# Home Work 05 – Predicting Robot Trajectory

Group Members: Geesara Prathap, Taimoor Shakeel

## 1 Problem Statement

Robot trajectory needs to predicted with the help of sensor fusion. To do this there are few assumptions to be made. Lego robot motion model is constant velocity based model. So this model can model as a first order system. To predict the trajectory velocity and position of the robot are considered.

## 2 Problem Formulation

### 2.1 Design State Transition Function and Measurment Function

System has position and constant velocity, so the state variable needs both of these. The matrix formulation could be

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

To be designed state transition form $t-1$ to $t$. The Newtonian equations for a subsequent time steps of robot can be written as bellow:

$$\begin{aligned} x_t &= x_{t-1} + v\Delta t \\ v_t &= v_{t-1} \end{aligned}$$

Same thing in matrix notation,

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \mathbf{F} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Where

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

The measurement function needs to implement,

$$\mathbf{z} = \mathbf{Hx}$$

. For data acquisition we used the **AndroSensor** android application which is freely available in the google app store. This application is capable of measuring accelerometer readings(incl. linear acceleration and gravity sensors), gyroscope readings, ambient magnetic field values, proximity sensor readings and few other measurements. In this assignment, there were two mobile phone were placed on robot and acquire separate sensor readings in parallel. Those two sensor reading can be found in the $sensor_reading$ directory. Here in order to measure the position of robot, linear acceleration which is calculated using accelerometer and gravity sensors of the mobile phone is used. Since sensor measures the linear acceleration of x, y and z direction, initially acceleration on each direction need to be convert into position by tow-wise integration. Once position is derived from the acceleration, it can be incorporate with the this model. Initially work with one accelerometer and then use both sensors reading and do the sensor fusion to measure the robot position in x and y direction separately. Since sensor does not measure the velocity, for one accelerometer $H$ can formed in the way.

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

But if two sensors are incorporated to measure the position, $H$ should be reformed as bellow,

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}$$

### 2.2 Design Process Noise and Measurement Noise

Since sensors are not perfect, some errors can be occurred during its measurement. Same goes for processing the model as well. Because of some internal and external effects on configuration space. In here[1], discrete time Wiener process is used to define the noise of the process. Thus, throughout the this assignment, $Q$ or processing noise is deducted using discrete time Wiener process.

$$\mathbf{Q} = \begin{bmatrix} \sigma_x^2 & \sigma_y \sigma_x \\ \sigma_x \sigma_y & \sigma_y^2 \end{bmatrix}$$

To measure the noise of the sensors ($R$) there is no standard methods. What I have done is, assume such that there is no correlation between two sensors and variance of each sensor is measured by using collected sensor data. If the model is incorporate with two sensors, $R$ can be defined in the way,

$$\mathbf{R} = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

### 2.3 Initial Conditions

Set the initial position at (0) with a velocity of (0). Covariance matrix $P$ sets to a large value like 100.

Let's consider when two sensors are being used,

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{P} = \begin{bmatrix} 100 & 0 \\ 0 & 100 \end{bmatrix}, \mathbf{Q} = \begin{bmatrix} \Delta t^3/3 & \Delta t^2/2 \\ \Delta t^2/2 & \Delta t \end{bmatrix}, \mathbf{H} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix}, \mathbf{R} = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_y^2 \end{bmatrix}$$

where $\Delta t = 0.1s$ and $\sigma_x^2$ which is variance of sensor 1 and $\sigma_y^2$ is the variance of sensor 2. Also velocity is set to $20m/s$ of Lego robot.

### 2.4 Sensor Data Prepossessing

Initially need to extract linear acceleration sensor reading form the corresponding sensor data, and integrate two-wise and derive position of robot. As an example,

```
position_vector1_x, position_vector1_y, position_vector1_z \
    = get_positon_vectors('./sensor_reading/Sensor_record_20
    171109_185848_AndroSensor1.csv', time_interval)
```

## 3 Robot Trajectory Estimation

In order to estimate robot trajectory, first apply filter in x direction and then y direction and translate robot coordinate system into coordinate system which is start at 0,0 position. Because as shown in Fig 1, robot can have negative coordinate because if robot move below its starting position it contains negative position relative to its starting position. So here simple translation model is used which is shown in Fig 2. Implementation of this can be found in *plot_robot_trajectory*().

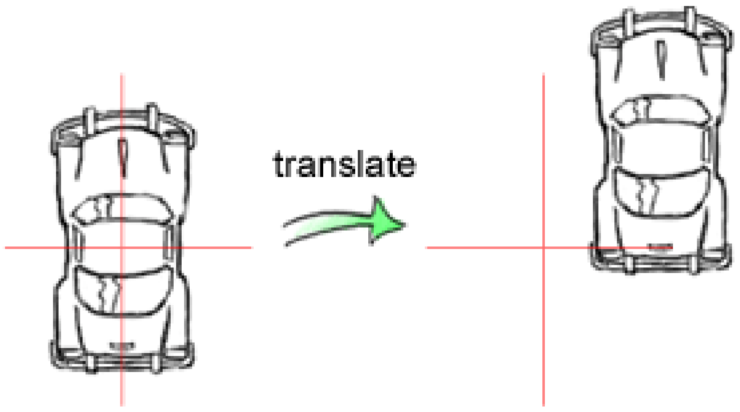

Figure 1: Root coordinate system transformation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 2: Root coordinate system transformation

## 4 Apply Kalman Filter

Filter initial configuration when using only one measurement,

```
    kf = KalmanFilter(dim_x=2, dim_z=1)
    kf.F = array([[1., dt], [0., 1.]])
    kf.H = array([[1., 0.]])
    kf.x = array([[0.], [0.]])
    kf.Q *= array([[(dt ** 3) / 3, (dt ** 2) / 2],
                   [(dt ** 2) / 2, dt]]) * 0.02
    kf.P *= 100
    kf.R[0, 0] = ps_sensor1_sigma
```

Filter initial configuration when using both measurement,

```
    kf = KalmanFilter(dim_x=2, dim_z=2)
    kf.F = array([[1., dt], [0., 1.]])
    kf.H = array([[1., 0.], [1., 0.]])
    kf.x = array([[0.], [0.]])
    kf.Q *= array([[(dt ** 3) / 3, (dt ** 2) / 2],
                   [(dt ** 2) / 2, dt]]) * 0.02
    kf.P *= 100
    kf.R[0, 0] = ps_sensor1_2sigma
    kf.R[1, 1] = ps_sensor2_2sigma
```

## 4.1 For sensor 01

Final value of K and P,

```
Final value of P: [[ 0.44255043  0.12392473]
 [ 0.12392473  0.07142246]] x direction
Final value of P: [[ 2.06896438  0.34783357]
 [ 0.34783357  0.1189648 ]] y direction
Final value of K: [[ 0.03307631]
 [ 0.00556078]] x direction
Final value of K: [[ 0.03307631]
 [ 0.00556078]] y direction
```

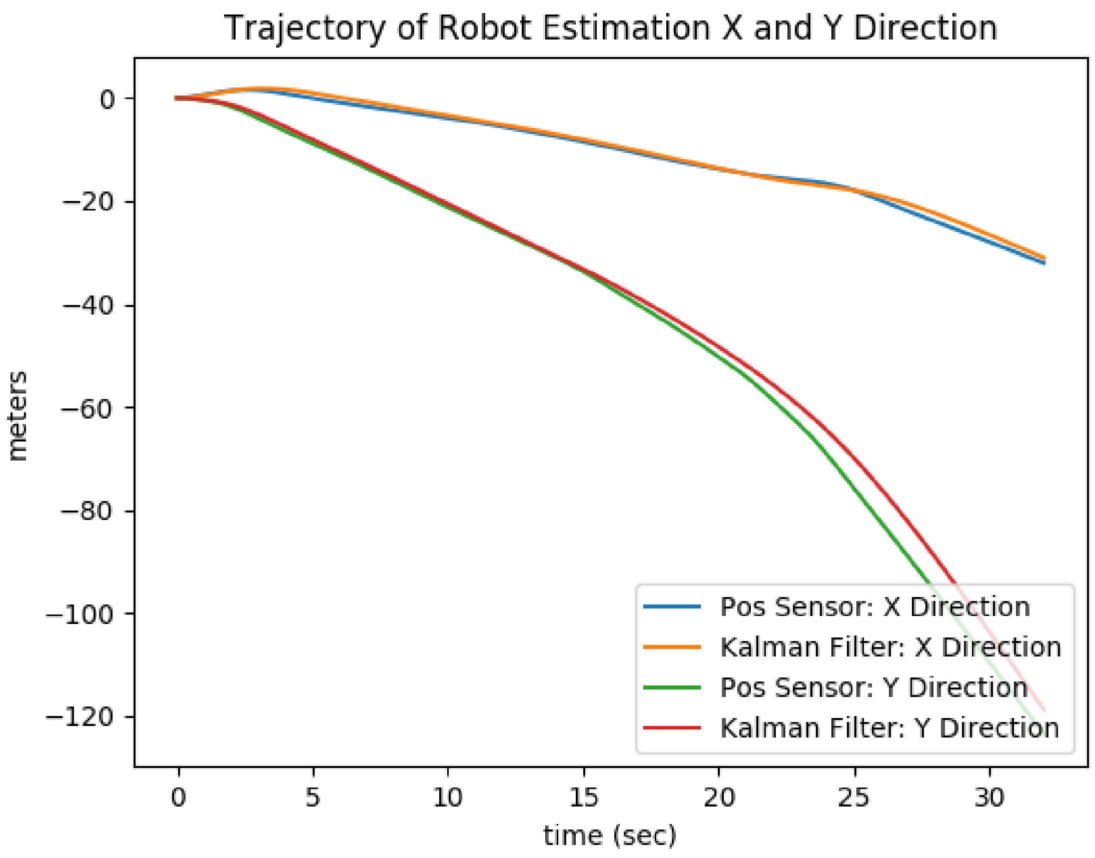

Figure 3: Robot trajectory estimation

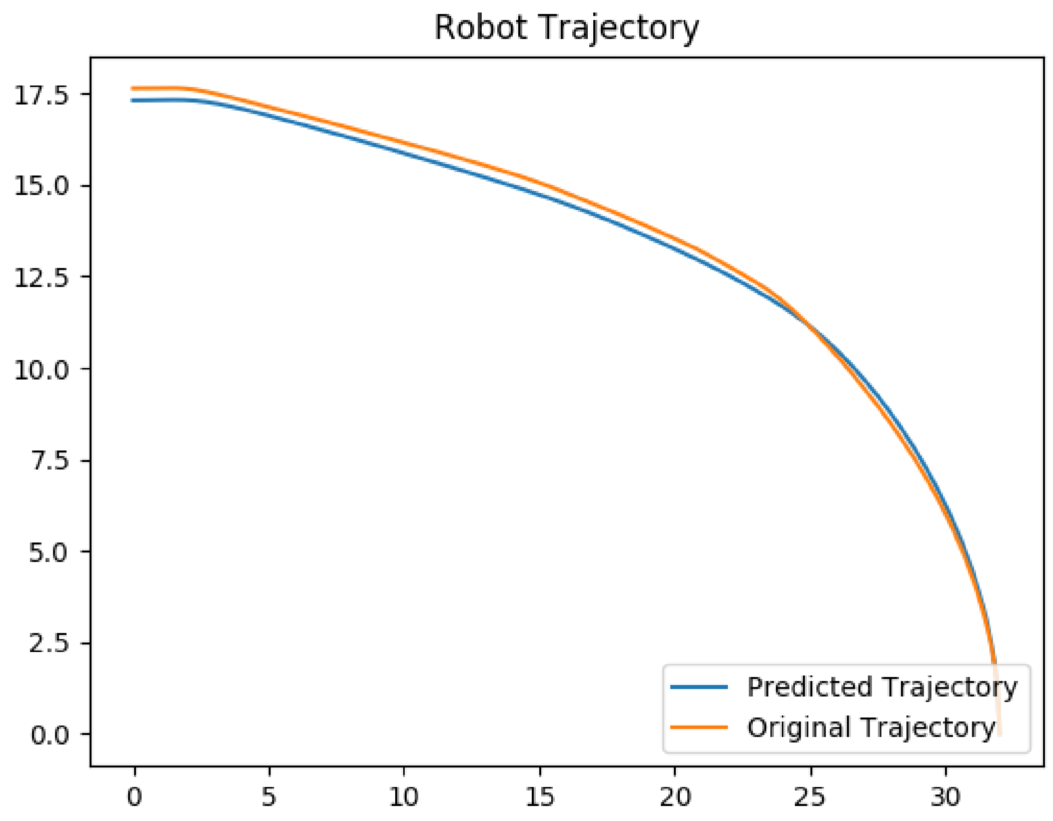

Figure 4: Comparison between robot trajectory projection with and without Kalman filter

### 4.2 For sensor 02

Final value of K and P,

```
Final value of P: [[ 0.83999421  0.19033049]
 [ 0.19033049  0.0882673 ]] x direction
Final value of P: [[ 2.24966799  0.36787084]
 [ 0.36787084  0.12231507]] y direction
Final value of K: [[ 0.0321876 ]
 [ 0.00526339]] x direction
Final value of K: [[ 0.0321876 ]
 [ 0.00526339]] y direction
```

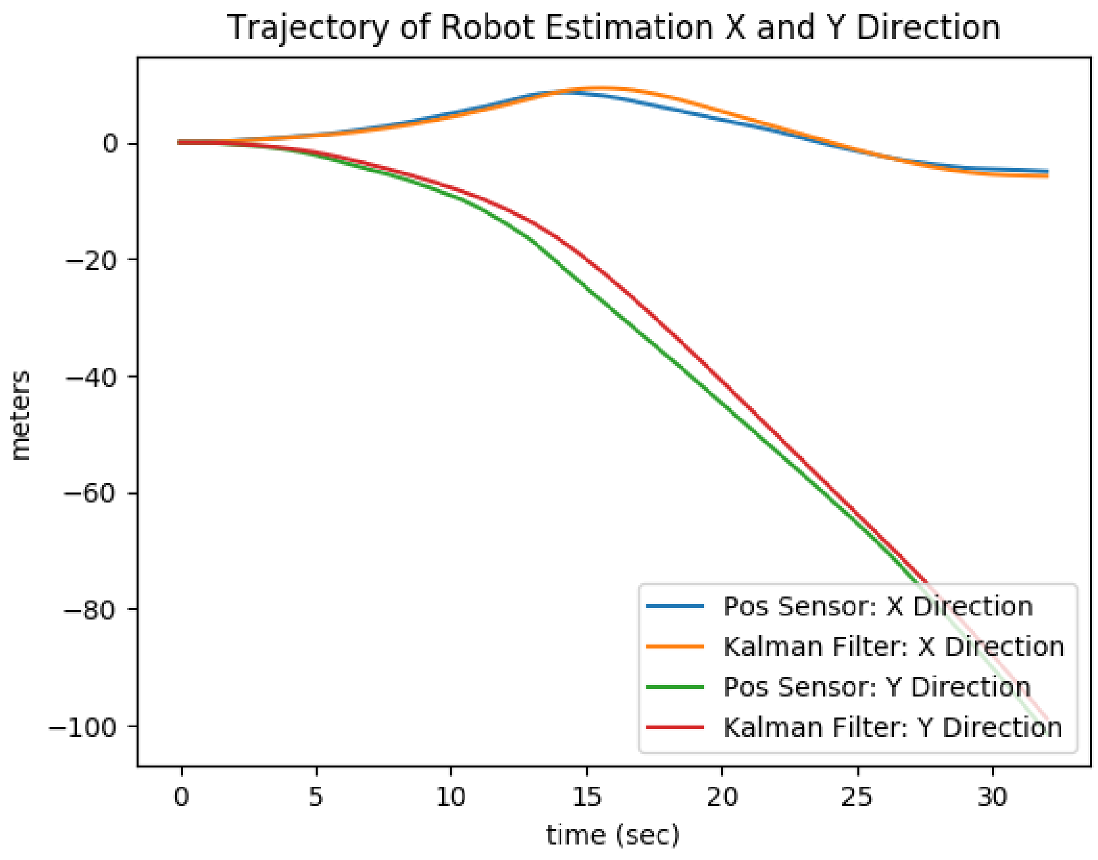

Figure 5: Robot trajectory estimation

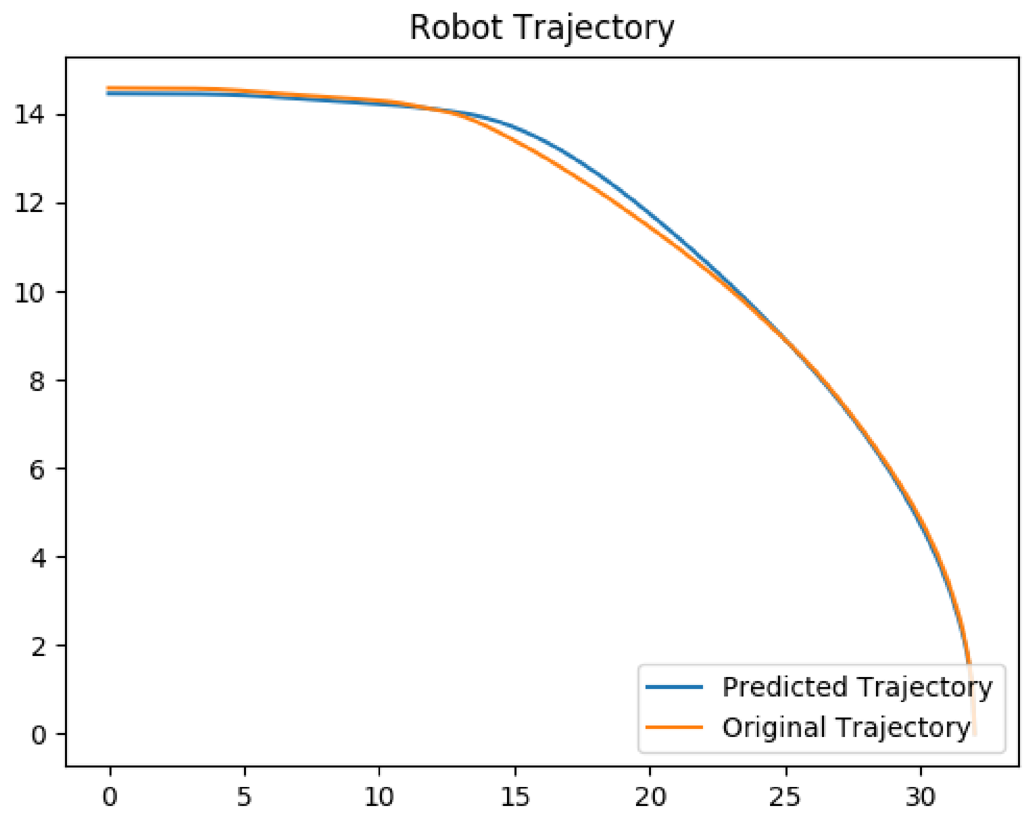

Figure 6: Comparison between robot trajectory projection with and without Kalman filter

### 4.3 For sensor 01 and 02 together

Final value of K and P,

```
Final value of P: [[ 0.33782789  0.10341486]
 [ 0.10341486  0.0653345 ]] x direction
Final value of P: [[ 1.27718213  0.25192063]
 [ 0.25192063  0.10139572]] y direction
Final value of K: [[ 0.02041817  0.01827355]
 [ 0.00402743  0.00360441]] x direction
Final value of K: [[ 0.02041817  0.01827355]
 [ 0.00402743  0.00360441]] y direction
```

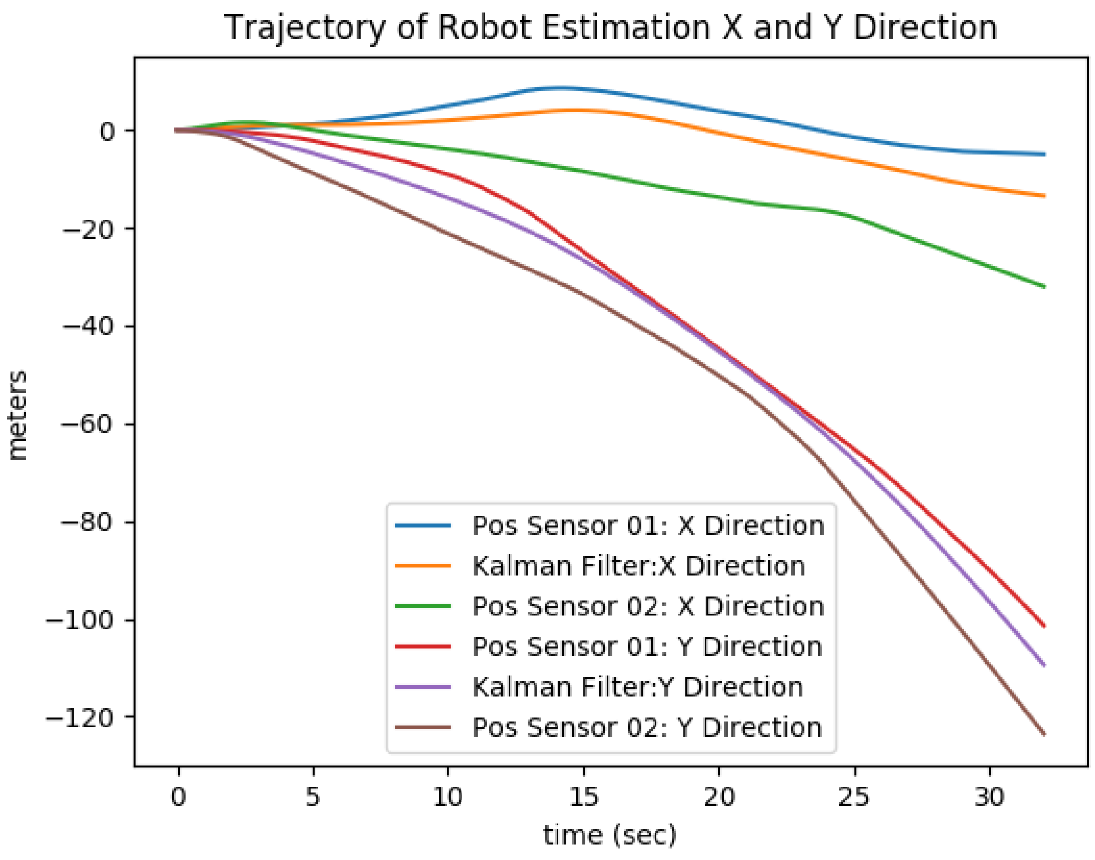

Figure 7: Robot trajectory estimation

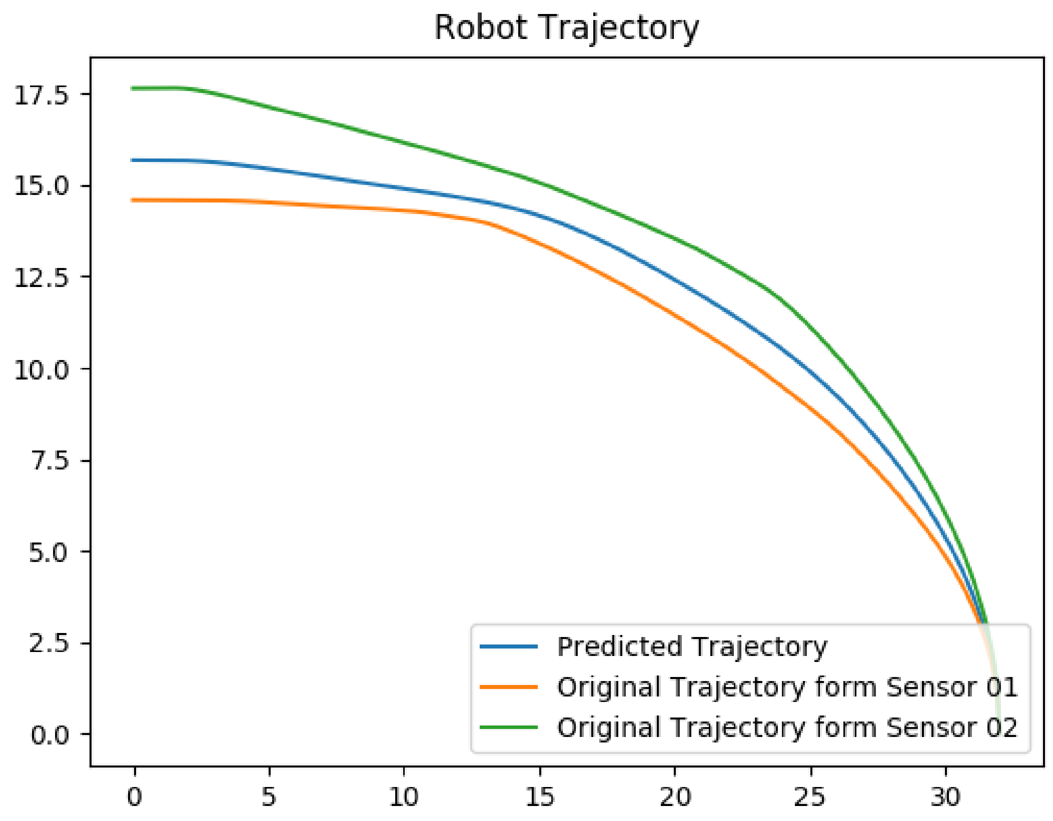

Figure 8: Comparison between robot trajectory projection with and without Kalman filter

## 5 Apply Extended Kalman Filter

Filter initial configuration when using only one measurement,

```
rk = ExtendedKalmanFilter(dim_x=2, dim_z=1)
rk.x = np.array([0., 0.0])
rk.F = np.array([[1., dt], [0., 1.]])
rk.R = np.diag([ps_sensor1_2sigma])
rk.Q = np.array([[(dt ** 3) / 3, (dt ** 2) / 2],
                 [(dt ** 2) / 2, dt]]) * 0.02
rk.P *= 100
```

Filter initial configuration when using both measurement,

```
rk = ExtendedKalmanFilter(dim_x=2, dim_z=2)
rk.x = np.array([0., 40.0])
rk.F = np.array([[1., dt], [0., 1.]])
rk.R = np.diag([ps_sensor1_2sigma, ps_sensor2_2sigma])
rk.Q = np.array([[(dt ** 3) / 3, (dt ** 2) / 2],
                 [(dt ** 2) / 2, dt]]) * 0.02
rk.P *= 100
```

Since position estimation function is linear, there won't be necessary to linearize in each iteration. So each time it should give same results when it tries to linearize. Implementation details can be found by observing $HJacobian\_at(x)$ and $hx(x)$ functions.

### 5.0.1 For sensor 01

Final value of K and P,

```
Final value of P: [[ 0.4679619   0.13106625]
 [ 0.13106625  0.07240845]] x direction
Final value of P: [[ 2.139574    0.35972965]
 [ 0.35972965  0.1199564 ]] y direction
Final value of K: [[ 0.05448272]
 [ 0.01525946]] x direction
Final value of K: [[ 0.03307401]
 [ 0.00556078]] y direction
```

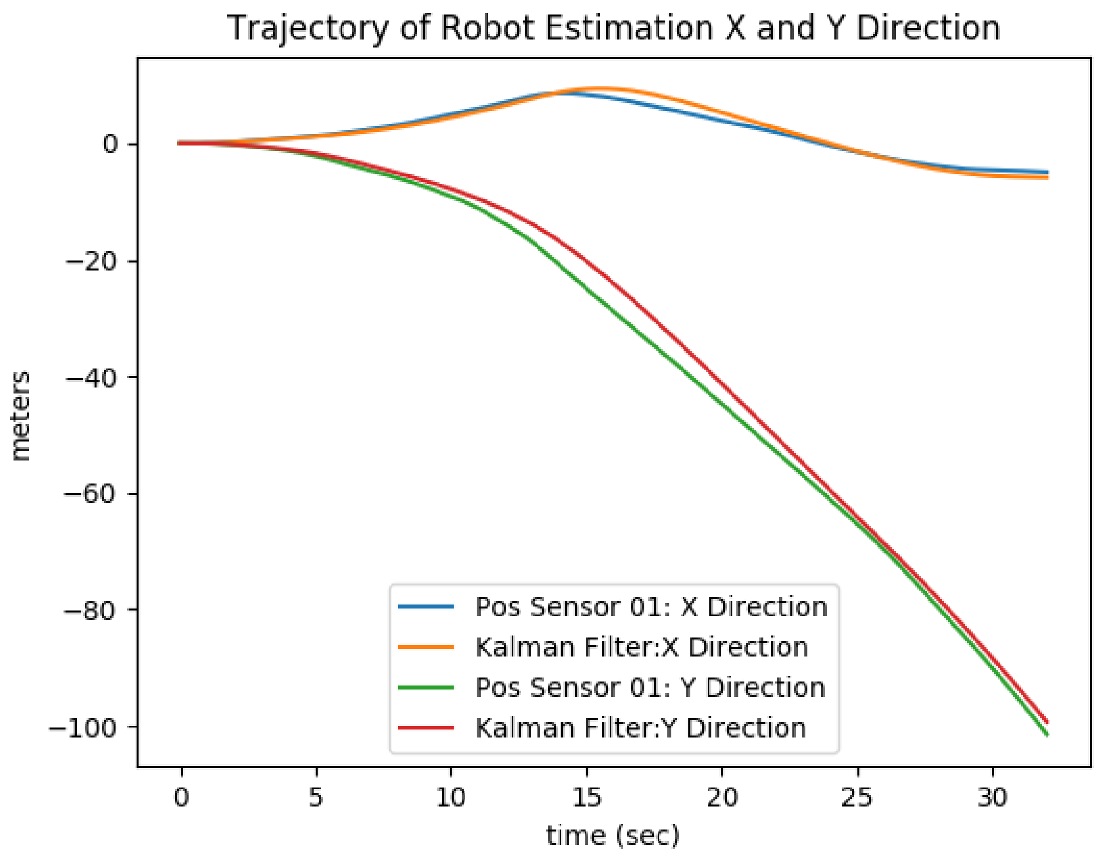

Figure 9: Robot trajectory estimation

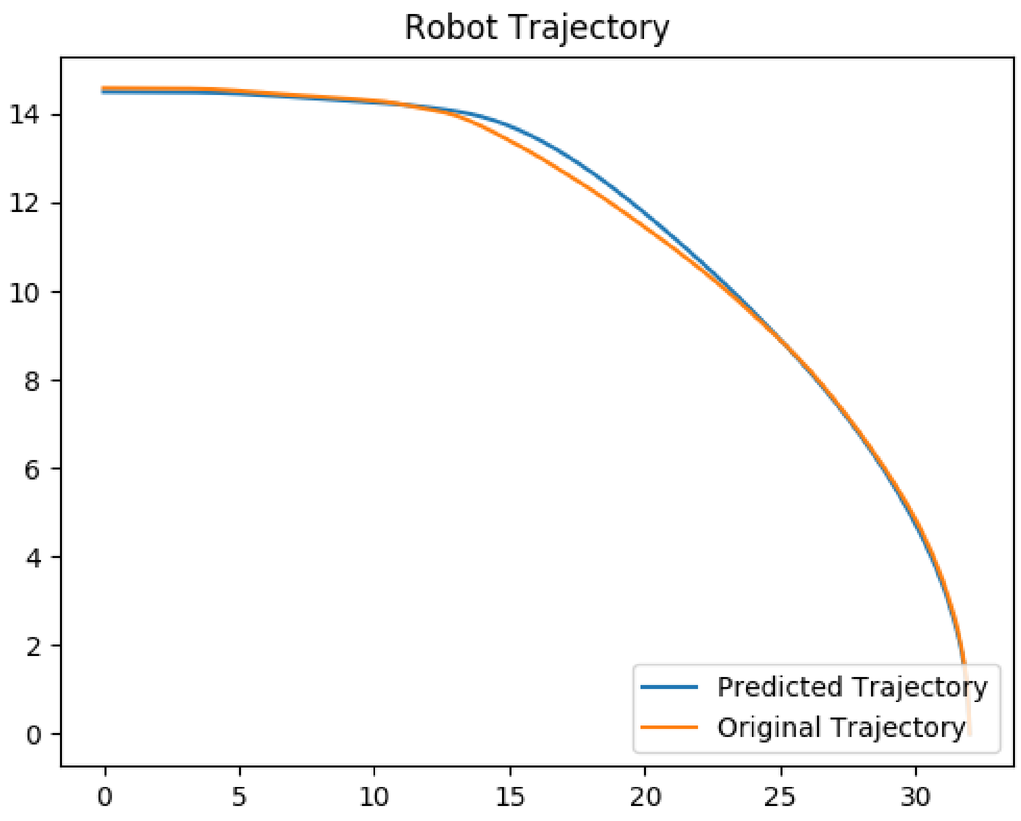

Figure 10: Comparison between robot trajectory projection with and without Extended Kalman filter

#### 5.0.2 For sensor 02

Final value of K and P,

```
Final value of P: [[ 0.87883425  0.19915665]
 [ 0.19915665  0.08925597]] x direction
Final value of P: [[ 2.32431464  0.38010199]
 [ 0.38010199  0.12330689]] y direction
Final value of K: [[ 0.04431489]
 [ 0.0100424 ]] x direction
Final value of K: [[ 0.03218549]
 [ 0.0052634 ]] y direction
```

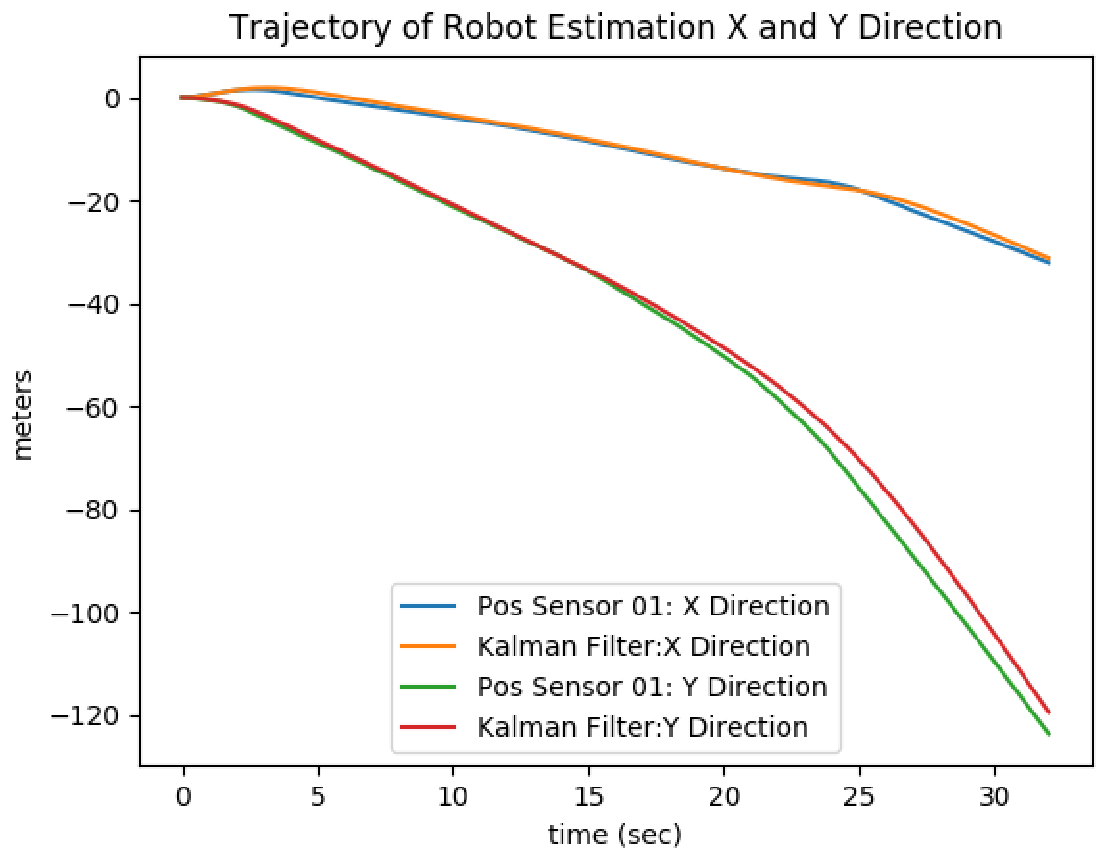

Figure 11: Robot trajectory estimation

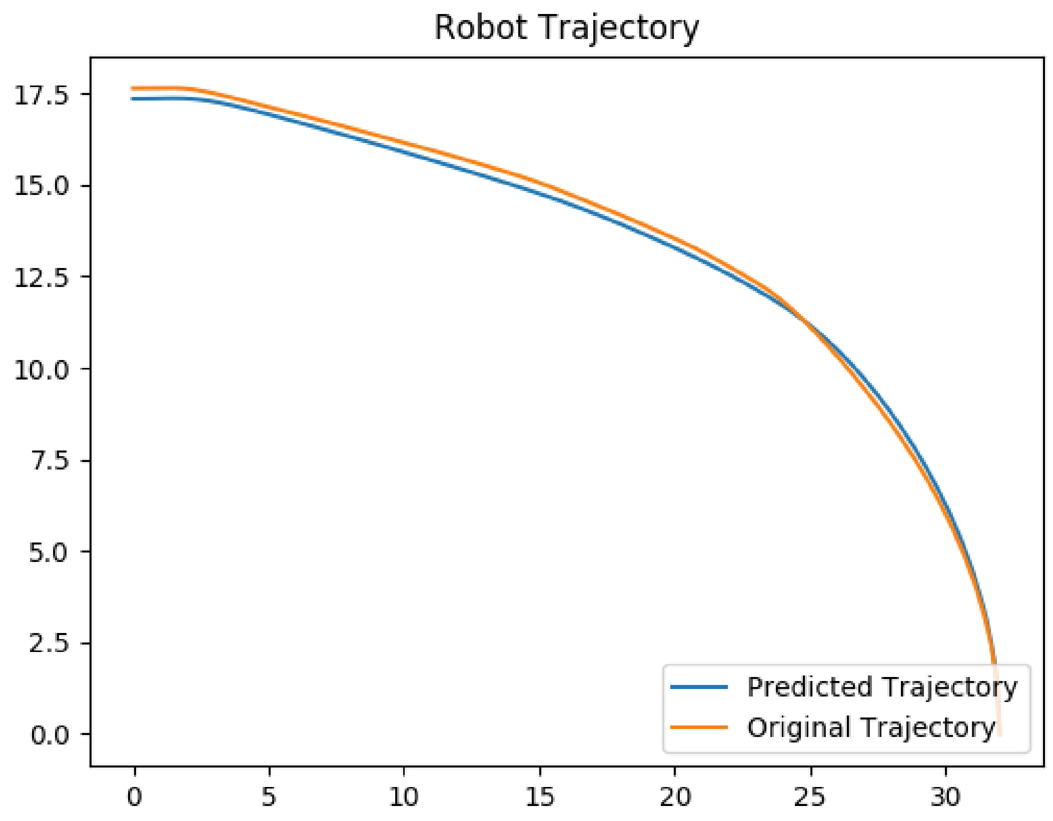

Figure 12: Comparison between robot trajectory projection with and without Extended Kalman filter

#### 5.0.3 For sensor 01 and 02 together

Final value of K and P,

```
Final value of P: [[ 0.35908411  0.10994752]
 [ 0.10994752  0.06631918]] x direction
Final value of P: [[ 1.32845507  0.26205969]
 [ 0.26205969  0.10238586]] y direction
Final value of K: [[ 0.04158872  0.0178207 ]
 [ 0.012734    0.0054565 ]] x direction
Final value of K: [[ 0.02041622  0.01827181]
 [ 0.00402744  0.00360441]] y direction
```

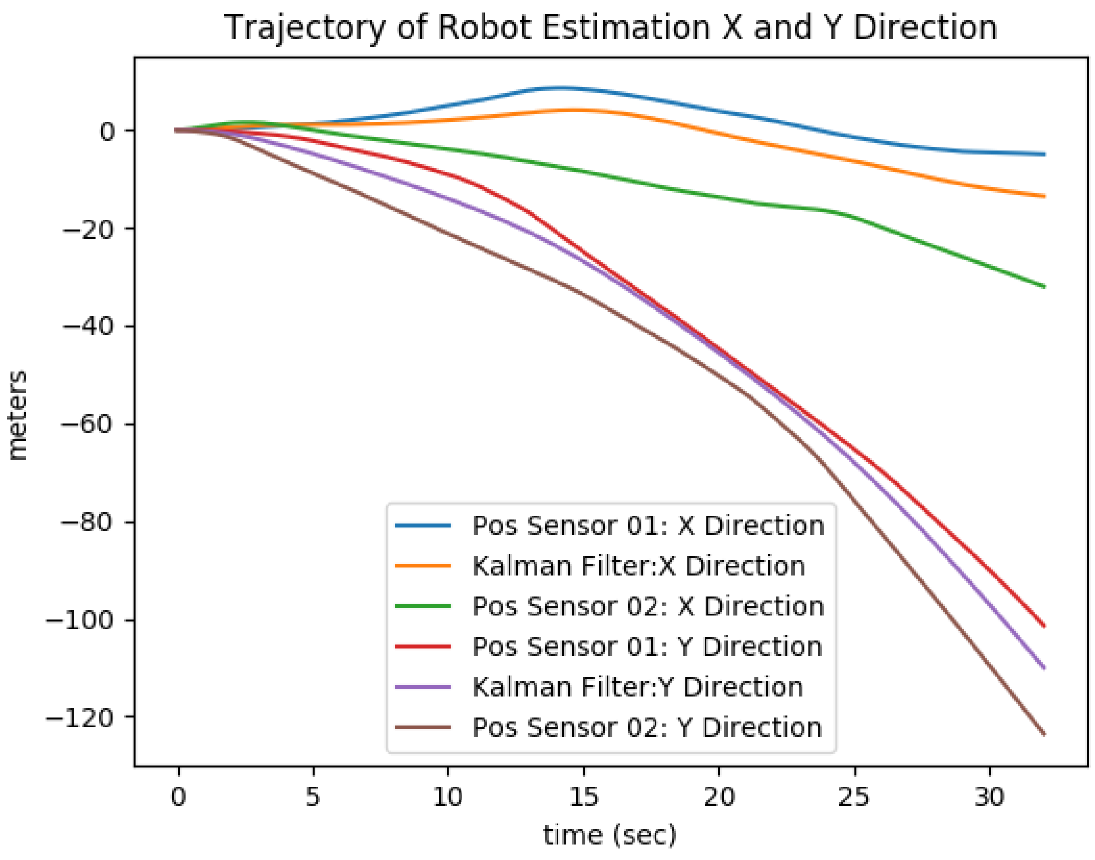

Figure 13: Robot trajectory estimation

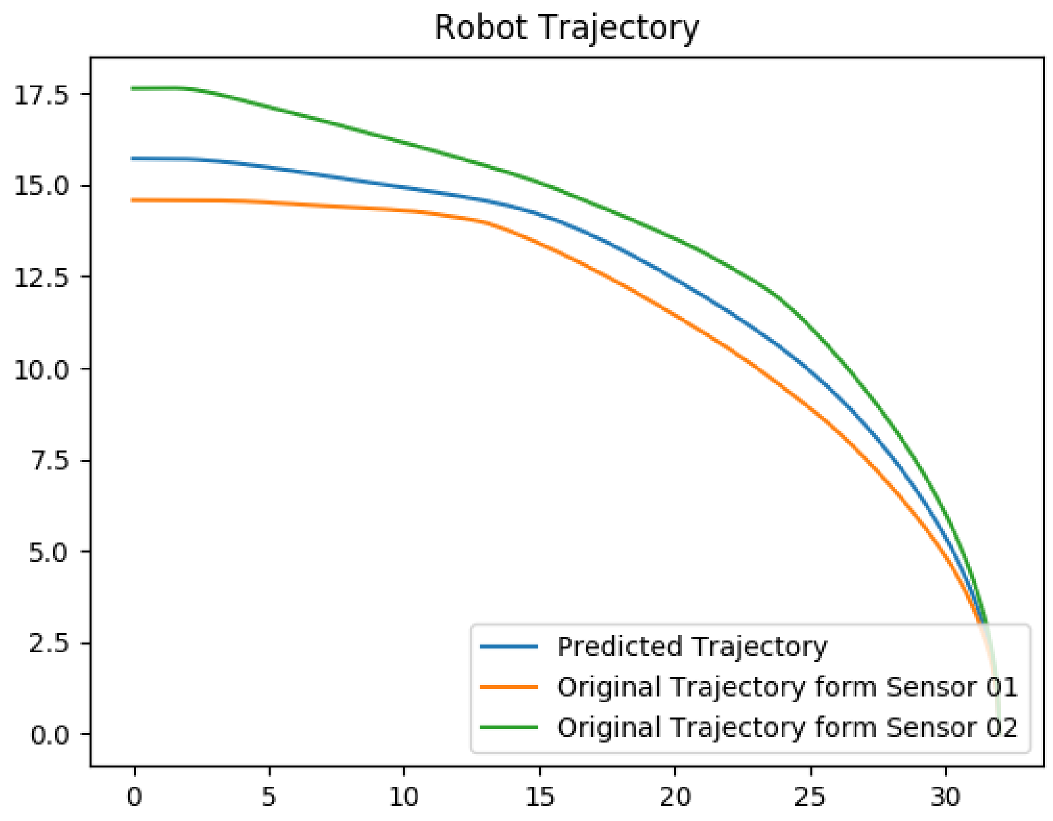

Figure 14: Comparison between robot trajectory projection with and without Extended Kalman filter

## 6 Apply Unscented Kalman Filter

Filter initial configuration when using only one measurement,

```
    sigmas = MerweScaledSigmaPoints(2, alpha=.1, beta=2., kappa=1.)
    ukf = UnscentedKalmanFilter(dim_x=2, dim_z=1, fx=f_cv, hx=h_cv, dt=
    ukf.x = np.array([0., 0.0])
    ukf.R = np.diag([ps_sensor1_2sigma])
    ukf.Q∗= array([[(dt ∗∗ 3) / 3, (dt ∗∗ 2) / 2],
                   [(dt ∗∗ 2) / 2, dt]]) ∗ 0.02
```

Filter initial configuration when using both measurement,

```
    sigmas = MerweScaledSigmaPoints(2, alpha=.1, beta=2., kappa=1.)
    ukf = UnscentedKalmanFilter(dim_x=2, dim_z=2, fx=f_cv, hx=h_cv, dt=
    ukf.x = np.array([0., 0.0])
    ukf.R = np.diag([ps_sensor1_2sigma, ps_sensor2_2sigma])
    ukf.Q∗= array([[(dt ∗∗ 3) / 3, (dt ∗∗ 2) / 2],
                   [(dt ∗∗ 2) / 2, dt]]) ∗ 0.02
```

Generates sigma points and weights according to Van der Merwe's 2004 dissertation method is used (MerweScaledSigmaPoints). In order to generate sigma points it is required to defined function which is used to map in order to generate sigma points which also need to be represented motion model of the robot. Here $f_cv$ is the function which is used for state transition function for a constant velocity robot.

### 6.0.1 For sensor 01

Final value of K and P,

```
Final value of P: [[ 0.44255708  0.12392472]
 [ 0.12392472  0.07142246]] x direction
Final value of P: [[ 2.06874621  0.34779318]
 [ 0.34779318  0.11895702]] y direction
Final value of K: [[ 0.0544931 ]
 [ 0.01525937]] x direction
Final value of K: [[ 0.03307271]
 [ 0.00556013]] y direction
```

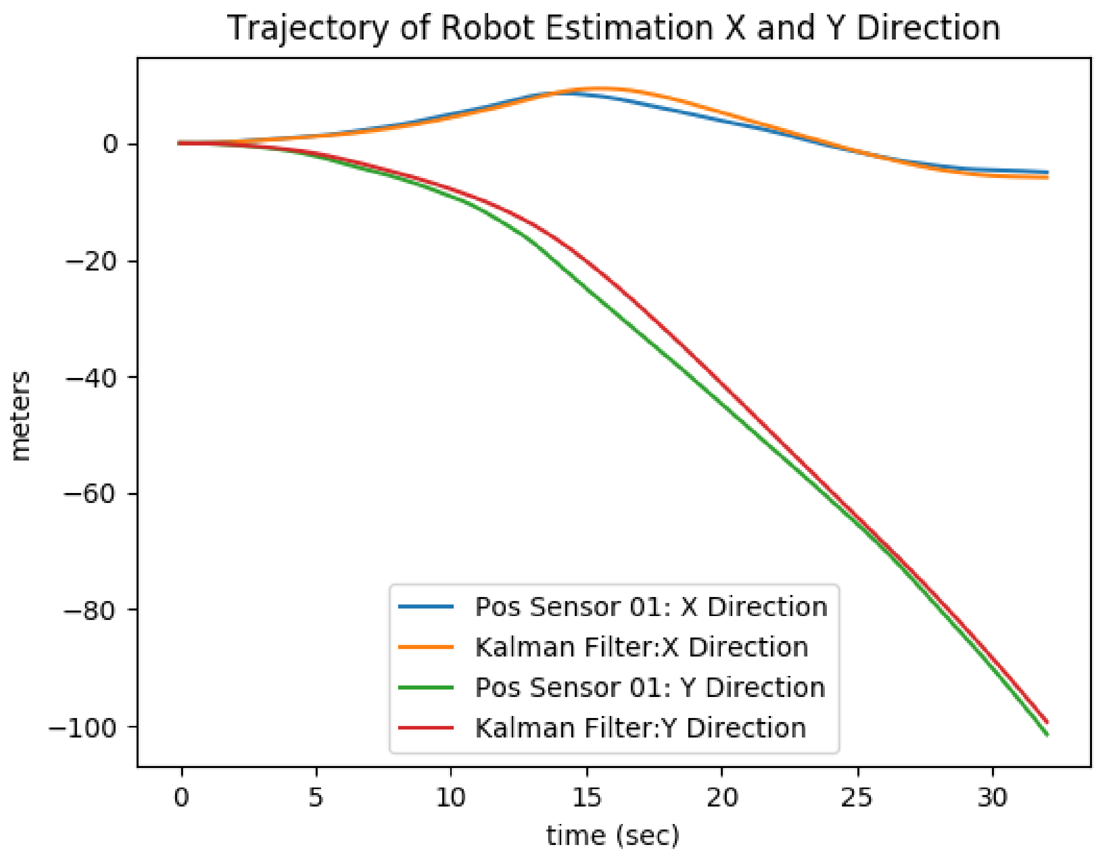

Figure 15: Robot trajectory estimation

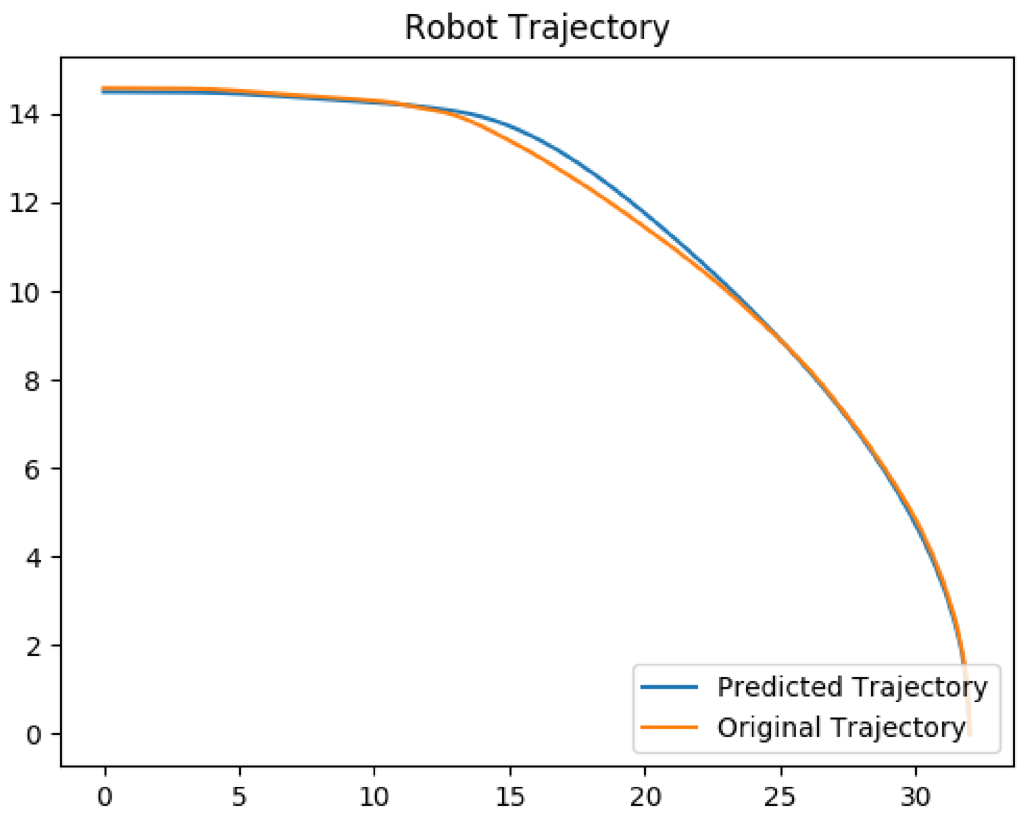

Figure 16: Comparison between robot trajectory projection with and without Unscented Kalman filter

### 6.0.2 For sensor 02

Final value of K and P,

```
Final value of P: [[ 0.84000067  0.19033037]
 [ 0.19033037  0.08826718]] x direction
Final value of P: [[ 2.24935201  0.36780698]
 [ 0.36780698  0.12230164]] y direction
Final value of K: [[ 0.04432044]
 [ 0.01004236]] x direction
Final value of K: [[ 0.03218299]
 [ 0.00526248]] y direction
```

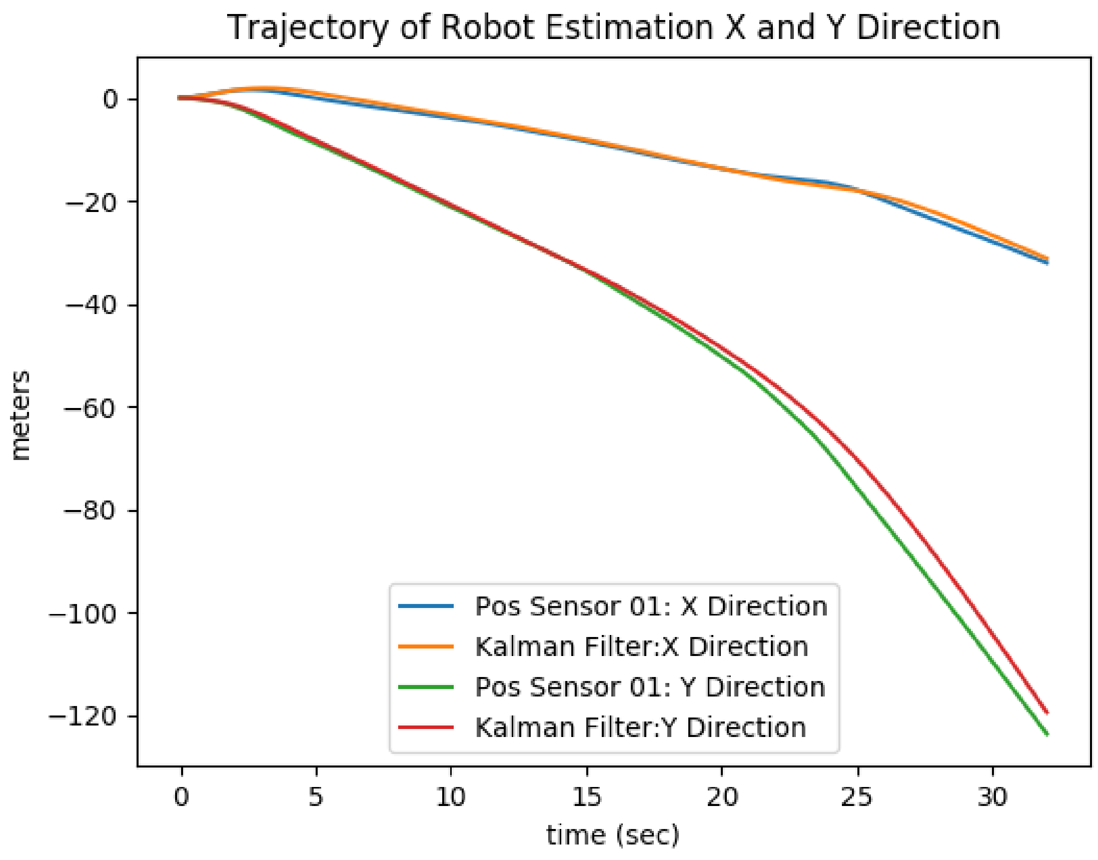

Figure 17: Robot trajectory estimation

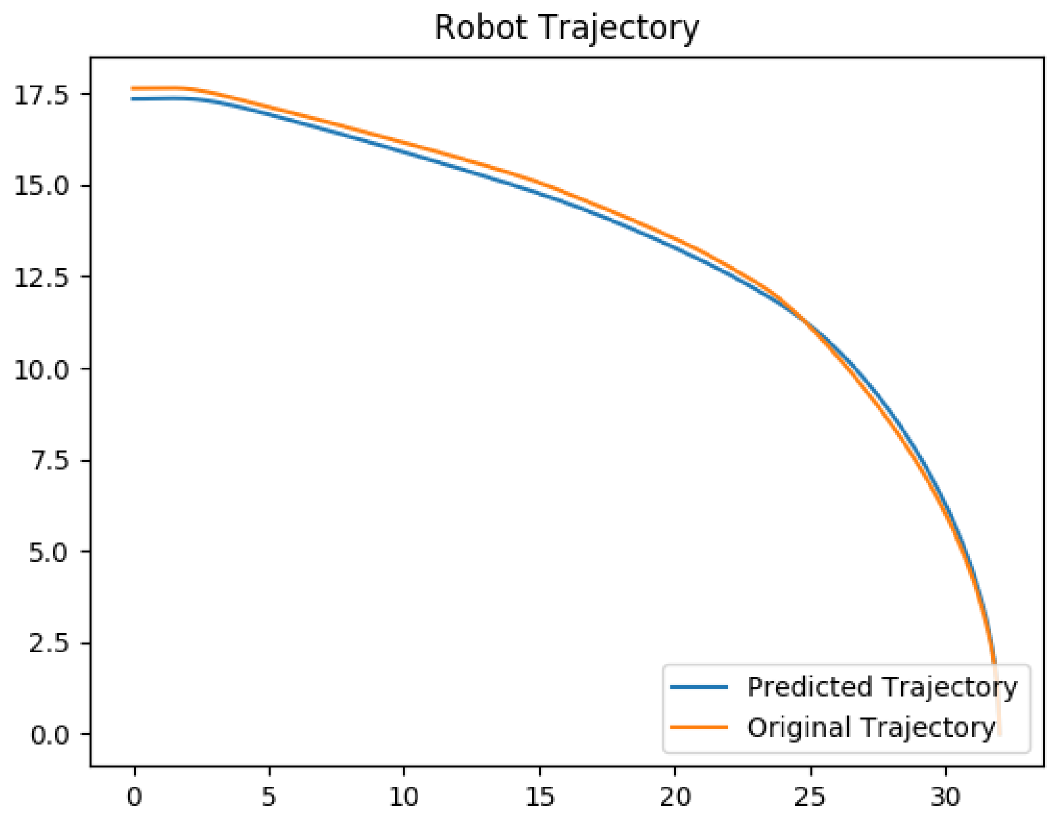

Figure 18: Comparison between robot trajectory projection with and without Unscented Kalman filter

### 6.0.3 For sensor 01 and 02 together

Final value of K and P,

```
Final value of P: [[ 0.33783456  0.10341486]
 [ 0.10341486  0.0653345 ]] x direction
Final value of P: [[ 1.27718039  0.25192063]
 [ 0.25192063  0.10139559]] y direction
Final value of K: [[ 0.04159818  0.01782475]
 [ 0.01273391  0.00545646]] x direction
Final value of K: [[ 0.02041804  0.01827343]
 [ 0.00402743  0.00360441]] y direction
```

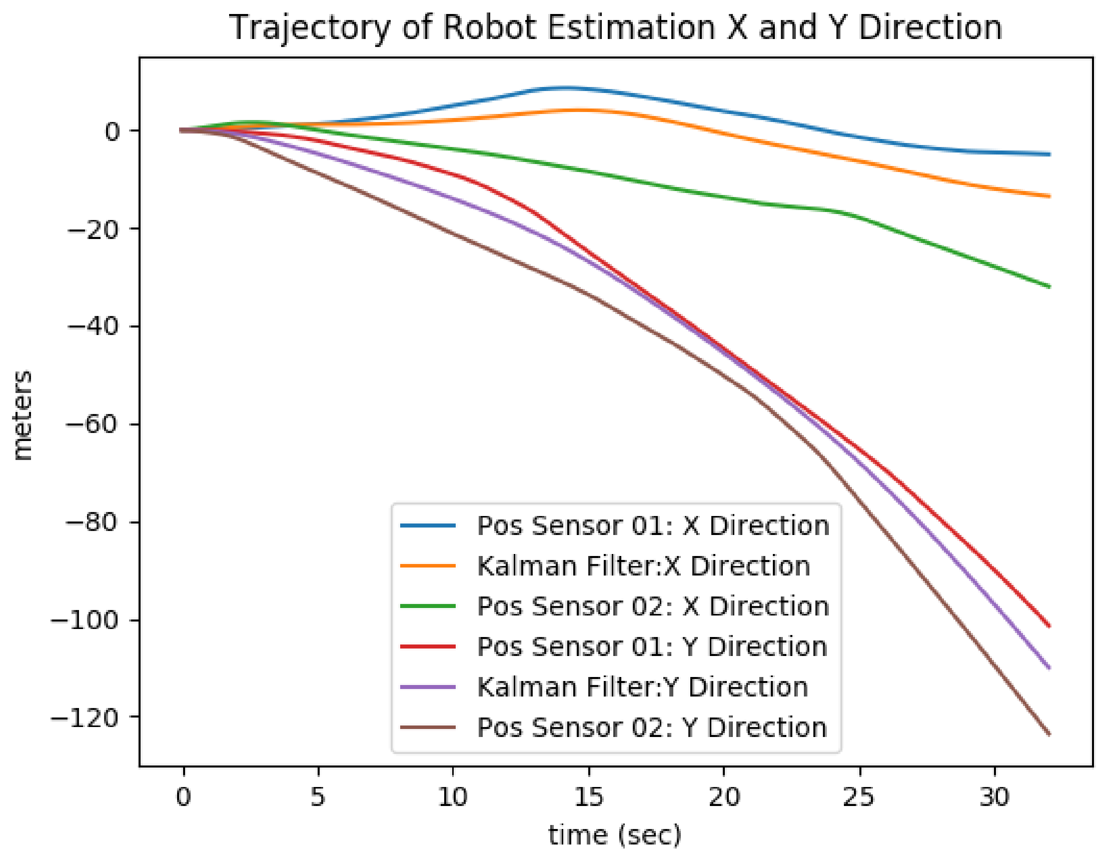

Figure 19: Robot trajectory estimation

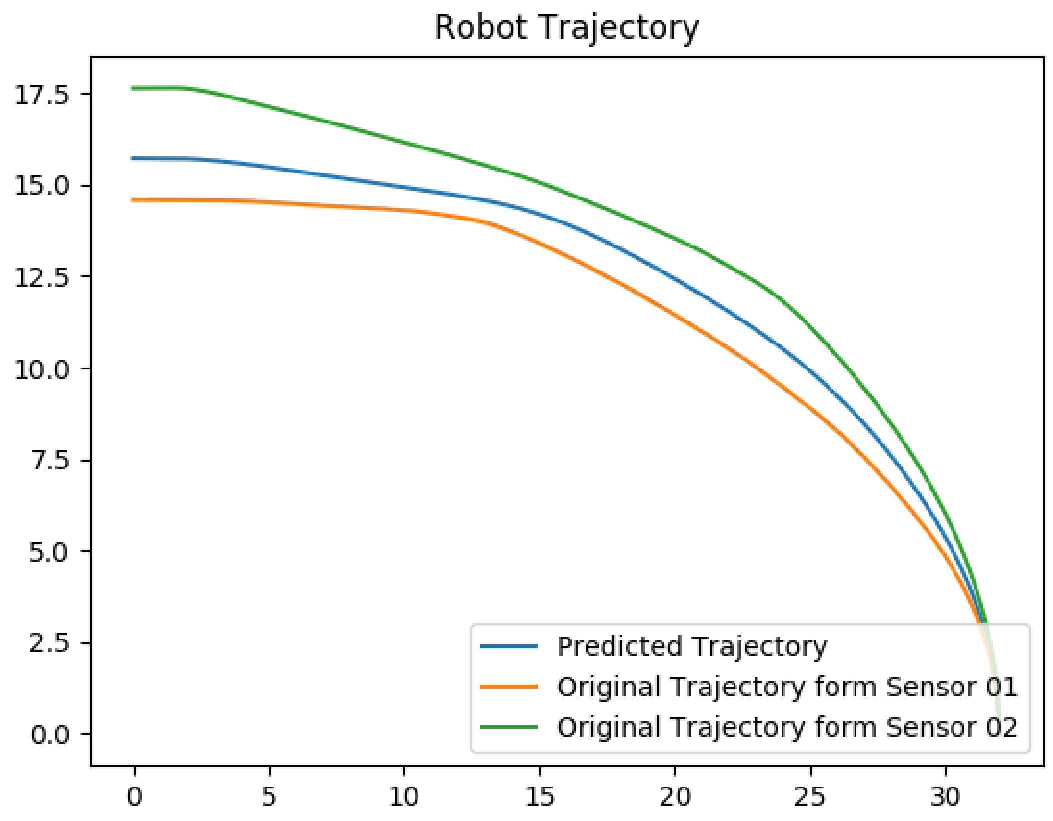

Figure 20: Comparison between robot trajectory projection with and without Unscented Kalman filter

## 7 Apply Particle Filter

Filter initial configuration,

```
    x = 0.1
    x_P = np.zeros(number_of_points)
    for i in range(0, number_of_points):
        x_P[i]= x + np.sqrt(robot_sigma) + np.random.uniform(-1, 1, 1)[
```

Here *number_of_points* is denoted as how many points are required to generate random points around current position of the robot. In this implementation it is equals to 100 points. Also assume that robot motion can be described according to the Gaussian distribution. Based on this assumption reweighing is done.

```
P_w[k] = ((1 / np.sqrt(2 * np.pi * ps_sensor1_sigma)) *
                      np.exp(-np.power(z - z_update[k], 2) / 2 * ps_sen
```

re-sampling is done using cumulative distribution.

```
        P_w = P_w / np.sum(P_w)
        cumsum = np.cumsum(P_w)
        for i in range(0, number_of_points):
            position = (np.random.rand() <= cumsum).sum()
            if (position == number_of_points):
                x_P[i] = x_P_update[position - 1]
            else:
```

```
x_P[i] = x_P_update[position]
```

But it always try to over fit the motion model and under fit the sensor measurement. So result is not correct. I think Particle filter is better for the robot localization, for sensor fusion it does not perform well out of the box. I mean proposal and sample distribution are selected correctly we may able to get good result. But for me I didn't work out.

### 7.0.1 For sensor 01

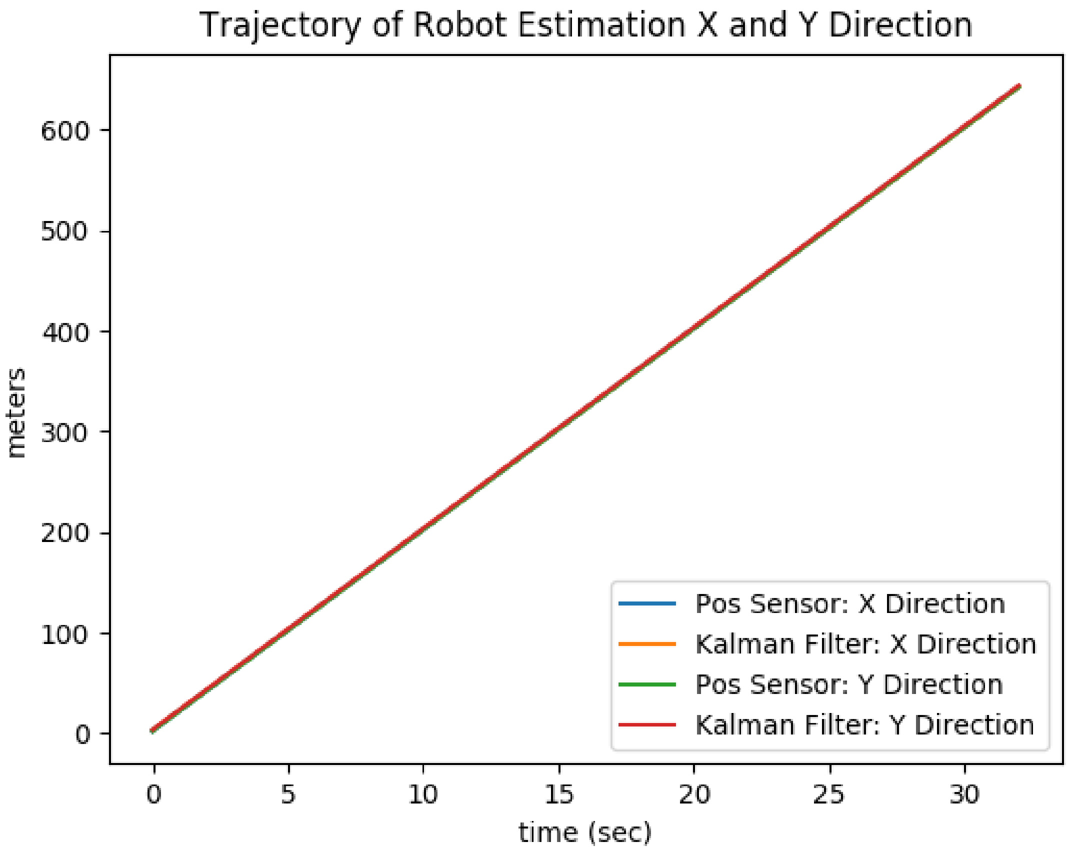

Figure 21: Robot trajectory estimation

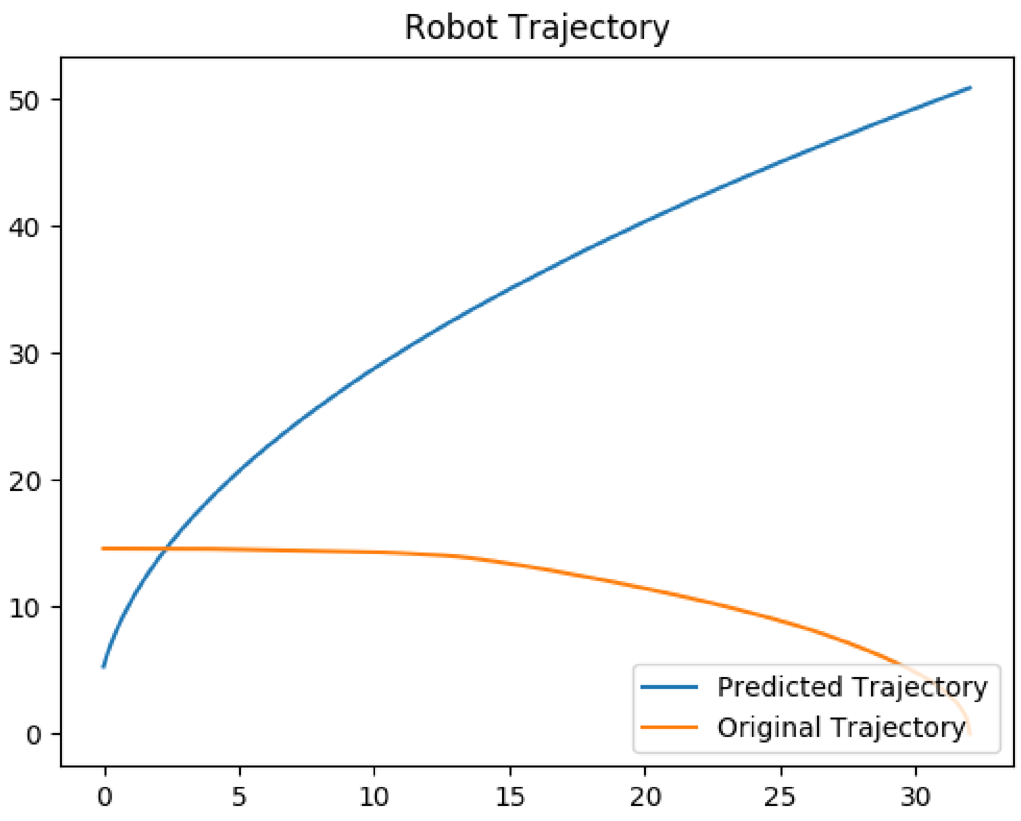

Figure 22: Comparison between robot trajectory projection with and without Particle filter

#### 7.0.2 For sensor 02

Final value of K and P,

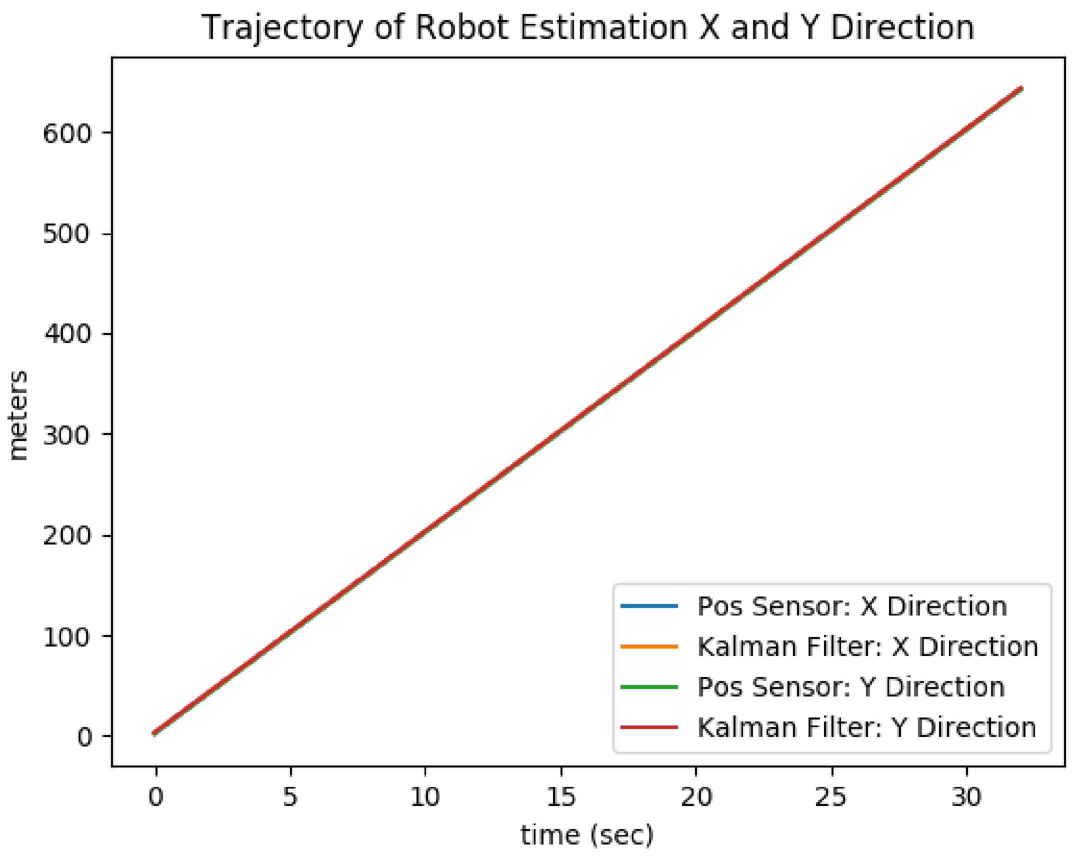

Figure 23: Robot trajectory estimation

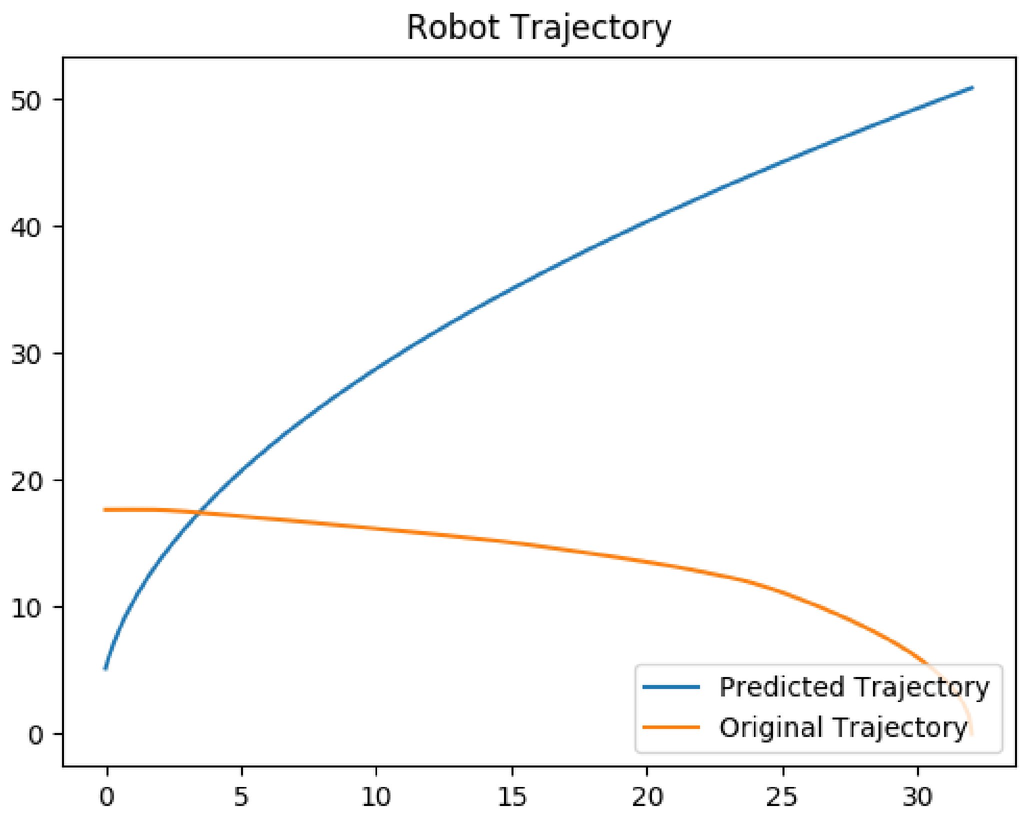

Figure 24: Comparison between robot trajectory projection with and without Particle filter

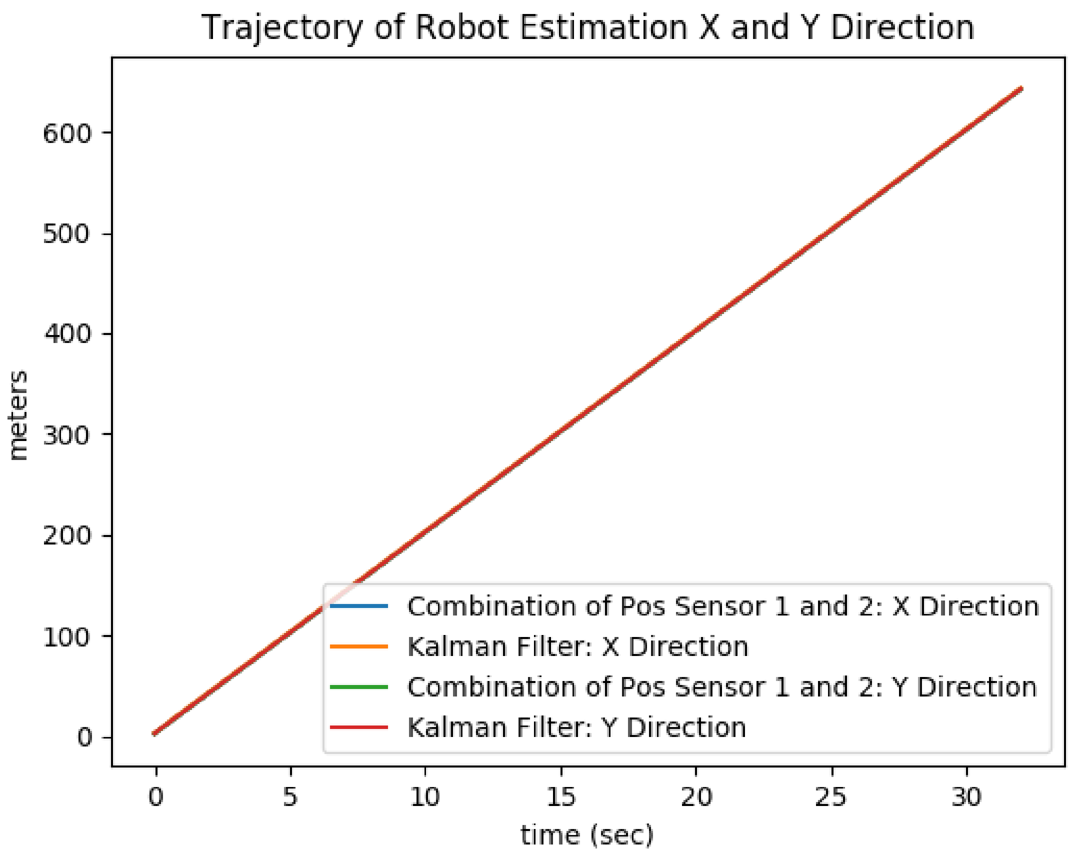

Figure 25: Robot trajectory estimation

### 7.0.3 For sensor 01 and 02 together

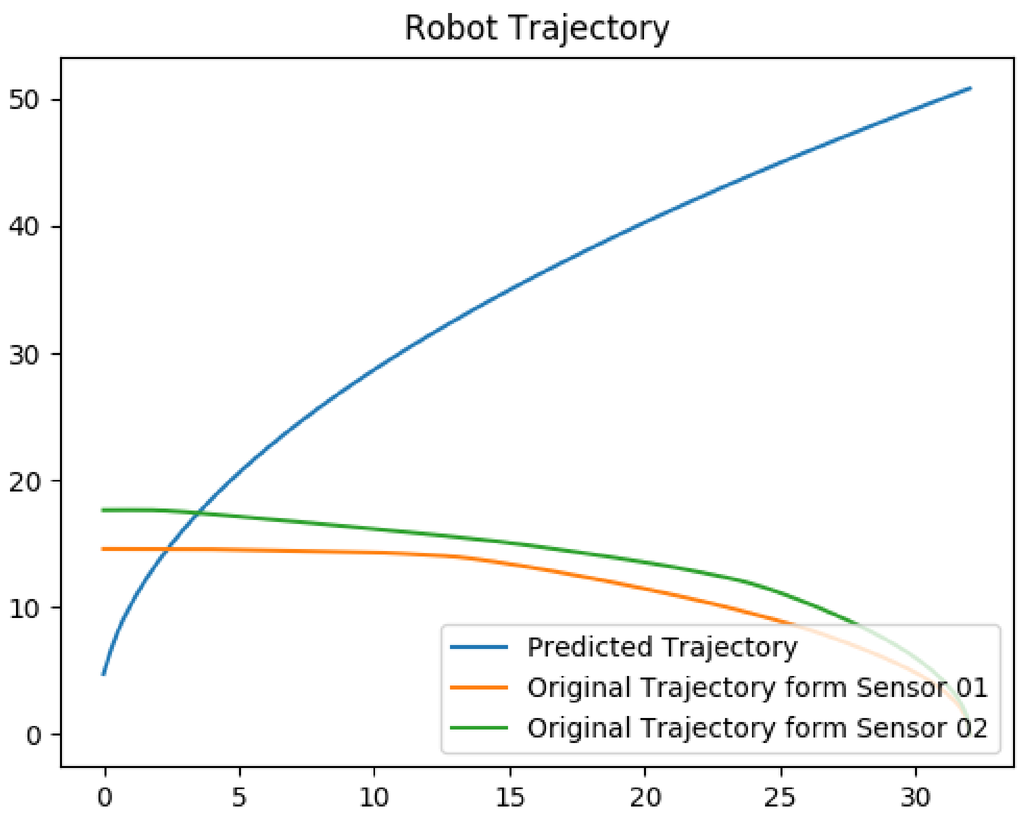

Figure 26: Comparison between robot trajectory projection with and without Particle filter

## 8 Filter Comparison

There are four filters are applied addressing same problem with same dataset. Here is the comparison among the algorithms. According to Fig 27, it can be clearly seen that result of Kalman filter, Extended Kalman filter and Unscented Kalman filter are giving same results. Because here we addressed the problem which is linear. Extended Kalman filter and Unscented Kalman filter are there in order to address non-linearity of the motion model. Here motion model is linear result is almost equals which should be the case. Since Particle filter is not performing well I removed when comparing them. For the reference, with Particle filer which is shown in Fig 28.

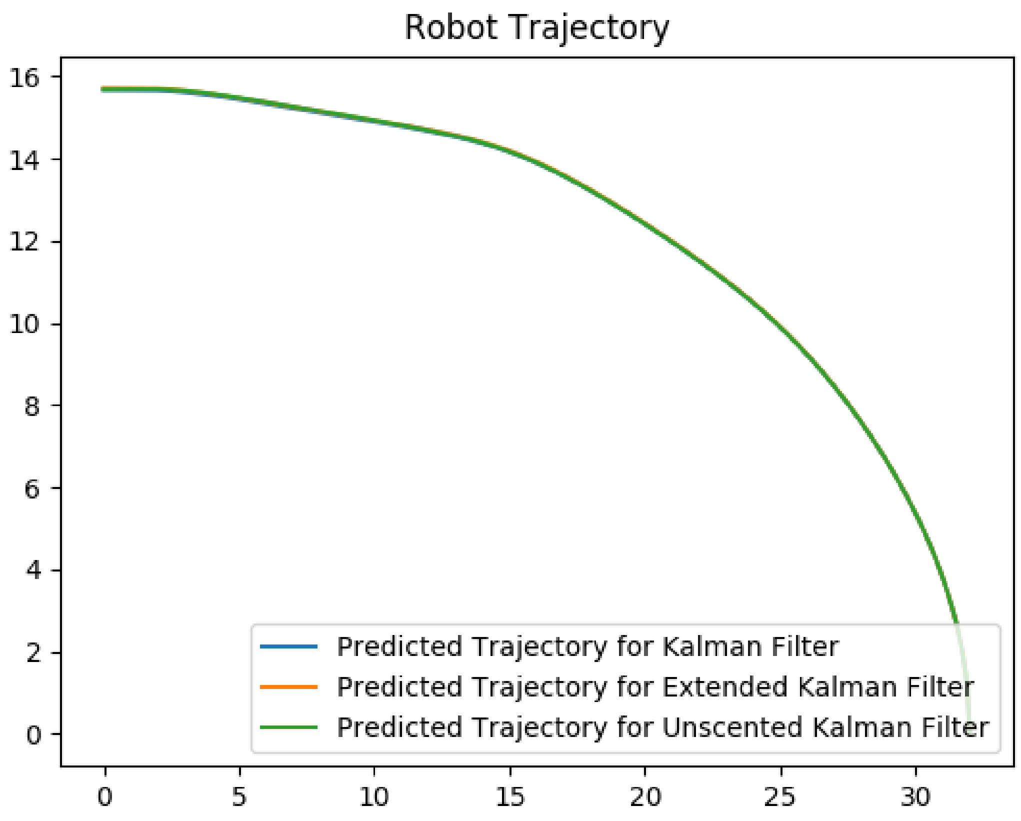

Figure 27: Comparison Result Except Particle Filter

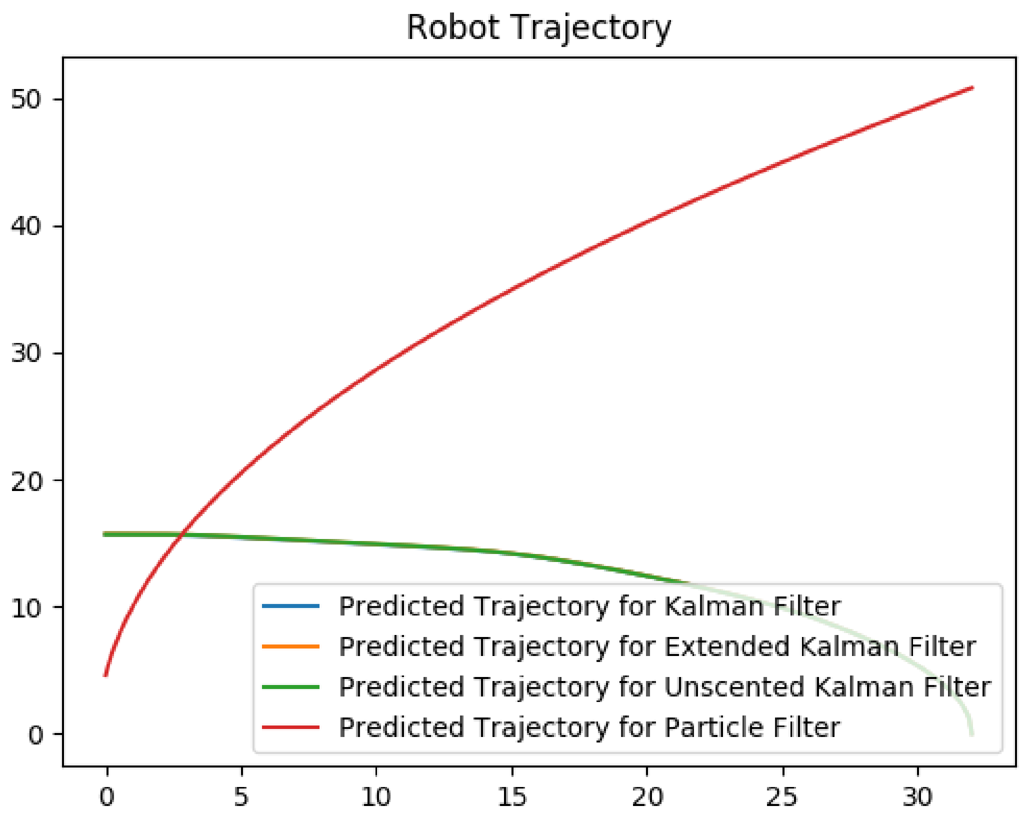

Figure 28: Comparison Result Including Particle Filter