

Record For Implementing Kalman Filter

Mingze CHEN

June 10, 2021

1 Implementing One Dimensional Kalman Filter

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

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

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

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

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

```
1   ~/pathTo/catkin_ws$ source ./devel/setup.bash
2   ~/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:

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

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

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

$$\begin{aligned} \text{Current state estimated value} = & \text{Predicted value of current state} + \\ & \text{Kalman Gain} * (\text{measured value} - \text{predicted value of the state}) \end{aligned}$$

which is:

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

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

One sample implementation ¹ can be like:

```
1 from collections import namedtuple
2 gaussian = namedtuple('Gaussian', ['mean', 'var'])
3 gaussian.__repr__ = lambda s: '({:.3f}, {:.3f})'.format(
4     s[0], s[1])
5 def update(prior, measurement):
6     x, P = prior          # mean and variance of prior
7     z, R = measurement    # mean and variance of measurement
8
9     y = z - x             # residual
10    K = P / (P + R)        # Kalman gain
11
12    x = x + K*y            # posterior
13    P = (1 - K) * P        # posterior variance
14    return gaussian(x, P)
15
16 def predict(posterior, movement):
17     x, P = posterior      # mean and variance of posterior
18     dx, Q = movement     # mean and variance of movement
19     x = x + dx
20     P = P + Q
21     return gaussian(x, P)
```

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

Its code snippet is here:

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

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

```

4     new_var = var1 + var2
5     return new_mean, new_var
6
7 # correct step function
8 def correct_step(mean1, var1, mean2, var2):
9     """
10    This function takes in two means and two squared
        variance terms, and return updated gaussian
        parameters.
11    """
12    # calculate the new gaussian parameters
13    new_mean = (var1 * mean2 + var2 * mean1) / (var1 + var2)
14    # also equals to var1 * var2 / (var1 + var2)
15    new_var = 1 / (1 / var1 + 1 / var2)
16    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:

```

1 $ husarion@husarion:~/pathTo/catkin_ws$ rosrun playground
    kalman_filter.py -s kf3105.csv

```

Two experiments were conducted:

1. let the robot move forward 1m, as shown in Figure 1 and 2
2. partial round movement with hard-coded velocity as shown in Figure 4 and perfect round movement with actual velocity from rostopic *velocity* as shown in Figure 5 and 6

R and Q are derived from extensive experiments:

```

1 process_var = 0.028 ** 2
2 sensor_var = 0.077 ** 2

```

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

²sensorData.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'

2 Observation

A bigger sensor measurement variance can make the localisation trajectory from Figure 2 smoother, which is closer to the real movement process, as shown in Figure 3.

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*
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

3 Batch Processing

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

Its plot is in Figure 7, we can see noise centered around 0, which proves that the filter is well designed.

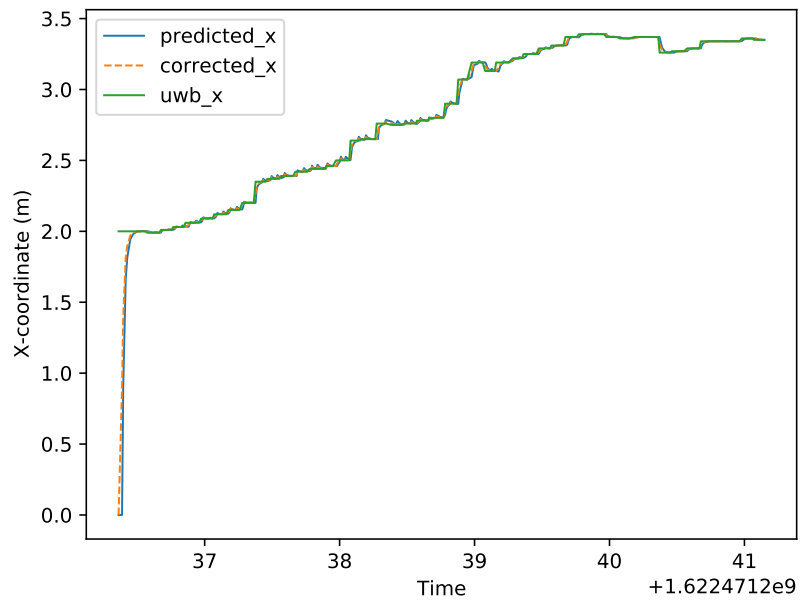


Figure 1: Plot from script *kfMergePlot.py*

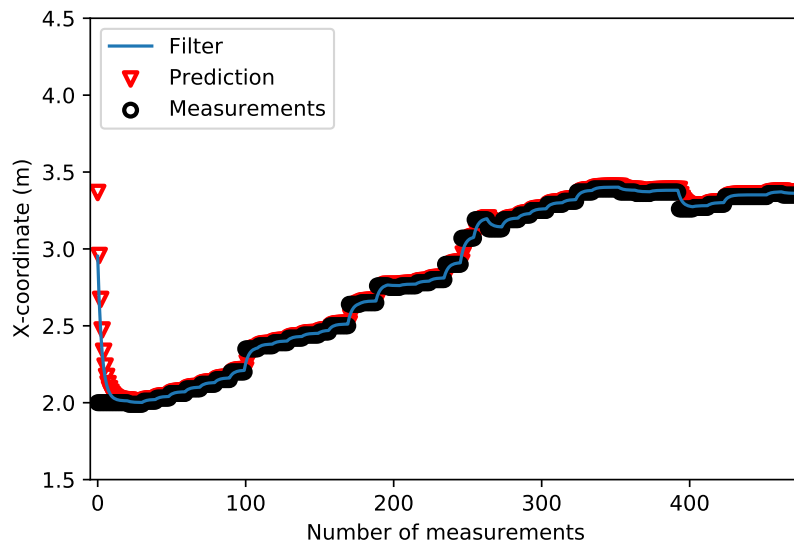


Figure 2: Single movement plot with $\text{sensor_var} = 0.077 ** 2$

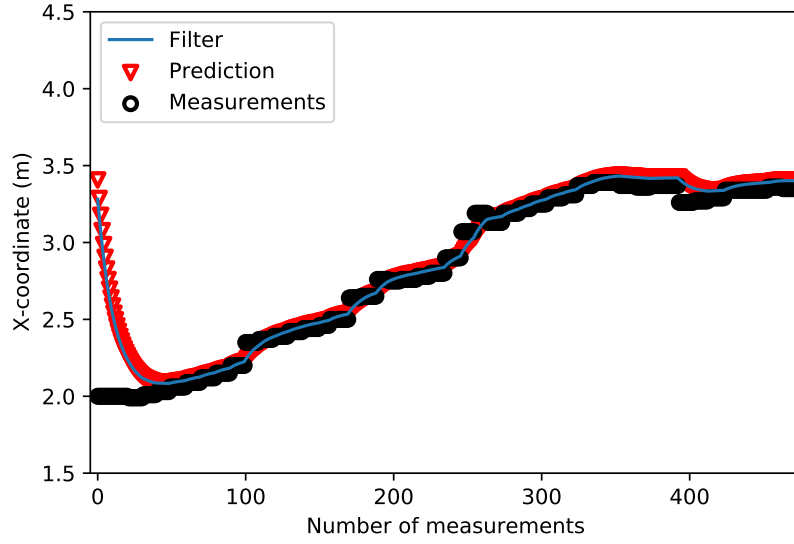


Figure 3: Single movement plot with $\text{sensor_var} = 0.3 \times 2$

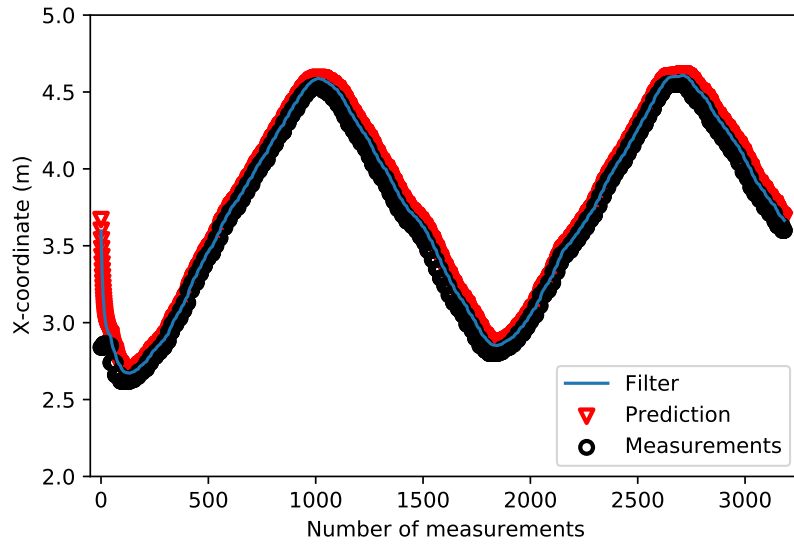


Figure 4: Plot when $\text{sensor_var} = 0.3 \times 2$ for partial round movement with hard-coded velocity

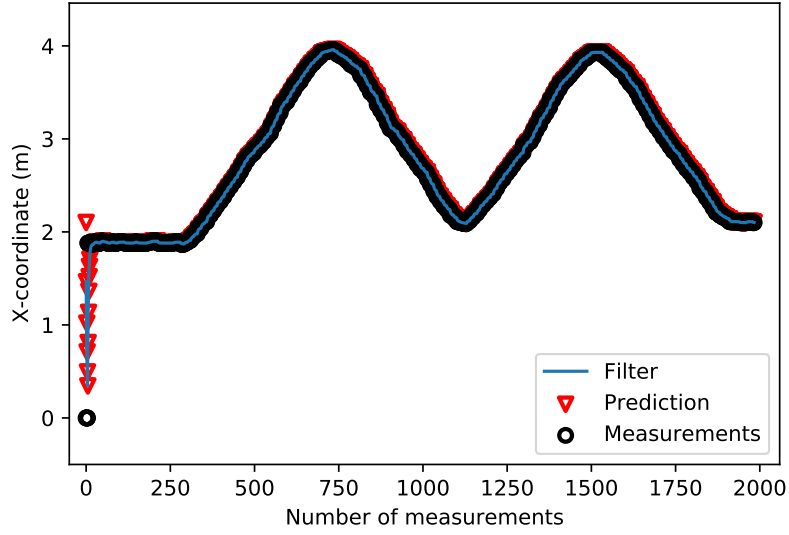


Figure 5: Plot when $\text{sensor_var} = 0.077 ** 2$ for perfect round movement with velocity acquired from rostopic */velocity*

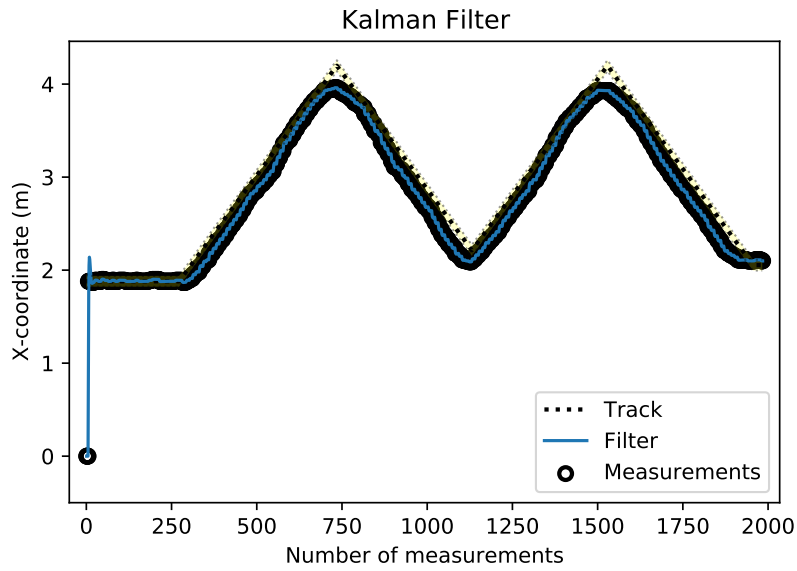


Figure 6: Plot when $\text{sensor_var} = 0.077 ** 2$ for perfect round movement with velocity acquired from rostopic */velocity* and with track as comparison

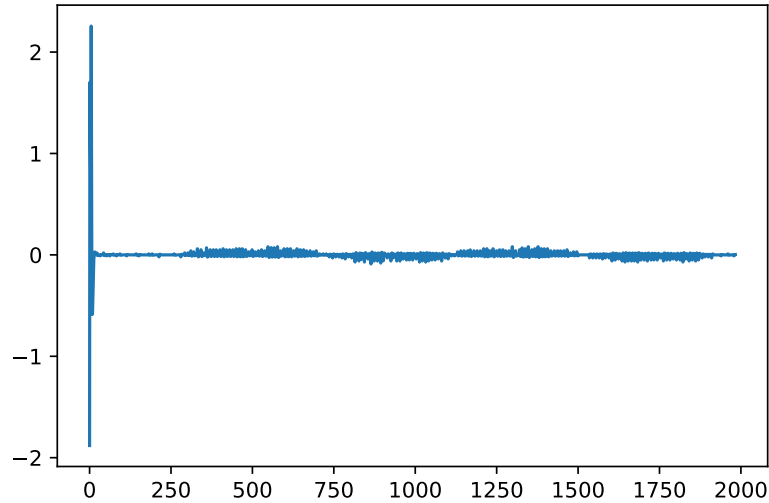


Figure 7: Residual

4 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 8:

```

1 Ms, Ps, _, _ = f.rts_smoother(Xs, Covs)
2
3 book_plots.plot_measurements(zs)
4 plt.plot(Xs[:, 0], ls='--', label='Kalman Position')
5 plt.plot(Ms[:, 0], label='RTS Position')
6 plt.legend(loc=4)

```

5 Idea for next step

1. Further separate the logic of filter and movement entirely, hide the implementation of movement from filter, no matter movement is through script or remote control.
2. Gather raw and filtered velocity, angular velocity and odometry data etc..
3. Extend one dimensional KF to multi-dimensional/variable KF, UKF and

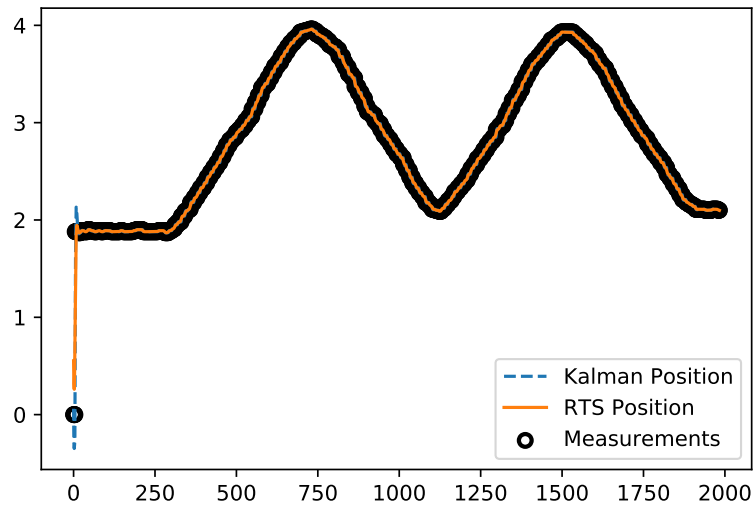


Figure 8: Smoothing result

EKF

4. Conduct a long distance movement in corridor and check the performance of sensor fusion algorithm
5. Consider designing a random walk model for the final demonstration