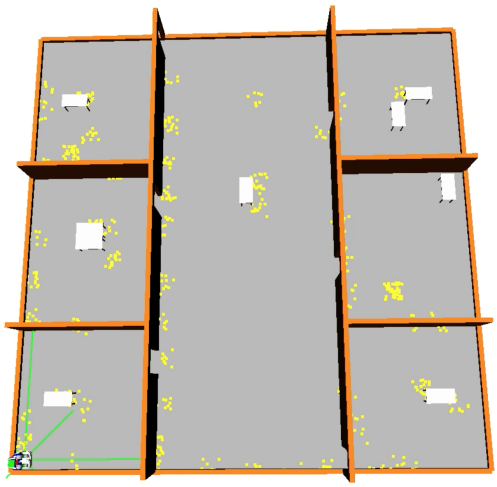


Figure 5: Particle Filter Kidnapped Implementation Example.

In conclusion, the whole implementation process of particle filter and its stimulation are shown in the flowchart in figure 6.

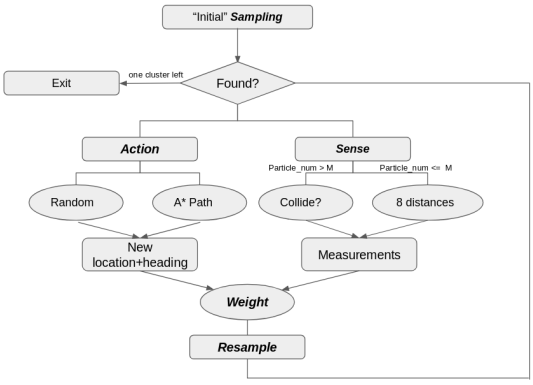



Figure 6: Particle Filter Diagram for Kidnapped Robot. Phases in Italic and bold are key function components mentioned above. Also, *Sense* function gives different types of observations based on the total number of particles to achieve a faster computation sensory process. M here is set to be 200 after experiments.

# 3 Results

In this section, we integrate two filtering algorithms on OpenRave. We test them in three types of environments: asymmetric, empty and symmetric ones. The asymmetric environment has two rooms with tables and open space. It's an easy environment to test the functionality of Kalman

filter and particle filter. The empty environment is friendly to Kalman filter because the estimations will not easily collide with any obstacle. It requires longer time for particle filter to finalize a good estimate. The symmetric one has 6 rooms mirrored to the central axis. The differences among the rooms are the number and position of the tables inside. It is easy for robot to get collision, so we require a more accurate estimate in this environment. It turns out that, Kalman filter frequenetly generates some estimates either being outside the room or colliding with the walls. But particle filter perform better than that, and that is related to particle filter's nonlinear assumption and implementation. The environments' sizes are $20 * 20$ in the Openrave settings. Each axis has a range of $[-10, 10]$.

We test Kalman filter, extended Kalman filter and particle filter in same environment with same action and *sense* function in each experiment. Initially, the robot can know or not know its start position. As for the case where robot know its initial starting place, we test Kalman filter, extended Kalman filter and particle filter together. The mean of a step is 1. The possibility of error in action is 0.08. The sensor's noise is 1.0 in normal distribution. Particle filter starts with 500 sampling points. For kidnapped robot, we only test the particle filter (because one of important assumption in Kalman filter is that one needs good initial estimate). The action has no error. The sonar sensor (8 directions obstacle detector) has 0.1 possibility to return erroneous distances estimates. The erroneous distance will be scaled by a number randomly. This scaling constant ranges from 1 to 1.5. We try to enhance the success rate and also maintain possible errors to stimulate in a realistic scenario. The experiments setting and results are shown below in sub-sections. We focus on the overall performance accuracy by plotting the estimated locations. Computational time required in each iteration is also recorded for comparison. As for the error, we care how often the filters give invalid locations (either colliding with objects or being out of boundary); and we also care the mean square error between ground truth and our estimates per iteration. We will detailize the results in Section 3.3 and summarize the performance in Section 3.4.

## 3.1 Asymmetric Environment

The asymmetric environment is shown in Figure 7. There are two rooms, walls and tables. The environment is not too complicate for particle filter because the asymmetric measurements give great information to filter invalid samples. It's not complicate for Kalman filter because there is open space and Kalman filter estimates will not easily collide with obstacles or boundary. Overall, it is a easy setting for two filters to help localize the robot.

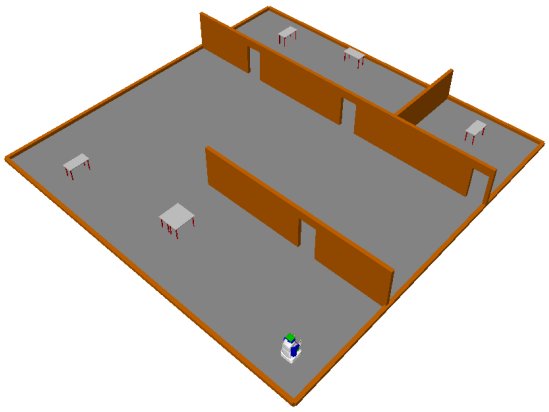


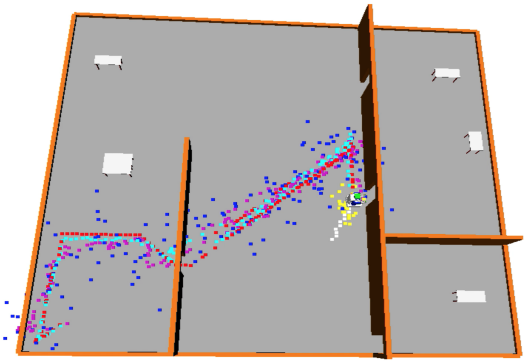
Figure 7: Asymmetric Environment.

### 3.1.1 Kalman Filter (KF) v.s. Particle Filter

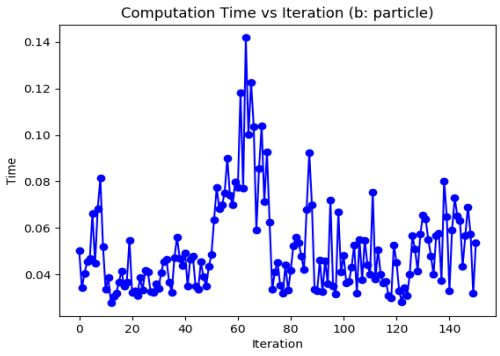
As mentioned above, action model and sensor model are nonlinear with non-Gaussian noise. So we need to conduct experiments to estimate the covariance matrix. We still assume there are zeros on off-diagonal entries (because the randomness in each channel is not correlated.) Set system

dynamics have $W$ with 1 on diagonal entries. By making up a numpy array with size of 1x1000 and its entries range from -10 to 10 (room size), we add an Gaussian noise with zero mean and variance to be 1, and re-calculate the variance of the whole array now. It seems to be about 40. In this way, we set measurement covairance matrix to have 40 on diagonal entries. As for the state covariance, initial $P$ is set to be $10 * \mathrm{I}_2$.

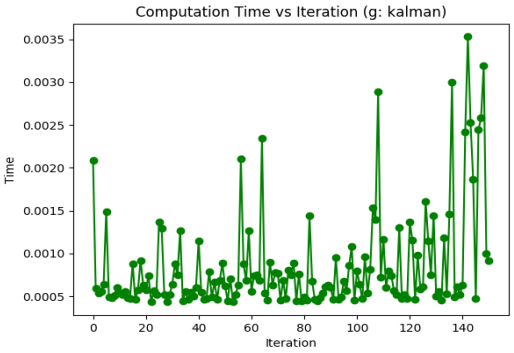
The simulation result is shown in Figure 8. In figure 8a, blue points are sensed locations, red points are true locations, yellow points are current particle filter's points, purple points are Kalman filter estimates and light blue are generated by particle filter. Even though observation gives noisy data, both particle filter and Kalman filter generate reasonable estimations. As for computational time, in fig 8b, particle filter uses about 0.14s every iteration and it even requires more time when it computes the distances to obstacles and evaluates the probabilities for all particles in sonar stimulation (this happens when total number of particles is less than 200). As for Kalman filter, its computation only involves matrices of small size. The process mainly ends within 0.003 second every iteration. As for the estimation error in 8d, Kalman filter and particle filter have bounded stablized error, which are smaller than the error in observations. This shows the improvement made by two filters.



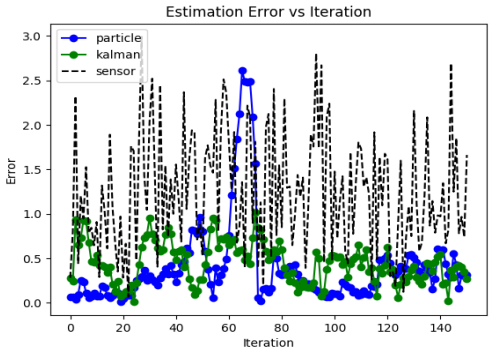
(a) Simulation Result.




(b) Particle Filter's Computational Time




(c) Kalman Filter's Computational Time




(d) Estimation Error of Both Filters

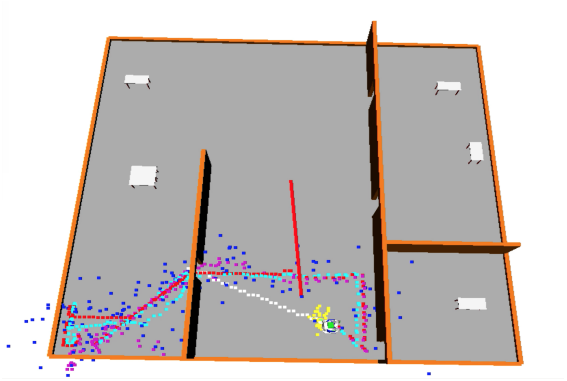
Figure 8: Kalman Filter v.s. Particle Filter in Asymmetric Environment. a) Blue points are sensed locations, red points are true locations, yellow points are current particle filter's points. As for estimation, purple points are generated from Kalman filte while light blue are generated from particle filter. White trajectory is planed A* path for getting to next target point.

### 3.1.2 Extended Kalman Filter (EKF) v.s. Particle Filter

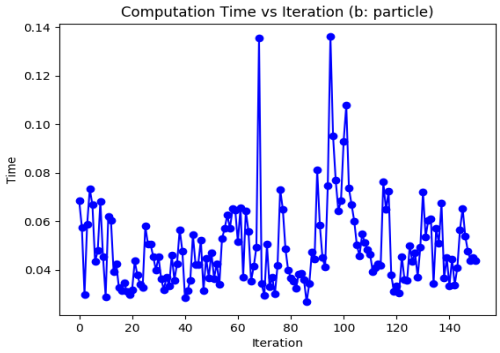
Functions are the same, expect for the measurements are distances to a fixed point (0,0) now. In this scenario, let say there is an underground beacon at (0,0,-10). Our robot carries a signal receiver which receives a rough estimation about horizontal distance to that beacon. We stimulate a nonlinear sensor model and use extended Kalman filter to localize. The derived formula for

linearized $H$ is described in EKF section. Keep $P$, $W$ the same, but only change $V$ to be 1 by 1 matrix now (diagonal entry still be 40). As for the particle filter, we don't change any utility.

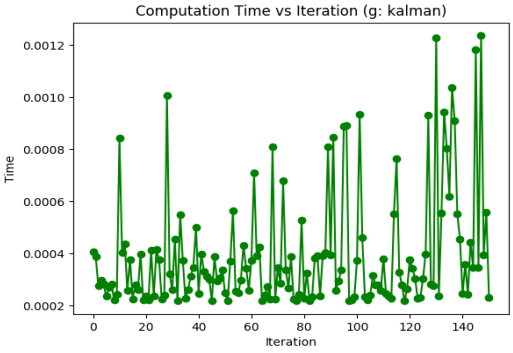
The simulation result is shown in Figure 9. In fig 9a, we notice that when robot start from the left lower corner and walk along the wall, some purple points are out of boundary. This shows Kalman filter's limitation; while the curve estimated by particle filter has a much lower deviation from the true path. The conclusion about computational time is similar to that of previous subsection. Particle filter uses more time per iteration, and computational time rapidly increases around 100 iterations, because robot now turns on its sonar sensors. But then it decreases because the number of particles maintained by the robot dramatically decreases. But overall, two filter perform well.



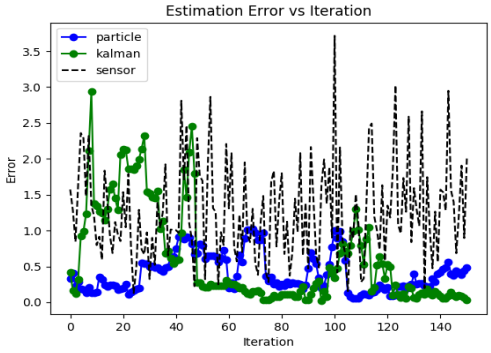
(a) Simulation Result.




(b) Particle Filter's Computational Time




(c) Extended Kalman Filter's Computational Time




(d) Estimation Error of Both Filters

Figure 9: Extended Kalman Filter v.s. Particle Filter in Asymmetric Environment. a) Red points are ground truth, purple points are extended Kalman filter results. Red line shows the connection to point (0,0,0) and noisy observed location (showed in dark blue points), and robot receives this erroneous distance measurements and begins its extended Kalman filtering process. As for estimation, purple points are generated from Kalman filte while light blue are generated from particle filter. Yellow point (beneath the robot now) shows the maintaining cluster from particle filter.

### 3.1.3 Particle Filter(Kidnapped Robot)

The simulation result is shown in Figure 10. After several iterations, robot can localize itself. Since the environment is asymmetric and sonar gives informative suggestions on valid particles, the localization usually ends within 2 3 iteration of sonar measurements.

## 3.2 Empty Environment

The empty environment is shown in Figure 11. The environment becomes "hard" for particle filter because measurements becomes somewhat similar among particles. There is fewer information

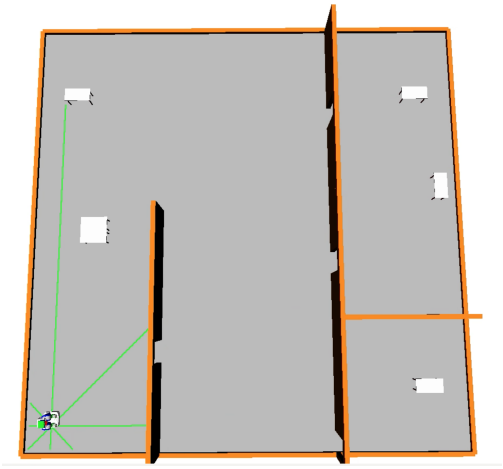


Figure 10: Asymmetric Environment Kidnapped Robot Particle Filter Example.

to filter invalid samples. It's not complicate for Kalman filter because there is open space and Kalman filter estimates will not easily collide with obstacles or boundary.

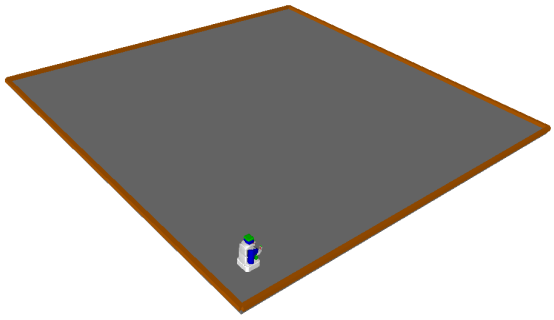


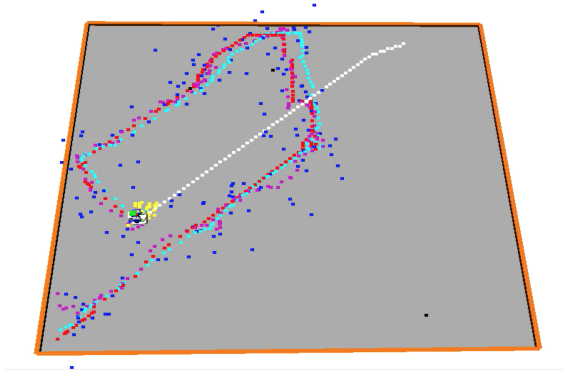
Figure 11: Empty Environment.

### 3.2.1 Kalman Filter (KF) v.s. Particle Filter

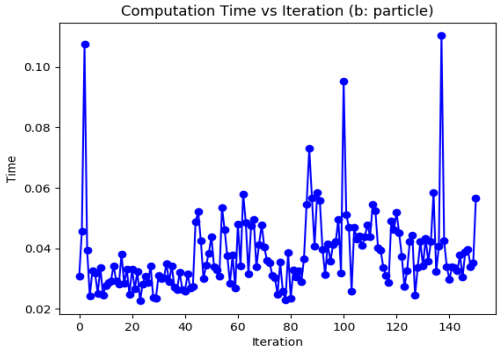
The overall settings for two filters are the same with experiment described in section 3.1.1. The simulation result is shown in Figure 12a. The conclusion about computational time is similar to that of previous subsection. In fig 12d, two filters perform well by generating better estimates whose error is smaller than that from noisy observation.

### 3.2.2 Extended Kalman Filter (EKF) v.s. Particle Filter

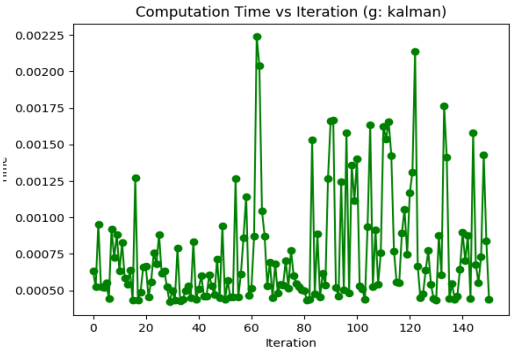
The overall settings for two filters are the same with experiment described in section 3.1.2. The simulation result is shown in Figure 13. The conclusion about computational time is similar to that of previous subsection. In fig 13a, we notice that purple points coincide with ground truth but diverge from ground truth when the robot is close to the fixed point (0, 0). This happens because (i) Kalman filter has good initial estimate at the beginning, (ii) the linearization of $H$ will be relatively sensitive to deviation when it's near the fixed position. In other words, if we are one step away from the true location (5,5) and true location (1,1), the $\delta H = g(\cdot) - \tilde{H}$ result from error given true location to be (5,5) is smaller than that from (1,1). [The absolute value gradient



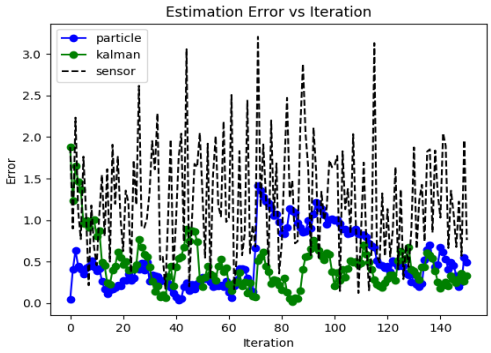
(a) Simulation Result.




(b) Particle Filter's Computational Time



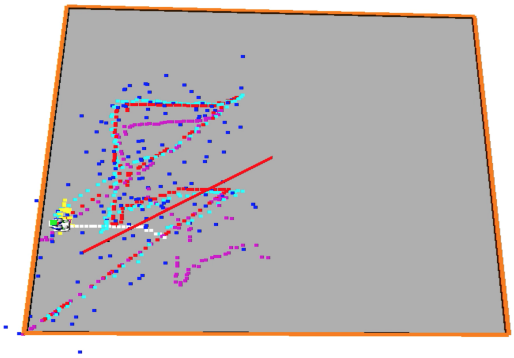

(c) Kalman Filter's Computational Time



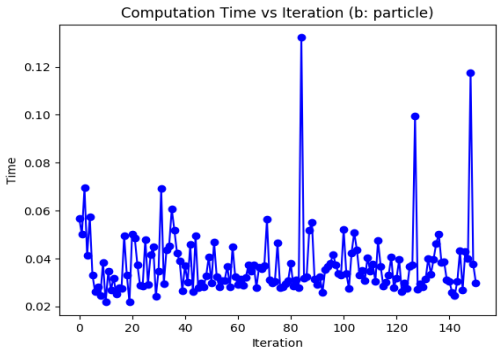

(d) Estimation Error of Both Filters

Figure 12: Kalman Filter v.s. Particle Filter in Empty Environment. a) Blue points are sensed locations, red points are true locations, yellow points are current particle filter's points, purple points are Kalman filter estimates and light blue are particle filter estimates. White trajectory is planed A* path for getting to next target point.

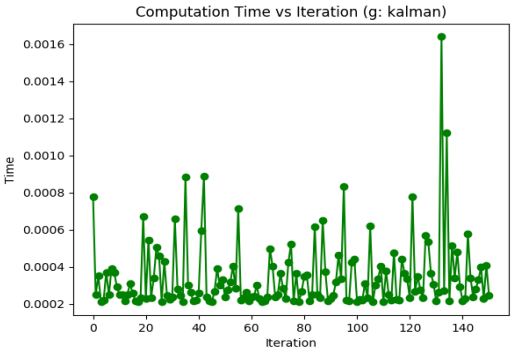
of $(g(\cdot))]$ at (1,1) is larger]. Also, the error from extended Kalman filter is larger than that from regular Kalman filter. In fig 13d, mean square error of Kalman filter goes to near 5 near the $40^{th}$ iteration, probably because robot gets close to the fixed beacon. Error of particle filter maintained at a small value throughout the process. Overall, in fig 13a, even if the red line shows distance measurement is noisy, robot can get a good estimate from particle filter and extend Kalman filter (as long as the robot gets away from the origin).



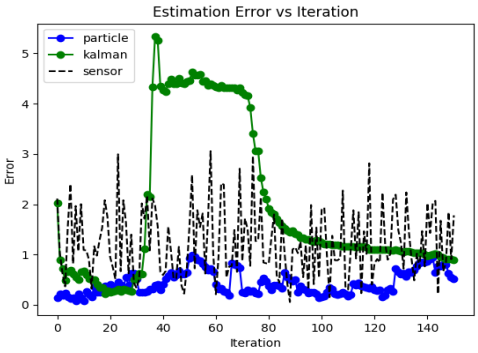
(a) Simulation Result.




(b) Particle Filter Time




(c) Extended Kalman Filter Time




(d) Estimation Error of Both Filters

Figure 13: Extended Kalman Filter v.s. Particle Filter in Empty Environment. a) Red points are ground truth, purple points are extended Kalman filter results, light blue are particle filter result. Red line shows the connection to point (0,0) and noisy observed location (showed in dark blue points), and robot receives this erroneous distance measurements and begins its extended Kalman filtering process. Yellow point (beneath the robot now) shows the particle filter estimate.

### 3.2.3 Particle Filter

The overall settings for two filters are the same with experiment described in section 3.1.3. The simulation result is shown in Figure 14. After several iterations, robot can localize itself. Since the environment is empty and points mirrored to origin get similar measurements in sonar stimulation, the localization usually ends at around 20 iterations. One can notice robot needs to explore with more steps to localize itself than the scenario of an asymmetric room.

## 3.3 Symmetric Environment

The symmetric environment is shown in Figure 15. There are six rooms, walls and tables. Each room has difference on table number and its orientation. Also, the door size is the same but the place with respect to center of the room is different. The environment becomes complicate for particle filter because symmetric measurements give less information to filter invalid samples. It becomes complicate for Kalman filter because there are narrow passages. Kalman filter estimates

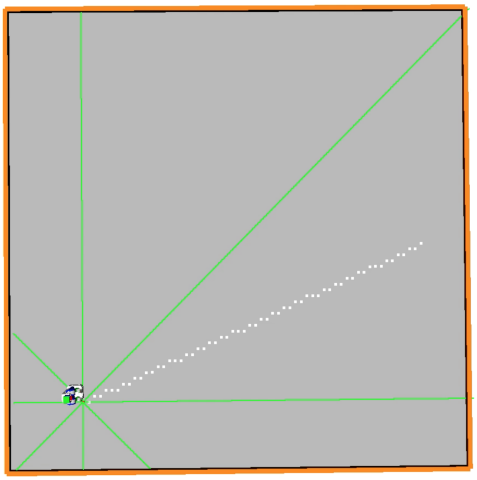


Figure 14: Empty Environment Kidnapped Robot Particle Filter Example.

will be easy to collide with obstacles or boundary when robot want to pass through the door. Overall, it is a difficult setting for two filters to help localize the robot accurately.

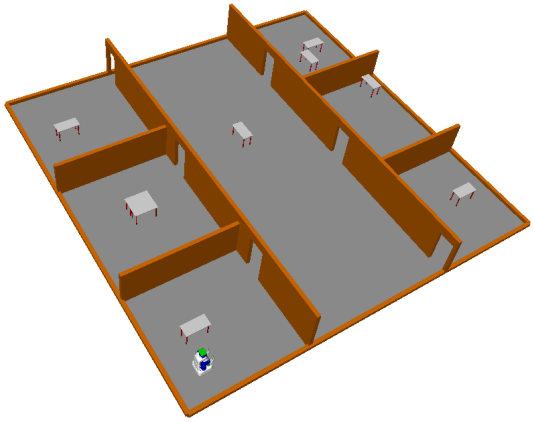


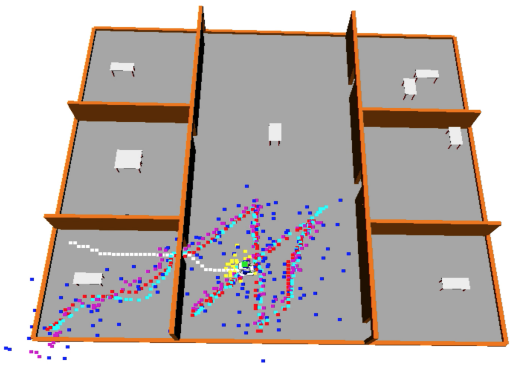
Figure 15: Symmetric Environment.

### 3.3.1 Kalman Filter (KF) v.s. Particle Filter

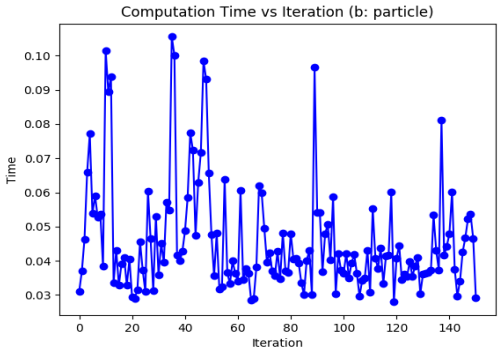
The overall settings for two filters are the same with experiment described in section 3.1.1. The simulation result is shown in Figure 16. In figure 16a, notice when robot walk along the wall Kalman filter generate some out-of-boundary estimate, while particle filter always generate valid estimation. The conclusion about computational time is similar to that of previous subsection. In fig 16d, two filters perform well by generating better estimates whose error is smaller than that from noisy observation.

### 3.3.2 Extended Kalman Filter (EKF) v.s. Particle Filter

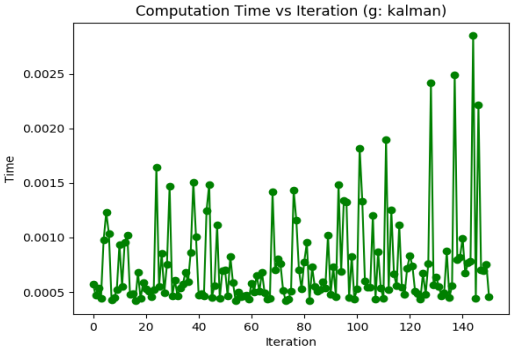
The overall settings for two filters are the same with experiment described in section 3.1.1. The simulation result is shown in Figure 17. The conclusion about computational time is similar to that of previous subsection. In fig 17d, two filters perform well by generating better estimates whose error is smaller than that from noisy observation. This time, robot select a path quite far away



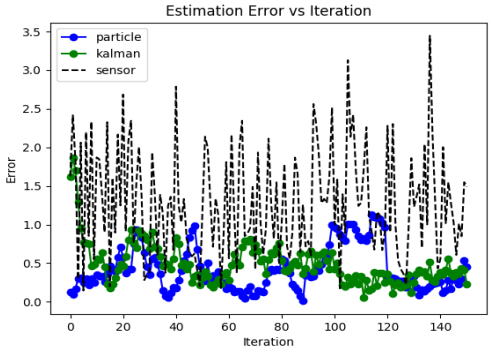
(a) Simulation Result.




(b) Particle Filter Time




(c) Kalman Filter Time




(d) Estimation Error of Both Filters

Figure 16: Kalman Filter v.s. Particle Filter in Symmetric Environment. a) Blue points are sensed locations, red points are true locations, yellow points are current particle filter's points, purple points are Kalman filter estimates and light blue are particle filter estimates. White trajectory is planed A* path for getting to next target point.

from the origin in the environment, so extend Kalman filter performs well throughout the process except few out-of-boundary estimates.

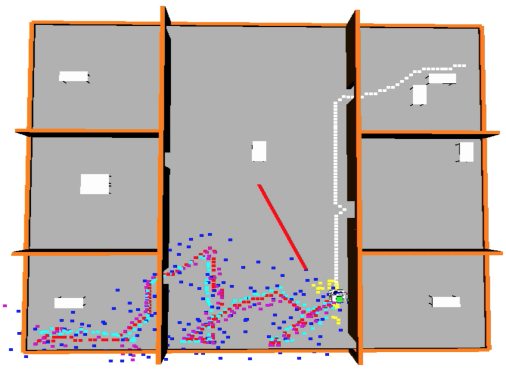
### 3.3.3 Particle Filter

The simulation result is shown in Figure 18. The overall settings for two filters are the same with experiment described in section 3.1.3. After several iterations, robot can localize itself. Since the environment contains more obstacles than that of empty room, the localization usually ends earlier than that of empty room.

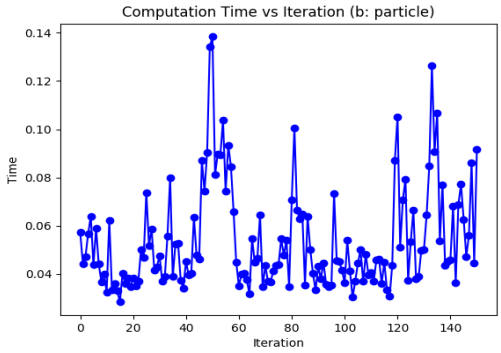
## 3.4 Conclusion

In this project, we explore two types of filters: Kalman filter and particle filter. We introduce the general ideas about these two filters. Implementation of algorithms are also reviewed. Finally, to test their functionality, we conduct experiments in both simple 2d playground and also in OpenRave environment. Considering to explore the advantages and drawbacks of the filters, we setup asymmetric, empty, and symmetric environment in OpenRave. We Compare Kalman filter (and its extension) to particle filter in known initial position cases, but also illustrate the case of a kidnapped robot using only the particle filter.

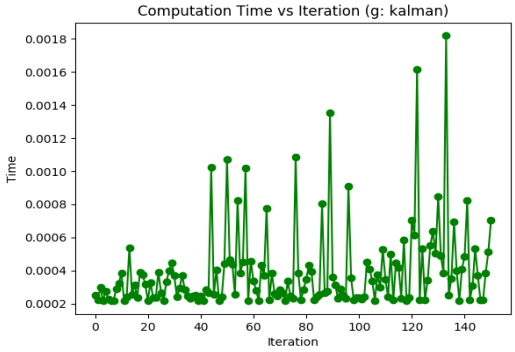
Kalman filter needs good modelling on both system dynamics and sensors. Specifically, regular Kalman filter needs linear matrix representation of the system along with the assumption of uncorrelated, zero mean Gaussian noise. Extend Kalman filter can deal with nonlinear model by local linearization on some nominal points, but still with Gaussin-Markov assumption. Also, ex-



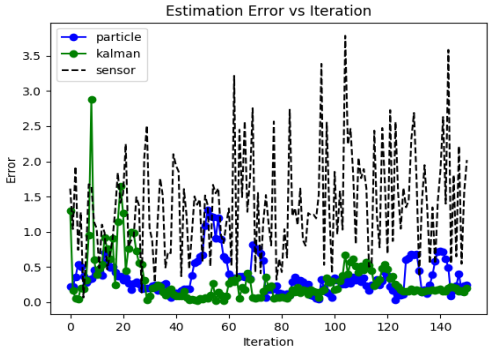
(a) Simulation Result.




(b) Particle Filter Time




(c) Kalman Filter Time




(d) Estimation Error of Both Filters

Figure 17: Extended Kalman Filter v.s. Particle Filter in Symmetric Environment. a) Red points are ground truth, purple points are extended Kalman filter results, light blue are particle filter result. Red line shows the connection to point (0,0) and noisy observed location (showed in dark blue points), and robot receives this erroneous distance measurements and begins its extended Kalman filtering process. Yellow point (beneath the robot now) shows the particle filter estimate.

tend Kalman filter is sensitive to noise when the nonlinear model changes rapidly near the nominal point. Generally, even if the system is nonlinear, we can conduct analysis and stimulate with some combination of Gaussian distributions and use the estimated mean and covariance to feed in the Kalman gain calculation. The computation of Kalman filter is efficient since it only involve direct calculation from matrices. Kalman filter gives estimation with bounded stable error. If the scenario doesn't require high accuracy, then Kalman filter is a good candidate to make estimates. Particle filters outperform Kalman filter in nonlinear system with requirement of high accuracy. But it needs more computational resources, because it need to maintain a great amount of particles and evaluate the changing distribution over time. The weight function (which transforms the possible observation to weight of that particle) should be carefully designed. Clusters diversity and abandoning invalid particle should also be paid attention. If one can trade off the things well, particle filter can give you estimates with high accuracy. One should select the filters based on specific requirements and assumptions of a scenario.

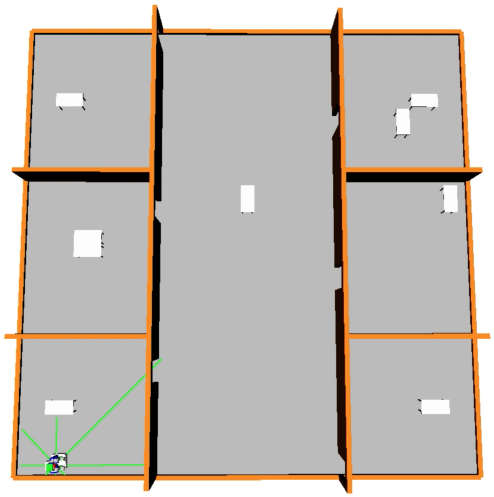


Figure 18: Symmetric Environment Kidnapped Robot Particle Filter Example.