Mobile Robot Localization using Kalman Filter

Nicholas Ward

February 2022

Abstract

In this project I will be implementing a Kalman Filter given GPS, IMU, and Wheel Encoder data. The objective is that the Kalman Filter will output a smooth and more accurate pose (position and orientation) of the robot. During testing noise was added to the GPS, IMU, and Wheel Encoder data to test the robustness of the Kalman Filter.

1 Introduction

The Kalman filter is one of the most important and commonly used estimation algorithm. They are ideal for systems which are consistently changing. The robot collected continuous GPS, IMU, and Wheel Encoder data allowing for a great introduction on the affects of the Kalman Filter.

2 Kalman Filter Algorithm

In this project I used Python. To properly code the Kalman Filter on a set of data, an understanding must be developed first. I show the main algorithm in Algorithm 1. One step that I took that wasn't on the prompt was that I decided to change the text file into a pandas data-frame. This was done due to personal preference, I have had an abundance of experience with pandas data manipulation so I thought it would be most efficient for me to use it with this project.

I used the numpy library for matrix multiplication, creating matrices with ones, and to find the reciprocal of the section in Algorithm 1/line 17. The numpy library is a convenient python library that allows for almost any type of operation onto a matrix. Extremely helpful in developing the Kalman Filter.

I then created functions to add noise to all the individual data sensors. I used a standard deviation of 0.1 and a mean of 0.5 to create the noise on any part of the data I see fit to show added results on why the Kalman Filter outperforms individual sensors. To see full code visit my GitHub repository.

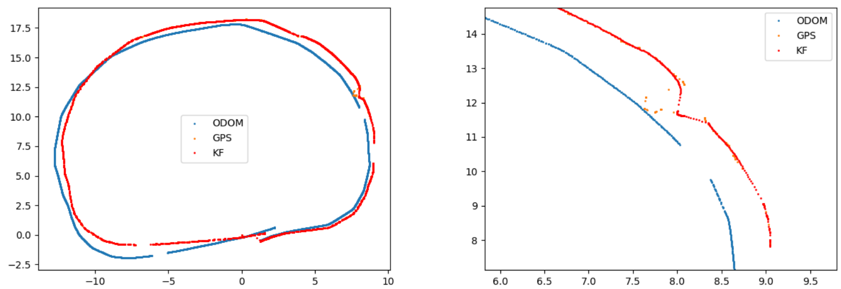
Algorithm 1 Kalman Filter

Change the .txt file into a pandas dataframe
Set new variables equal to columns of the dataframe so that the actual data will not change
Initialize eight different matrices (s.x, s.A, s.Q, s.H, s.R, s.B, s.u, s.P)
for t=1, length of Odom X data **do**
    Update transition matrix A (s.A)
    $$A = \begin{bmatrix} 1 & 0 & delta_t * cos(Odom_{th}(t)) & 0 & 0 \\ 0 & 1 & delta_t * sin(Odom_{th}(t)) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & delta_t \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$
    Update transition matrix R (s.R)
    $$R = \begin{bmatrix} GCX(t) & 0 & 0 & 0 & 0 \\ 0 & GCY(t) & 0 & 0 & 0 \\ 0 & 0 & 0.01 & 0 & 0 \\ 0 & 0 & 0 & ICH(t) & 0 \\ 0 & 0 & 0 & 0 & 0.01 \end{bmatrix}$$
    Kalman filter function call to begin filtering the data
end for
Plot the the results
function KALMAN FILTER CALL($self$)
    Prediction stage for state vector and covariance
    $X^-(k) = A(k)X(k-1)$
    $P^-(k) = A(k)P(k-1)A^T(k) + Q(k)$
    Computer Kalman gain factor
    $K(k) = P^-(k)H^T(k)[H(k)P^-(k)H^T(k) + R(k)]^{-1}$
    Correction stage based on measurement
    $X(k) = X^-(k) + K(k)[Z(k) - H(k)X^-(k)]$
    $P(k) = P^-(k) - K(k)H(k)P^-(k)$
end function

3 Results

In this section I will be showing results that back up the claim that the Kalman Filter (KF) outperforms individual sensor data. Throughout the tests the individual sensor data is GPS, IMU, and Encoder data. I followed this testing procedure when it comes to adding noise to the datasets:

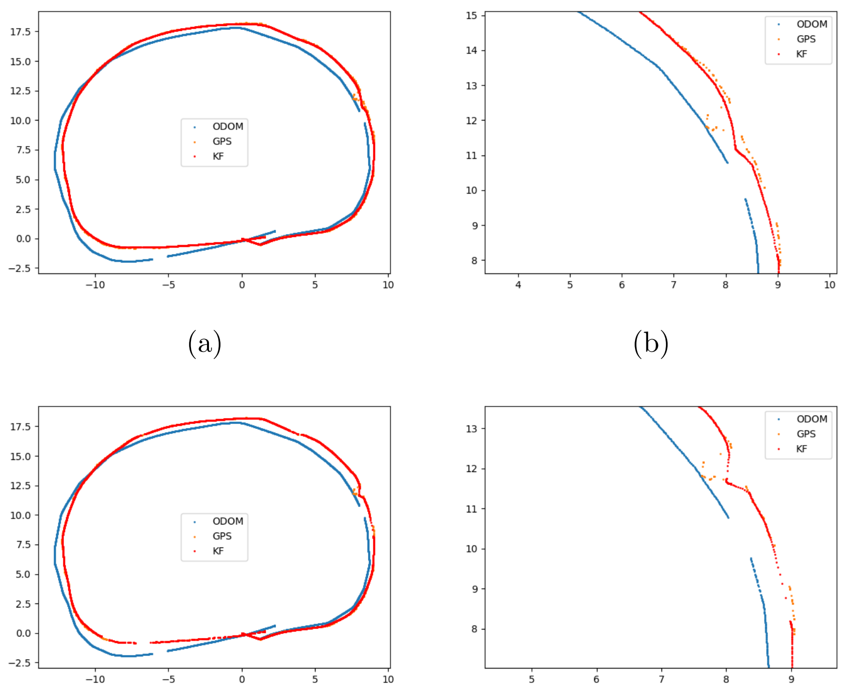
- No noise added to any of the sensor datasets
- Noise added to GPS covariance dataset
- Noise added to certain periods of the GPS covariance dataset
- Noise added to the IMU covariance dataset
- Noise added to certain periods of the IMU covariance dataset
- Noise added to both the GPS position and the covariance dataset
- Noise added to certain periods of the GPS position an the convariance dataset



(a) (b)

Figure 1: The results in (a) and (b) did not have any noise added to the datasets. (b) is a close up of graph (a)

In Figure 1, the results show a new KF plot follows the GPS data plots very closely and is a little bit further away from the ODOM data. But this doesn't tell us much on whether KF outperforms either ODOM or GPS. Figure 1 will just be a baselines test with no noise data. Now lets get into some noise to hopefully better show the power of KF.

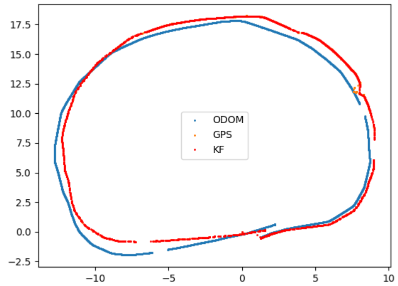


(a) (b)

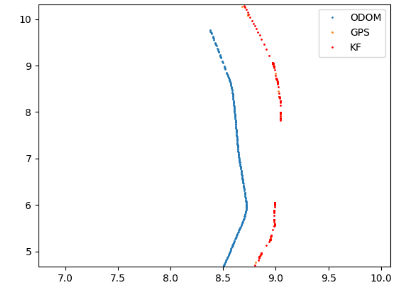
(c) (d)

Figure 2: The results in (a) and (b) had noise added to the entire GPS covariance data. (c) and (d) had the noise added in increments of 1000 to the data

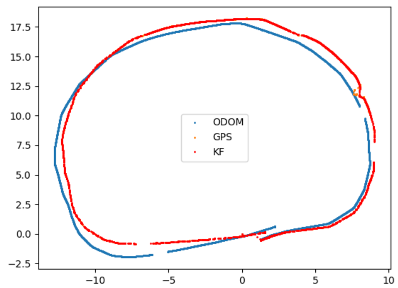
Now we can see a slight change in the GPS data with very slight differences in the amount it scatters on the graph. But the difference is quite hard to tell. But even with the noise in the GPS covariance we can still see that KF still has maintained its line and doesn't have a problem pretty much perfectly maintaining the the original line seen in Figure 1. This shows some promising results on the ability for KF to handle noise.



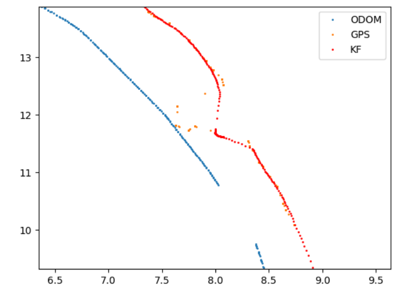
(a)



(b)



(c)

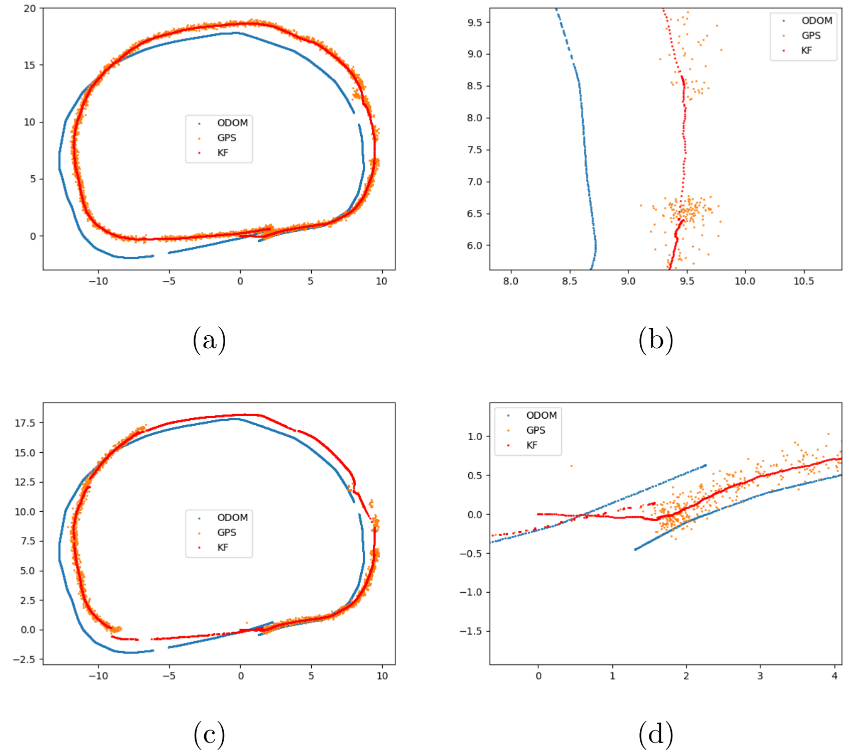


(d)

Figure 3: The results in (a) and (b) had noise added to the entire IMU covariance data. (c) and (d) had the noise added in increments of 1000 to the data

Now with the addition of noise in the entire IMU dataset (Figure 3a, Figure 3b) and the addition of incremental noise to the IMU dataset (Figure 3c, Figure 3d we can start to see some more noticeable changes to the KF plots. The first thing that one can observe is that KF doesn't connect in some places like it used to like when we added The GPS covariance data. It looks more like the original KF plot without noise. I believe this is because the IMU covariance data has less of a weight that the GPS data does on the KF.

Based off of Figure 1, Figure 2, and Figure 3, it seems that adding a little bit of noise allows the KF output to generalize a bit more and close gaps better. We will test this theory next in the next figure by adding noise to both the GPS position data and the GPS covariance data.



(a) (b) (c) (d)

Figure 4: The results in (a) and (b) had noise added to both the GPS covariance data and the GPS position data. (c) and (d) had the noise added in increments of 1000 to the data

The hypothesis that adding more noise to the dataset allows the KF to generalize better seems to hold true by the results found in Figure 4. We can see that any gaps that used to be there are completely gone in Figure 4a and now KF generalizes the trajectory better than it used to. Even in complete gaps in the GPS data we can see KF generalizing a path to connect the gap together as seen in Figure 4b.

All in all, the KF does indeed outperform the single sensor information by a significant amount. Noisy data will most likely occur in raw sensor information, and it can be from a number of reasons. But using a KF seems to be almost a 'must have' in the field of robotics because of it's ability to make smooth and more accurate pose. Without it, we would have a very difficult time with path planning.

4 Apendix

```
'''

Odometry or Encoder data from Seekur Mobile robot
field.O_x is Odometry data in x direction/coordinate of the robot
field.O_y is Odometry data in y direction/coordinate of the robot
field.O_t is Odometry data of the orientation or heading of the robot
(For the covariance of the Odometry data you can give a specific number, for example:
    0.001.)

Data from Microstrain IMU attached on the robot
field.I_t is IMU data of the orientation or heading of the robot
field.Co_I_t  is the IMU data of the Covariance of the orientation of the robot


Novatel DGPS  data attached on the robot
```

```
field.G_x is GPS data in x direction/coordinate of the robot
field.G_y is GPS data in y direction/coordinate of the robot
field.Co_gps_x is GPS data of the Covariance in x direction of the robot
field.Co_gps_y is GPS data of the Covariance in y direction of the robot

'''

from cmath import tan, cos, sin
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt

class KalmanFilter():
    def __init__(self):
        self.text_file = 'EKF_DATA_circle.txt'
        self.data = 0

        self.time = []

        self.Odom_x = []
        self.Odom_y = []
        self.Odom_theta = []

        self.Gps_x = []
        self.Gps_y = []

        self.Gps_Co_x = []
        self.Gps_Co_y = []

        self.IMU_heading = []
        self.IMU_Co_heading = []

        self.matrix_z = []

        self.velocity = 0.14
        self.dist_wheels = 1
        self.omega = 0
        self.total = 0
        self.delta_t = 0.001

        #storing the data for kalman filter
        self.true = []
        self.X1 = []
        self.X2 = []
        self.X_heading = []

    def begin_kalman(self):
        print(self.total)
        for i in range(self.total):
            self.matrix_a[0,2] = self.delta_t * cos(self.Odom_theta[i])
            self.matrix_a[1,2] = self.delta_t * sin(self.Odom_theta[i])

            self.matrix_R[0,0] = self.Gps_Co_x[i]
            self.matrix_R[1,1] = self.Gps_Co_y[i]
            self.matrix_R[3,3] = self.IMU_Co_heading[i]

            self.matrix_z = np.array(
                [[self.Gps_x[i]],
                [self.Gps_y[i]],
```

```
                [self.velocity],
                [self.IMU_Co_heading[i]],
                [self.omega]]
            )
            #put kalman filter function here
            X_f = self.kalman_filter()
            self.X1.append(X_f[0][0])
            self.X2.append(X_f[1][0])
            self.X_heading.append(X_f[3][0])

        plt.plot(self.Odom_x, self.Odom_y,'.', label='ODOM', markersize = 2)
        plt.plot(self.Gps_x, self.Gps_y, '.', label='GPS', markersize = 2)
        plt.plot(self.X1, self.X2, '.', color='red', label='KF',markersize = 2)
        plt.legend()
        plt.show()

    def kalman_filter(self):
        #prediction stage for state vector and co  variance

        X_neg = np.dot(self.matrix_a, self.initial_states)
        #self.initial_states = X_neg
        P = np.dot(np.dot(self.matrix_a, self.matrix_P), np.transpose(self.matrix_a)) + self.matrix_q

        #compuute kalman gain factor
        input = np.dot(np.dot(self.matrix_H, P), np.transpose(self.matrix_H)) + self.matrix_R
        inverse_input = np.linalg.inv(input)
        K = np.dot(np.dot(P, np.transpose(self.matrix_H)), inverse_input)

        #correctionstage base on measurement
        matrix_y = np.dot(self.matrix_H, X_neg)
        X_f = X_neg + np.dot(K, (self.matrix_z - matrix_y))
        self.initial_states = X_f
        self.matrix_P = P - np.dot(np.dot(K, self.matrix_H), P)
        return X_f

    def create_dataframe(self):
        #convert the text file into a pandas dataframe

        read_file = pd.read_csv(self.text_file)
        read_file.to_csv(self.text_file + '.csv', index=None)

        self.data = read_file

    def get_data(self):
        self.time = self.data['%time']
        self.Odom_x = self.data['field.O_x']
        self.Odom_y = self.data['field.O_y']
        self.Odom_theta = self.data['field.O_t']

        self.Gps_x = self.data['field.G_x']
        self.Gps_y = self.data['field.G_y']

        self.Gps_Co_x = self.data['field.Co_gps_x']
        self.Gps_Co_y = self.data['field.Co_gps_y']

        self.IMU_heading = self.data['field.I_t']
        self.IMU_Co_heading = self.data['field.Co_I_t']

```

```
        self.omega = self.velocity * tan(self.Odom_theta[1]) / self.dist_wheels

        #matching with robot's heading initially

        self.IMU_heading = self.IMU_heading + (0.32981-0.237156) * np.ones((len(self.IMU_heading), ), dtype=int)

        self.initial_states = np.array(
            [[self.Odom_x[0]],
            [self.Odom_y[0]],
            [self.velocity],
            [self.Odom_theta[0]],
            [self.omega]]
        )

        self.total = len(self.Odom_x)

    def noise_gps_inc(self):
        noise_mean = 0.5
        noise_std = 0.12
        gps_noise = noise_std * np.random.randn(len(self.Odom_x), 2) +  noise_mean * np.ones((len(self.Odom_x), 2))

        for i in range(1000):
            self.Gps_x[i] += gps_noise[i][0]
            self.Gps_y[i] += gps_noise[i][1]
            self.Gps_Co_x[i] += gps_noise[i][0]
            self.Gps_Co_y[i] += gps_noise[i][1]

        for i in range(2000,3000):
            self.Gps_x[i] += gps_noise[i][0]
            self.Gps_y[i] += gps_noise[i][1]
            self.Gps_Co_x[i] += gps_noise[i][0]
            self.Gps_Co_y[i] += gps_noise[i][1]


    def noise_gps(self):
        noise_mean = 0.5
        noise_std = 0.12
        gps_noise = noise_std * np.random.randn(len(self.Odom_x), 2) +  noise_mean * np.ones((len(self.Odom_x), 2))

        self.Gps_x = self.data['field.G_x'] + gps_noise[:,0]
        self.Gps_y = self.data['field.G_y'] + gps_noise[:,1]

        self.Gps_Co_x = self.data['field.Co_gps_x'] + gps_noise[:,0]
        self.Gps_Co_y = self.data['field.Co_gps_y'] + gps_noise[:,1]

    def noise_odom(self):
        noise_mean = 0.5
        noise_std = 0.1

        odom_noise = noise_std * np.random.randn(len(self.Odom_x), 2) +  noise_mean * np.ones((len(self.Odom_x), 2))

        self.Odom_x = self.data['field.O_x'] + odom_noise[:,0]
        self.Odom_y = self.data['field.O_y'] + odom_noise[:,1]

    def noise_imu(self):
        noise_mean = 0.5
```

```
        noise_std = 0.1

        imu_noise = noise_std * np.random.randn(len(self.Odom_x), 2) +  noise_mean *
    np.ones((len(self.Odom_x), 2))

        #for i in range(1000):
         #   self.IMU_Co_heading += imu_noise[i][0]

        #for i in range(2000,3000):
           # self.IMU_Co_heading += imu_noise[i][0]

        self.IMU_Co_heading = self.data['field.Co_I_t'] + imu_noise[:,0]

    def define_matrix(self):
        self.matrix_a = np.array(
            [[1, 0, self.delta_t*cos(self.Odom_theta[0]), 0, 0],
            [0, 1, self.delta_t*sin(self.Odom_theta[0]), 0, 0],
            [0, 0, 1,                                    0, 1],
            [0, 0, 0,                                    1, self.delta_t],
            [0, 0, 0,                                    0, 1]]
        )

        self.matrix_q = np.array(
            [[0.0004, 0, 0, 0, 0],
            [0, 0.0004, 0, 0, 0],
            [0, 0, 0.001, 0, 0],
            [0, 0, 0, 0.001, 0],
            [0, 0, 0, 0, 0.001]]
        )

        self.matrix_H = np.array(
            [[1,0,0,0,0],
            [0,1,0,0,0],
            [0,0,1,0,0],
            [0,0,0,1,0],
            [0,0,0,0,1]]
        )

        self.matrix_R = np.array(
            [[.04,  0,   0,   0,   0],
            [0,  .04,  0,   0,   0],
            [0,   0,  .01 , 0,   0],
            [0,   0,   0,   0.01,  0],
            [0,   0,   0,   0,  .01]]
        )

        self.matrix_B = np.array(
            [[1,   0,   0,   0,   0],
            [0,   1,   0 ,  0 ,  0],
            [0,   0,   1,   0,   0],
            [0 ,  0,   0,   1,   0],
            [0,   0,   0,   0,   1]]
        )

        self.matrix_u = np.array([[0],[0],[0],[0],[0]])

        self.matrix_P = np.array(
            [[.001,  0,   0,   0,   0],
            [0,  .001,  0,  0,   0],
            [0,   0,  .001,  0,   0],
```

```
            [0,   0,   0,  .001,  0],
            [0,   0,   0,   0,  .001]]
        )

    def plot_data(self):
        fig, ax = plt.subplots(4)
        ax[0].plot(self.time, self.Odom_theta)
        ax[1].plot(self.time, self.data['field.I_t'])
        ax[2].plot(self.Odom_x, self.Odom_y)
        ax[3].plot(self.time, self.IMU_heading)
        plt.show()

if __name__ == '__main__':
    filter = KalmanFilter()
    filter.create_dataframe()
    filter.get_data()
    #filter.noise_gps()
    filter.noise_gps_inc()
    #filter.noise_imu()
    #filter.noise_odom()
    #filter.plot_data()
    filter.define_matrix()

    filter.begin_kalman()
    filter.kalman_filter()
```