Extended Kalman Filter

Bradley Selee

October 27, 2022

1 Introduction

The purpose of this project was to implement an extended kalman filter (EKF) on a sinusoidal model. We were given a set of set of sinusoidal data with actual and measured position values. The measured values were used to process the EKF and the true values were used to compare the output of our EKF.

EKFs are similar to kalman filters but create models in the non-linear domain. The basic goal of the EKF is to estimate a position in non-linear unknowns. Like the kalman filter, once an EKF is created, the dynamic noise, Q, and measure noise, R, can be treated as hyperparameters and adjusted to fit the data. Adjusting the $\frac{Q}{R}$ ratio leads to a more accurate filter. In this project, once the filter is created, three $\frac{Q}{R}$ ratios are used and their outputs are discussed.

2 Methods and Materials

For this project, my implementation was created solely using Python. The only libraries used were NumPy for matrix operations and to load in the data, and Matplotlib to plot all the graphs. The code was developed in Windows Subsystem for Linux (WSL) using Visual Studio Code as the text editor. My code implementation is stored on GitHub in a private repository.

2.1 Code Design and Implementation

In my project, one class for the EKF was created. This class contains an attribute for each matrix and sets the initial conditions of the EKF. There are also two methods, one for the prediction step and one for the update step. The primary code loops through one pass of the dataset, performing a prediction and update step. After each iteration, the estimated position is saved and finally plotted at the end of the dataset.

2.2 Data

There was one given dataset for this project, a two column text file which represents a sinusoid. The first column is the true value, and the second column is the measured value.

The data contains 780 samples.

3 Results

There was only one part for this project, which was to implement EKF for a sinusoidal model. The results show a line plot of the estimated values, overlapped on scatter plot with the actual values. There are three plots shown, one for each different $\frac{Q}{R}$ ratio

3.1 EKF: Fine-tuned

Figure 1 shows the EKF estimated positions for the $\frac{Q}{R}$ ratio described in equation 1. For a fine-tuned filter, the graph shows very accurate position estimates with some inaccuracies.

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1.1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 0.15 \end{bmatrix} \tag{1}$$

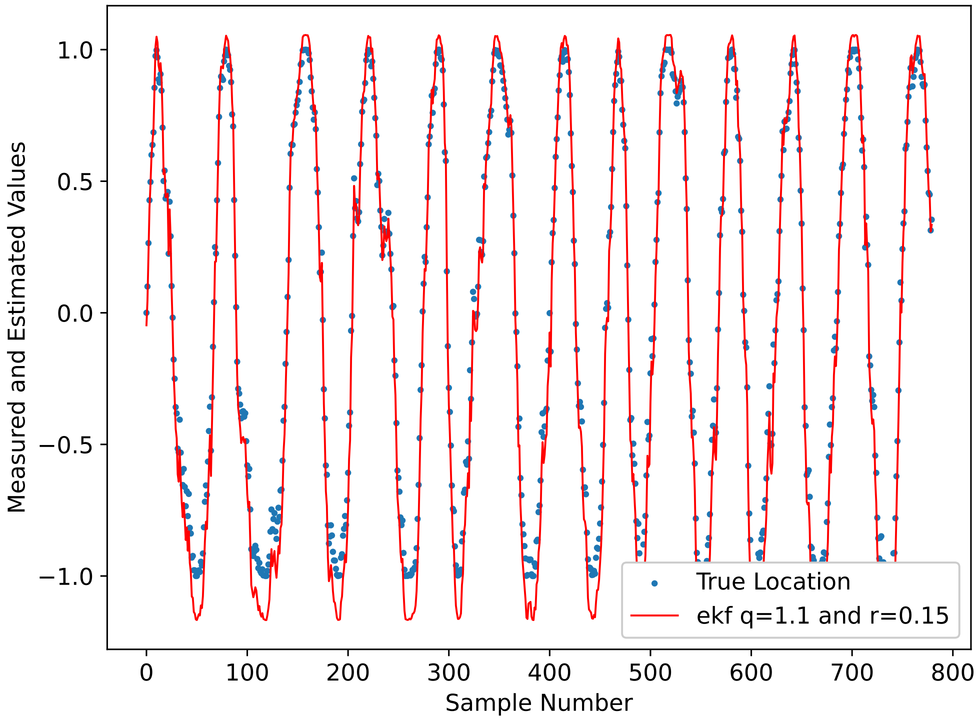

Figure 1: Extended Kalman Filter estimated predictions line plot on top of a scatter plot of the true position. This filter was adjusted for fine-tuned Q and R values. Color was used in order to see the true values.

3.2 EKF: High Measure Noise

Figure 2 shows the EKF estimated positions for the $\frac{Q}{R}$ ratio described in equation 2. This result demonstrates the estimation error when the measure noise is high but the dynamic noise is similar to the fine-tuned value. From the graph, it appears the estimation is close to the correct position and sinusoidal shape, but the filter is overshooting the estimation on the extremes.

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.8 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 2.0 \end{bmatrix} \tag{2}$$

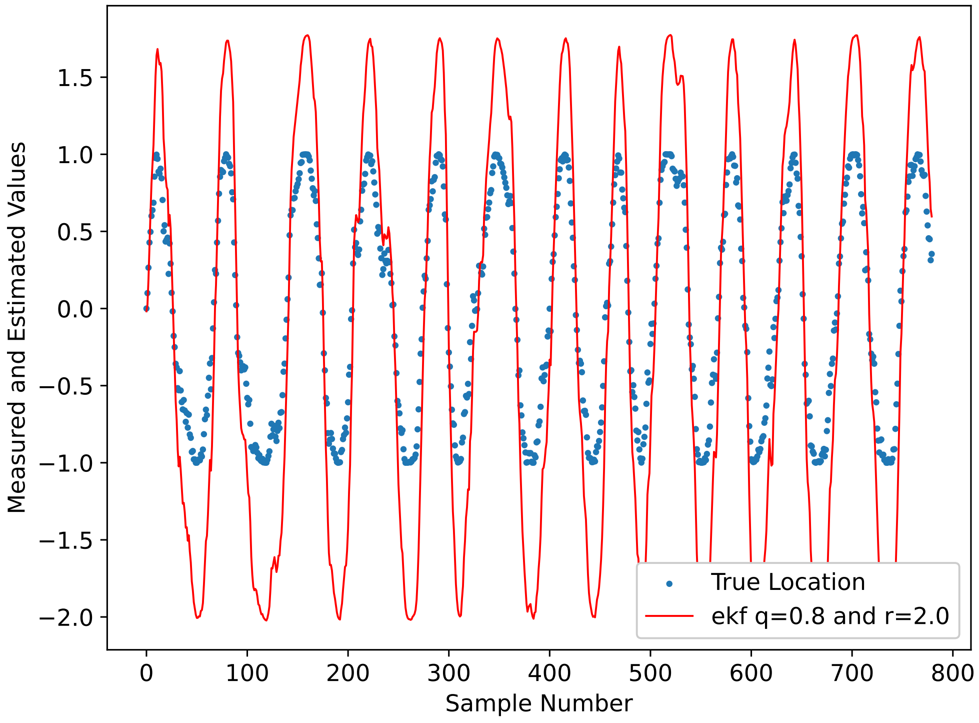

Figure 2: Extended Kalman Filter estimated predictions line plot on top of a scatter plot of the true position. This filter was adjusted to have high measure noise.

3.3 EKF: Low Dynamic Noise

Figure 3 shows the EKF estimated positions for the $\frac{Q}{R}$ ratio described in equation 3. This result demonstrates the estimation error when the measure noise is high and very little dynamic noise exists. From the graph, it appears the filter estimation is able to understand that the data oscillates, but the estimates are no where near the correct positions nor the actual shape of the actual data. The filter is unable to respond to the measured values.

$$Q = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0.0001 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad R = \begin{bmatrix} 5.0 \end{bmatrix} \tag{3}$$

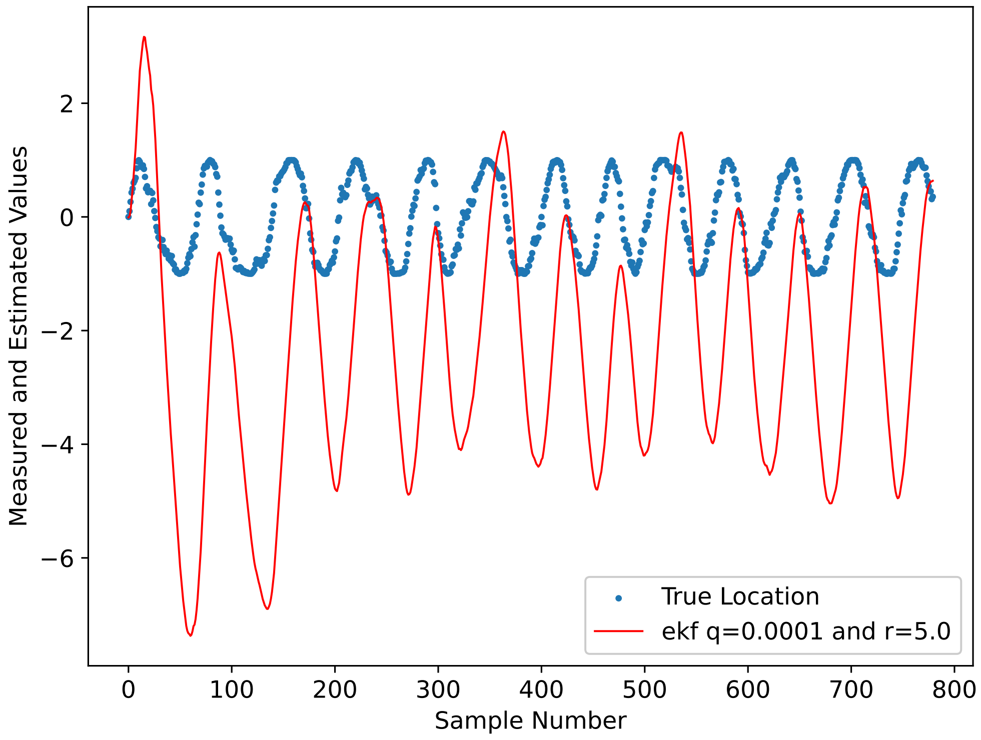

Figure 3: Extended Kalman Filter estimated predictions line plot on top of a scatter plot of the true position. This filter was adjusted to have high measure noise and very little dynamic noise.

4 Conclusion

Overall, I think this project was successful. My implementation creates an EKF from the given data. The user can adjust the initial Q and R ratios to allow for fine tuning for the specific scenario/dataset. Finally, my implementation shows extreme and normal ranges of Q/R ratios. This implementation is modular non-linear models and can estimate position to a large degree. This project does have a few limitations. Depending on the type of data, another estimation approach may be more accurate than the EKF. While this filter holds true in the linear and non-linear domain, a regular kalman filter may be more accurate in the linear domain. If I were to do this project again, I would have adjusted the dynamic noise a little more. The low dynamic noise does show a clear trend of not adapting to the measured values, but something seems slightly off which I could not figure out.

5 Appendix

The following appendix is the Python code for this lab.

```
"""lab5.py

Bradley Selee
ECE 8540
Project 5
Due: 10/27/2022
"""
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from numpy.random import randn

# Matplotlib settings
# Axis font size should be a similar to your paper font size
plt.rcParams.update({'font.size': 12})

class ExtendedKalmanFilter:
    def __init__(self, n_noise = 0.1, r_init=0.01, q_init=0.01):
        self.T = 1 # sampling time
        self.init_state = lambda x: (1/10) * np.cos(x/10)
        self.x_t = np.array([[0], # initial state
                            [0],
                            [0]])
        self.y_y = np.array([0])
        self.df_dx = np.array([[1, self.T, 0],
                             [0, 1, 0],
                             [self.init_state(self.x_t[0,0]), self.T, 0]])
        self.df_da = np.array([[0, 0, 0],
                             [0, 1, 0],
                             [0, 0, 0]])

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

        # Dynamic noise covariance
        self.Q = np.array([[0, 0, 0],
                         [0, q_init, 0],
                         [0, 0, 0]])

        # observation equation g(h, n) = h + n
        self.g_func = lambda h: h + n_noise

        # Measurement noise covariance
        self.R = np.array([r_init])
        # State covariance matrix - Initialized to identity matrix
        self.S = np.array(np.identity(self.dg_dx.shape[1]))# (P)

    def predict(self):

        # Calculate error covariance
        self.S = np.dot(self.df_dx, np.dot(self.S, self.df_dx.T)) +
           np.dot(self.df_da, np.dot(self.Q, self.df_da.T))

        return self.x_t[0]

    def update(self, y):
        """Update the state variables

        Args:
           y: The new measured value
        """
        # Intermediate step
        J = np.dot(self.dg_dx, np.dot(self.S, self.dg_dx.T)) + np.dot(self.dg_dn,
           np.dot(self.R, self.dg_dn.T))

        #print(J)
        #exit()
        # Kalman Gain
        K = np.dot(np.dot(self.S, self.dg_dx.T), np.linalg.inv(J))

        #print(self.g_func(self.x_t[2,0]))

        # Updated state

        self.x_t = self.x_t + K * (y - self.g_func(self.x_t[2,0]))

        # update df_dx at each iteration but i'm not sure if its needed (I don't
           think it is)
        #self.df_dx[2, 0] = self.init_state(self.x_t[0, 0])

        # Update state (error) covariance matrix
        I = np.identity(self.dg_dx.shape[1])
        self.S = np.dot(I - (np.dot(K, self.dg_dx)), self.S)
        return self.x_t[0] # Return estimated x and y


def plot_ekf(measurements, estimates, test_case, plot_title='ekf_filter',
    fig_name=None):
    fig = plt.figure(figsize=(8,6))
    line_x = np.arange(0, measurements.shape[0])
    plt.scatter(line_x, measurements, label='True Location',s=5)
    #plt.title(plot_title) # Suggested not to put title plots
    plt.xlabel('Sample Number')
    plt.ylabel('Measured and Estimated Values')
    plt.plot(line_x, estimates, color='red', label=f'ekf q={test_case[1]} and
       r={test_case[0]}', linewidth=1)
    plt.legend(loc="lower right")
    if fig_name:
        fig.savefig(fig_name, bbox_inches='tight') # bbox=tight trims white space
           around image


def main():
    sin_data_file = 'sin-data.txt'

    # Part 2 lists for plotting

    # Load sine data
    sin_data = np.loadtxt(sin_data_file)
    sin_data_actual = sin_data[:,0]
    sin_data_measured = sin_data[:,0]
    data_len = len(sin_data)

    fig_name_ext = ['fine_tuned', 'high_meas', 'no_meas']
    # EKF Loop  # (R, Q)
    test_cases = [(0.15, 1.1), # Fine-tuned R/Q Ratio
                  (2.0, 0.8), # High measure noise
                  (5.0, 0.0001)] # Low dynamic noise - hard to find the shape
    pred_x = [[] for _ in range(len(test_cases))] # list used for predictions for
       each test case
    np.set_printoptions(suppress=True)

    for index, test in enumerate(test_cases):
        ekf = ExtendedKalmanFilter(n_noise=0.05, r_init=test[0], q_init=test[1])
        for i in range(data_len):
            ekf.predict()
            new_measured_position = sin_data_measured[i]
            x = ekf.update(new_measured_position)
            pred_x[index].append(x[0]) # single 2d array so we need to index it
               again
        plot_ekf(sin_data_actual, pred_x[index], test,
            fig_name=f'ekf_{fig_name_ext[index]}.eps')

    # Show all 3 plots
    plt.show()
```

```
if __name__ == '__main__':
    main()
```

References

No further references were needed for this project.