Record For Implementing Sensor Fusion

Mingze Chen

June 24, 2021

1 Environment Setup

After connecting to the robot through *ssh*, then execute the following command:

```
husarion@husarion: $ roscore
```

start 2nd. command line window and **execute following command**:

```
$ roslaunch rosbot_ekf all.launch rosbot_pro:=true
```

start 3rd. command line window and launch the *robot_localization* through executing:

```
~/pathTo/catkin_ws$ source ./devel/setup.bash
~/pathTo/catkin_ws$ roslaunch playground start_filter.launch
```

Now our purpose is implementing *KF* algorithm into existing *move.py* script.

To launch the *UWB tag* through executing:

```
~/pathTo/catkin_ws$ source ./devel/setup.bash
~/pathTo/catkin_ws$ roslaunch localizer_dwm1001 dwm1001.launch
```

2 First Order Kalman Filter

First order Kalman Filter tracks a first order system, e.g. our state vector is: $x = \{x,\ \dot{x}\}^T$, which corresponds to X-coordinate and X-velocity.

In Kalman filter theory, the most important part is **Status update equation**:

The Kalman expression or **status update equation** is:

$$Current\ state\ estimated\ value = Predicted\ value\ of\ current\ state\ +$$
$$Kalman\ Gain\ *\ (measured\ value - predicted\ value\ of\ the\ state)$$

which is:

$$\hat{X}(t) = X_p(t) + K \times [\ X_m(t) - X_p(t)\ ] \tag{1}$$

where $K = \dfrac{\sigma_p^2}{\sigma_p^2\ +\ \sigma_m^2}$

One sample implementation [1] can be like:

```python
from collections import namedtuple
gaussian = namedtuple('Gaussian', ['mean', 'var'])
gaussian.__repr__ = lambda s: '(={:.3f}, 2={:.3f})'.format(s[0], s[1])

def update(prior, measurement):
    x, P = prior        # mean and variance of prior
    z, R = measurement  # mean and variance of measurement

    y = z - x        # residual
    K = P / (P + R)  # Kalman gain

    x = x + K*y      # posterior
    P = (1 - K) * P  # posterior variance
    return gaussian(x, P)

def predict(posterior, movement):
    x, P = posterior # mean and variance of posterior
    dx, Q = movement # mean and variance of movement
    x = x + dx
    P = P + Q
    return gaussian(x, P)
```

Inside the *playground* package there's a **Python** script called *kalman_filter.py*.

Its code snippet is here:

[1] https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/04-One-Dimensional-Kalman-Filters.ipynb

```
def predict_step(mean1, var1, mean2, var2):
    global new_mean, new_var
    new_mean = mean1 + mean2
    new_var = var1 + var2
    return new_mean, new_var

# correct step function
def correct_step(mean1, var1, mean2, var2):
    """
    This function takes in two means and two squared
        variance terms, and return updated gaussian
        parameters.
    """
    # calculate the new gaussian parameters
    new_mean = (var1 * mean2 + var2 * mean1) / (var1 + var2
        )
    # also equals to var1 * var2 /(var1  +  var2)
    new_var = 1 / (1 / var1 + 1 / var2)
    return new_mean, new_var
```

Although no variable called kalman_gain was calculated explicitly, through mathematical derivation we can know, they're the same.

Now execute:

```
$ husarion@husarion:~/pathTo/catkin_ws$ rosrun playground
    kalman_filter.py -s kf3105.csv
```

Following experiments were conducted:

1. let the robot move forward 1m, as shown in Figure 6 and 7
2. partial round movement with hard-coded velocity as shown in Figure 9 and perfect round movement with actual velocity from rostopic *velocity* as shown in Figure 10 and 11
3. Move the robot forward for about 4m with remote control alone (See 1) and the result is shown in Figure 1, 2, 3, 4 and 5

Listing 1: Kalman Filter with control input only code

```
from matplotlib.pyplot import figure

cdf = pd.read_csv('straightControl.csv')
cuwb_x = cdf.uwb_x
uwb_y = cdf.uwb_y
vel_linear_x = cdf['vel_linear_x'].fillna(0.0)
la_x = cdf.la_x
```

```python

dt = 1.
R = sensor_var
kf = KalmanFilter(dim_x=2, dim_z=1, dim_u = 1)
kf.P *= 10
kf.R *= R

# Here process_var is definitely not a proper parameter, because the remote control is not perfect,
# it can't be constant velocity of 0.5 m/s although assigned so

kf.Q = Q_discrete_white_noise(2, dt, 0.3)
kf.F = np.array([[1., 0], [0., 0.]])
kf.B = np.array([[dt], [ 1.]])
kf.H = np.array([[1., 0]])
print(kf.P)

zs = cuwb_x
xs = []
cmd_velocity = np.array([0.5])
for z in zs:
    kf.predict(u=cmd_velocity)
    kf.update(z)
    xs.append(kf.x[0])

figure(figsize=(16, 12), dpi=80)
plt.plot(xs, label='Kalman Filter')
bp.plot_measurements(zs)
plt.xlabel('Number of measurements')
plt.legend(loc=4)
plt.ylabel('X-coordinate (m)')
plt.rcParams.update({'font.size': 22})
# plt.savefig("kalman_with_control.pdf", dpi=300)
```

R and Q are derived from extensive experiments:

```python
process_var = 0.028 ** 2
sensor_var = 0.077 ** 2
```

Afterwards, position data based on calculation and *KF* are collected[2] in files called *kf3105.csv*, *kf0306.csv* and **sensorData.csv**[3].

[2]Before collecting data, always make sure that all four anchors are working properly, otherwise it's very likely that the columns *uwb_x* and *uwb_y* are both 0.0 in the entire process.

[3]sensorData.csv includes 'time', 'la_x', 'la_y', 'uwb_x', 'uwb_y', 'vel_linear_x', 'vel_angular_z', 'mpu_ang_vel_z', 'mpu_linear_acc_x', 'mpu_linear_acc_y', 'odom_linear_x', 'odom_angular_z', 'odom_filtered_linear_x', 'odom_filtered_angular_z', 'odom_yaw', 'odom_filtered_yaw'

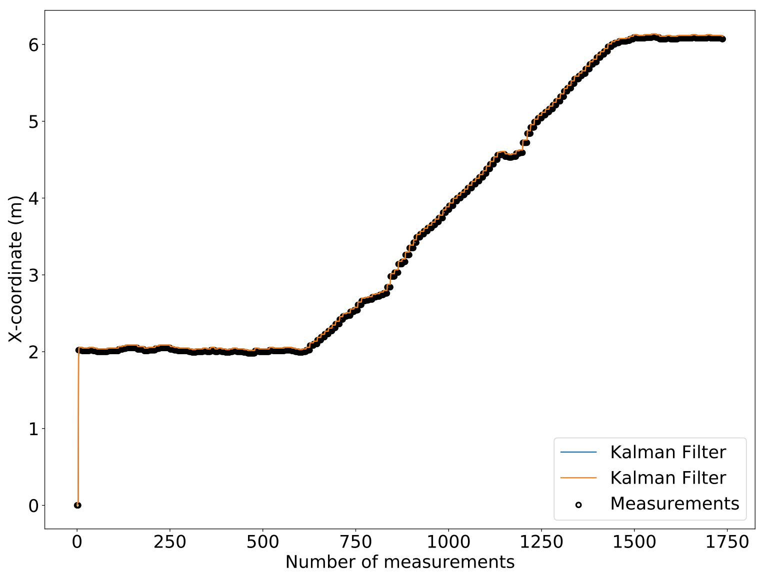

Figure 1: Single movement plot with remote control alone

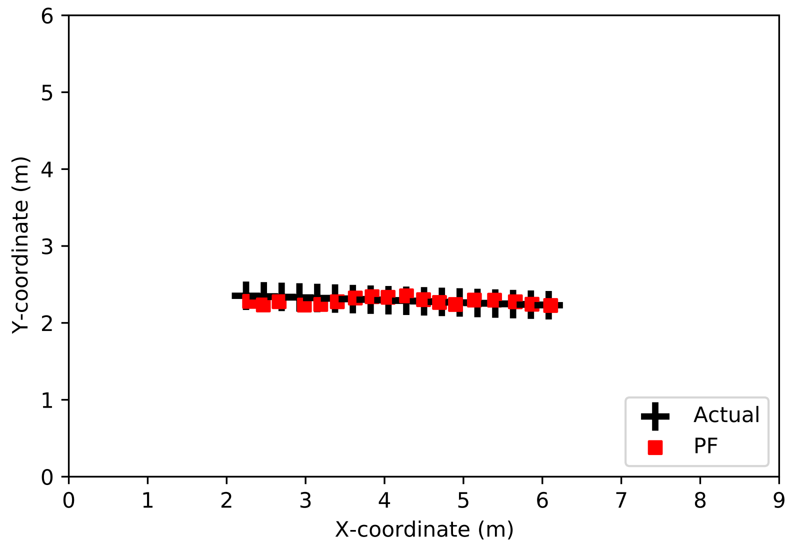

Figure 2: Single movement plot with remote control alone - 18 iteration 5k particles

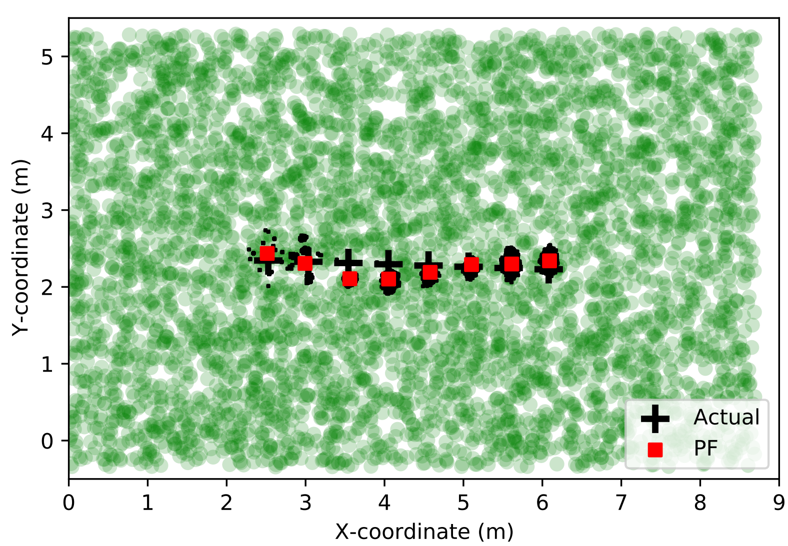

Figure 3: Single movement plot with remote control alone - 8 iteration 5k particles

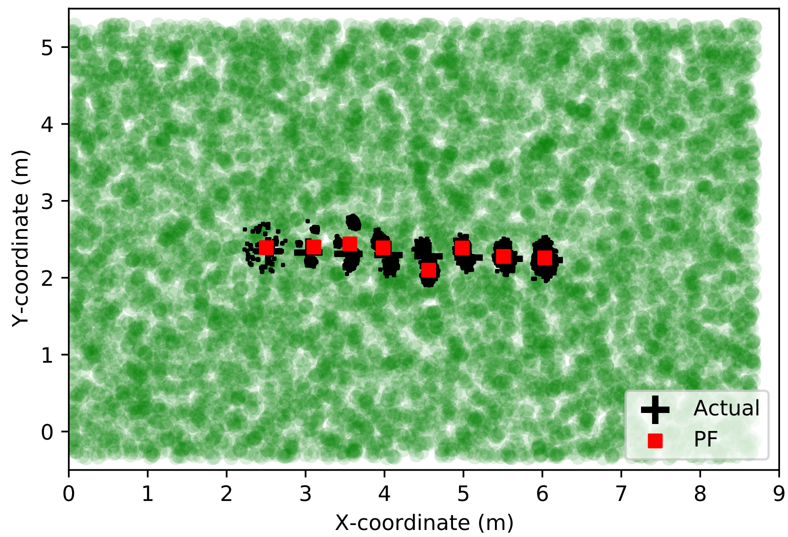

Figure 4: Single movement plot with remote control alone - 8 iteration 10k particles

3 Observation

- A bigger sensor measurement variance can make the localisation trajectory from Figure 7 smoother, which is closer to the real movement process, as shown in Figure 8.
- According to the comparison from Figure 3 and 4 we can see, when we increase the numbers from 5k to 10k, after the first iteration, there's definitely more particles around the ground truth position of the robot, which makes the result more reliable, at the same time, the cost is the increment of the time for computation.

Implemented KF tracks the position of tag so closely after convergence that the measurement error from UWB tag affect the KF result which is not expected.

Our calculation process is:

1. Initial position was calculated based on *UWB*

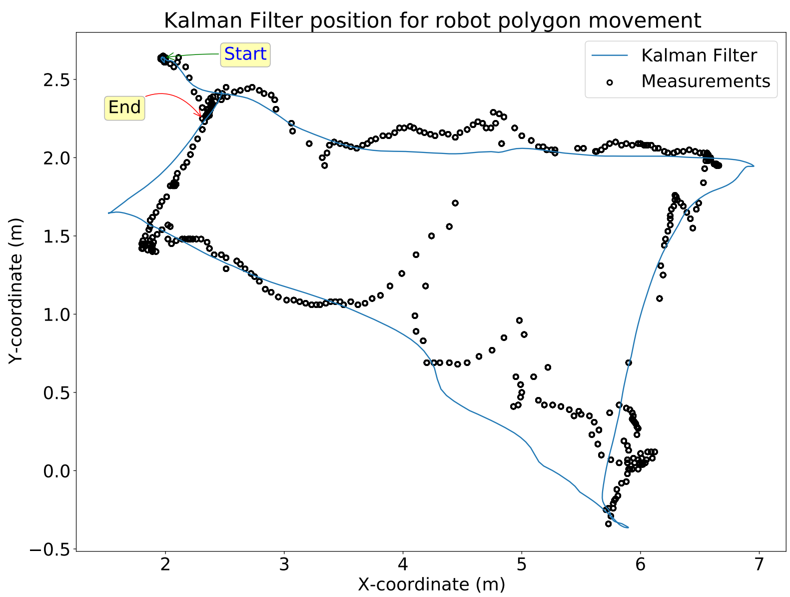

Figure 5: Polygon movement plot with remote control alone inside the lab

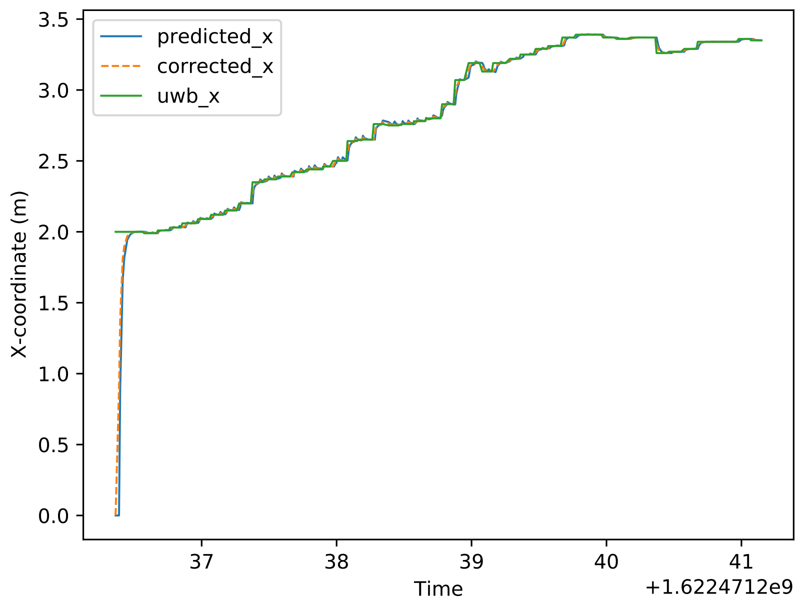

Figure 6: Plot from script *kfMergePlot.py*

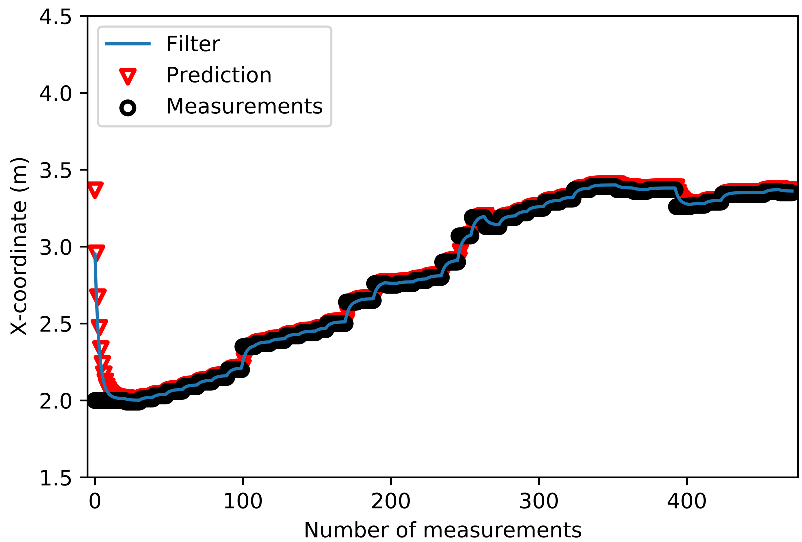

Figure 7: Single movement plot with sensor_var = 0.077 ** 2

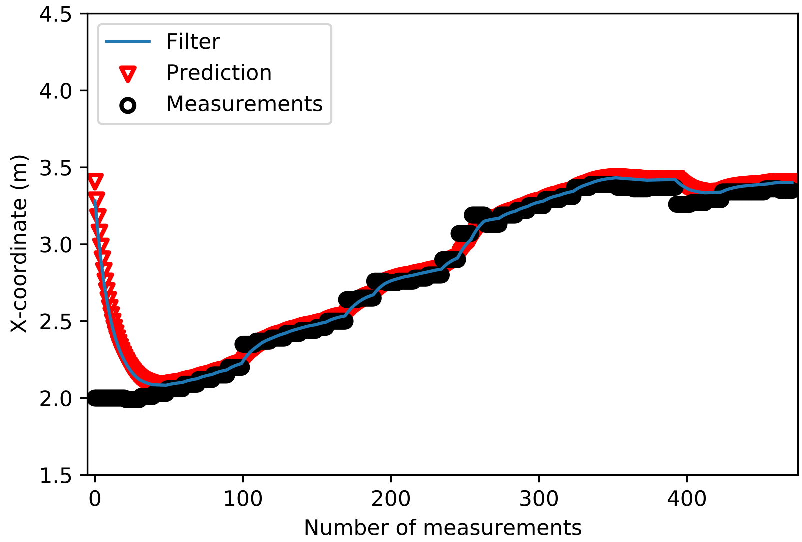

Figure 8: Single movement plot with sensor_var = 0.3 ** 2

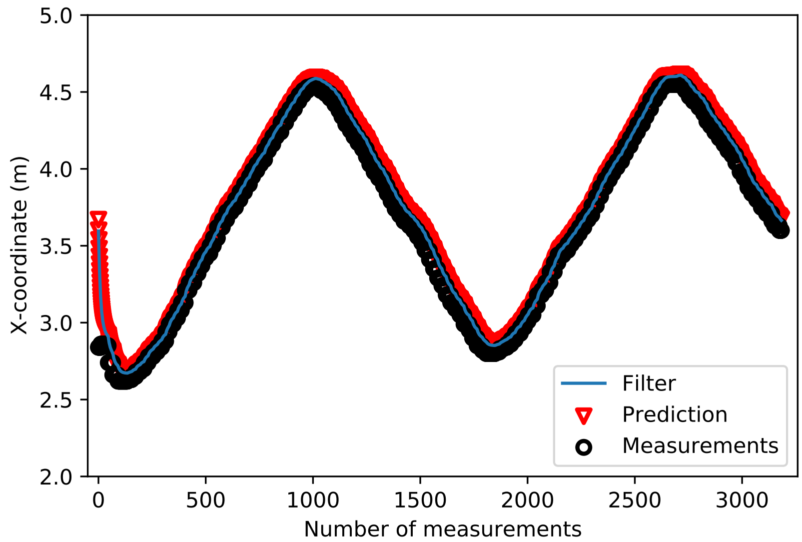

Figure 9: Plot when sensor_var = 0.3 ** 2 for partial round movement with hard-coded velocity

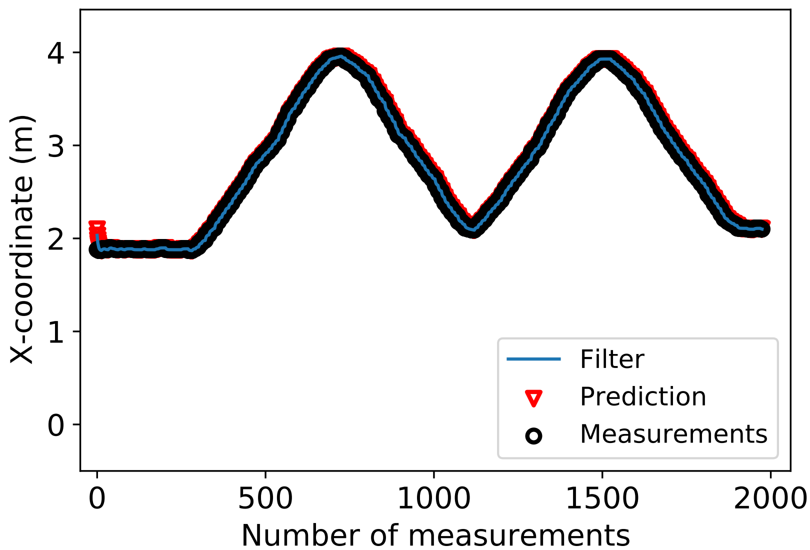

Figure 10: Plot when sensor_var = 0.077 ** 2 for perfect round movement with velocity acquired from rostopic */velocity*

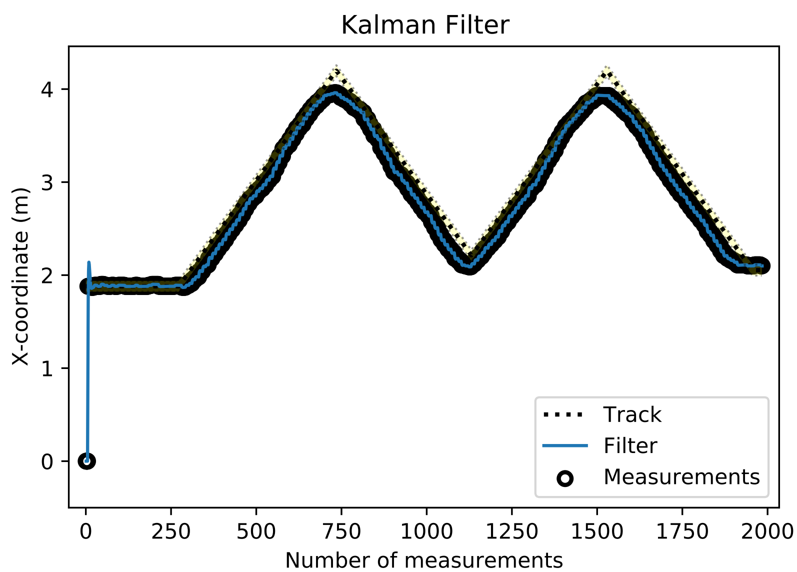

Figure 11: Plot when sensor_var = 0.077 ** 2 for perfect round movement with velocity acquired from rostopic */velocity* and with track as comparison

2. Afterwards, every small step was calculated based on an *internal EKF* with only *IMU* and *Odometry* as input and an external *KF* with fused output from the *EKF* and *UWB tag*

The computation of KF affects the efficiency of robot movement logic more or less, so the better solution is:

1. moving the robot
2. gathering all necessary data in a csv file
3. conducting sensor fusion algorithm

4 Batch Processing

```python
from filterpy.common import Saver
f = pos_vel_filter(x=(uwb_x[ini_index], 0.), R=sensor_var,
    Q=process_var, P=P)
s = Saver(f)
xs, _, _, _ = f.batch_filter(zs, saver=s)
s.to_array()
plt.plot(s.y);
```

Its plot is in Figure 12, we can see noise centered around 0, which proves that the filter is well designed.

5 Smoothing the Results

In order to further eliminate the noise, *rts_smoother()* from *KalmanFilter* was used to smooth the result, its plot is shown in Figure 13:

```python
Ms, Ps, _, _ = f.rts_smoother(Xs, Covs)

book_plots.plot_measurements(zs)
plt.plot(Xs[:, 0], ls='--', label='Kalman Position')
plt.plot(Ms[:, 0], label='RTS Position')
plt.legend(loc=4)
```

6 Two Dimensional Kalman Filter

Our new state vector is $x = \{x,\ \dot{x},\ y,\ \dot{y}\}^T$ which corresponds to x-coordinate, x-velocity, y-coordinate and y-velocity.

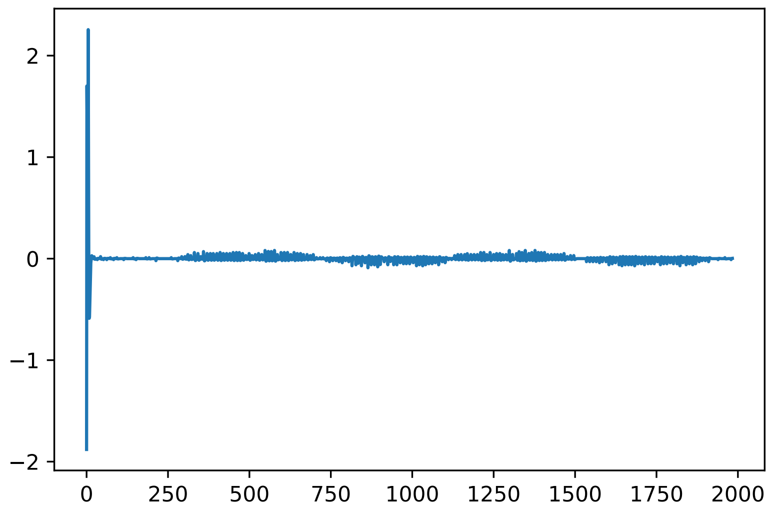

Figure 12: Residual

To verify the performance of KF in two dimension, Python scripts *move_oval.py* and *pentagon_move.py* are used to generate data separately into *oval.csv* and *pentagon.csv* and their corresponding plot are in Figure 14 and 15.

7 Results and Conclusions

Order one KF will work well while the car is moving in a straight line at a constant speed, but cars turn, speed up, and slow down, in which case a second order filter will perform better. A lower order KF can also track a higher order process with higher process noise.

8 Idea for next step

1. Gather coordinates for trace on the ground of the lab as ground truth data
2. Using remote control to move the robot along the trace on the ground of the lab at least ten times and run two dimensional kalman filter
3. Further separate the logic of filter and movement entirely, hide the imple-

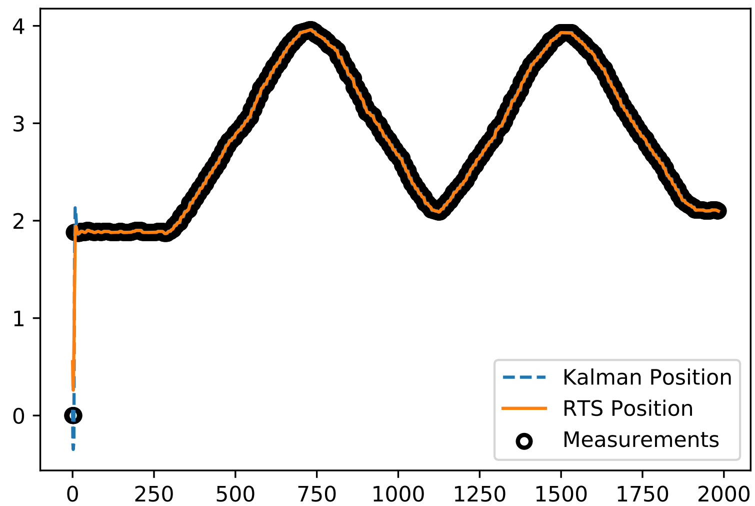

Figure 13: Smoothing result

mentation of movement from filter, no matter movement is through script or remote control.

4. Gather raw and filtered velocity, angular velocity and odometry data etc..

5. Extend one dimensional KF to multi-dimensional/variable KF, UKF and EKF

6. Conduct a long distance movement in corridor and check the performance of sensor fusion algorithm

7. Consider designing a random walk model for the final demonstration

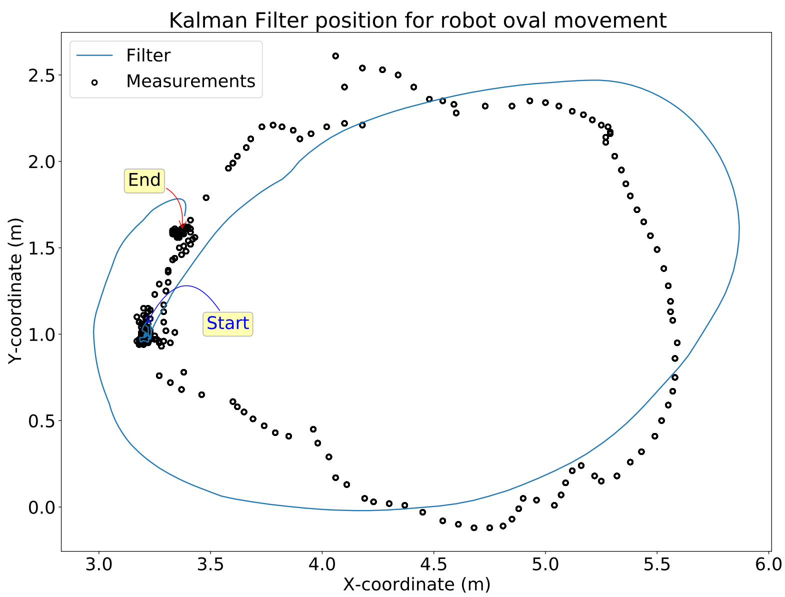

Figure 14: Oval movement

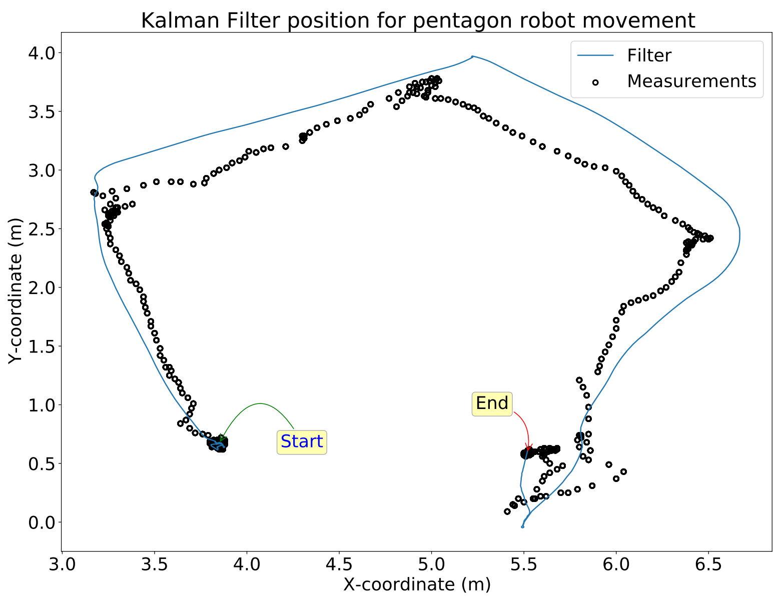

Figure 15: Pentagon movement