Introduction to Data Assimilation

Kalman Filter

Name: Hla Hany Mohamed Helmy Ahmed

Student ID: 612208001

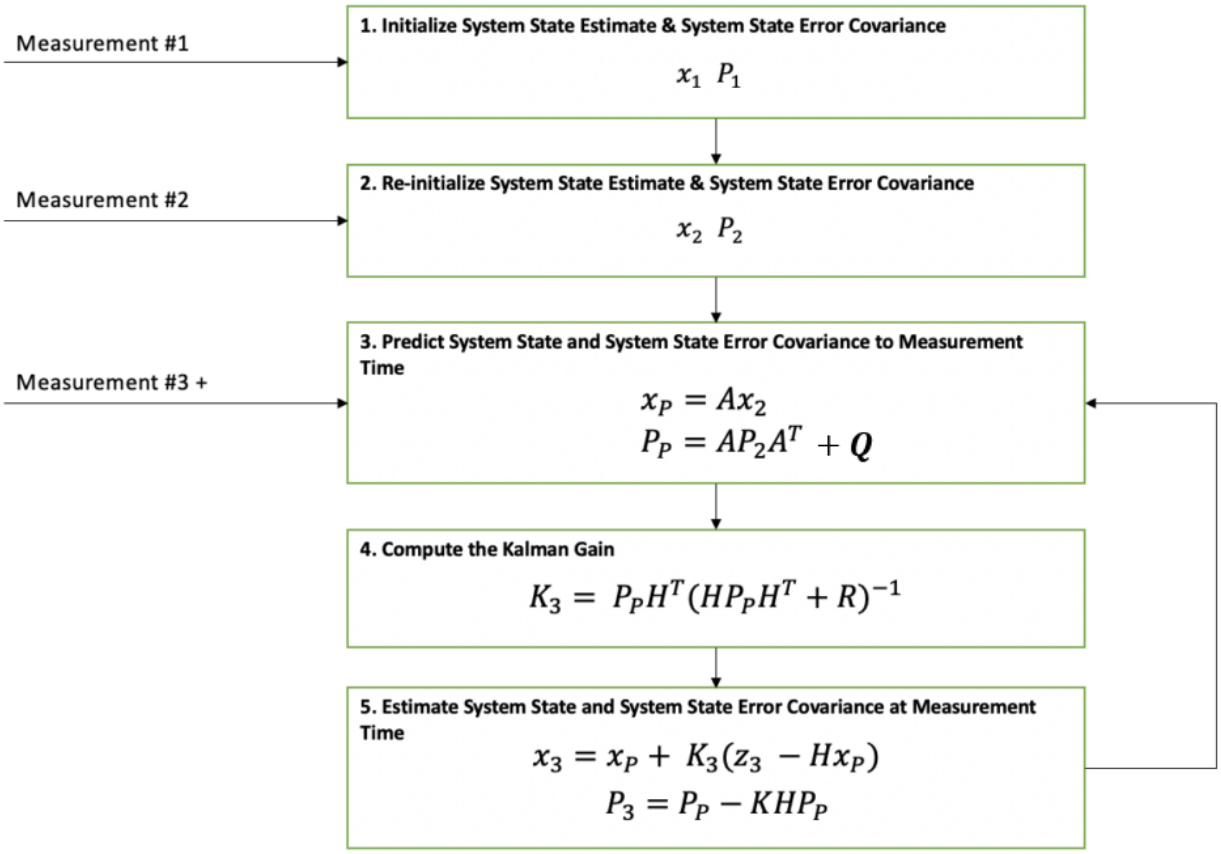
Submitted to: Prof. Serge Richard

Abstract

This report explains how to design and implement a Kalman filter using Python programming. If we look at the Kalman Filter as a black box, we will find that it has inputs and outputs. Give the sequence of noisy inaccurate measurements as inputs, the outputs are more accurate estimates. In a nutshell, Kalman Filter can be regarded as a generic algorithm that can estimate observable and unobservable system parameters with improved accuracy in real-time. These estimates allow systems greater control and thus more capabilities. For example, Kalman filter can be used for object tracking. In other word, it estimates position and velocity from less accurate position measurements. Moreover, it can be used to estimate the body weight on digital scale by tailoring the measured pressure on the surface. One major disadvantage of Kalman Filter is its limitation to linear systems, while the real-life models are nonlinear. However, they have the advantage that they are light on memory as they don't need to keep any history other than the previous state. In this paper, we will discuss one simple example of Kalman Filter on Python, it estimates the velocity from position.

Table of Contents

Flow Diagram of the Kalman Filter Algorithm



x	state variable	n x 1 column vector	Output
P	state covariance matrix	n x n matrix	Output
z	measurement	m x 1 column vector	Input
A	state transition matrix	n x n matrix	System Model
H	state-to-measurement matrix	m x n matrix	System Model
R	measurement covariance matrix	m x m matrix	Input
Q	process noise covariance matrix	n x n matrix	System Model
K	Kalman Gain	n x m	Internal

Kalman Filter Python Implementation

Compute Measurements

For testing a Kalman Filter, input data is required. In this example, the input(...) function is used to provide position measurements of a car moving with a constant velocity of 50 m/s. It initially sets the position to 0 and the velocity at 50 m/s. Next, random noise, v, is calculated and added to position measurement. Also, another random noise, w, is computed and added to the car velocity accounting for random accelerations. Finally, the current position and current velocity are saved for the next measurement step.

```
def input(updateNumber):
    if updateNumber == 1:
      input.currentPosition = 0
      input.currentVelocity = 50 #in m/s

    dt = 0.1

    w = 7 * np.random.randn(1)
    v = 7 * np.random.randn(1)

    z = input.currentPosition + input.currentVelocity*dt + v
    input.currentPosition = z - v
    input.currentVelocity = 50 + w
    return [z, input.currentPosition, input.currentVelocity]
```

Filter Measurements

The first measurement includes the following:

- **z:** position measurement
- **R:** covariance matrix of the measurement noise.
- **T:** measurement timestamp

$$\boldsymbol{z = pos_m} \qquad \boldsymbol{R = \sigma^2_{pos_m}} \qquad \boldsymbol{t = t_m}$$

After the first measurement, we get the following outputs:

- **x:** state vector for position and velocity
- **P:** state covariance matrix representing the uncertainty in x
- **T:** estimate timestamp

$$\boldsymbol{x = \begin{bmatrix} pos \\ vel \end{bmatrix}} \qquad \boldsymbol{P = \begin{bmatrix} \sigma^2_{pos} & \sigma_{posVel} \\ \sigma_{posVel} & \sigma^2_{vel} \end{bmatrix}} \qquad \boldsymbol{T = t_m}$$

- **A** is the state transition matrix for a system model that assumes constant linear motion.
- **H** is the state-to-measurement matrix used to convert the system state estimate from the state space to the measurement space. The H matrix is a simple matrix that is set up to reduce the state estimate and error covariance to position only values rather than position and velocity.
- **Q** is the 2x2 covariance matrix associated with the noise in states. It represents the idea/feature that the state of the system changes over time. The systems actual accelerations and decelerations contribute to this error.

$$\boldsymbol{H = \begin{bmatrix} 1 & 0 \end{bmatrix}} \qquad \boldsymbol{A = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix}} \qquad \boldsymbol{Q = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}}$$

Other examples can have a second position measurement to estimate the velocity with a linear approximation because we know that velocity is the change of the object's position over a specified time interval. However, the goal is to have a system state estimate that can be updated for future measurement with the Kalman Filter equations, no matter how different the system initialization is.

First, the predicted state, x_p, and predicted state covariance matrix, P_p, are computed by propagating the existing state and state covariance in time to align with the new measurements. P_p tells us the variability of the state x_p.

$$\boldsymbol{x_p = Ax = \begin{bmatrix} pos + vel.\Delta T \\ vel \end{bmatrix}}$$

$$\boldsymbol{P_p = APA^T + Q = \begin{bmatrix} 1 & \Delta T \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_{pos}^2 & \sigma_{posVel} \\ \sigma_{posVel} & \sigma_{vel}^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \Delta T & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix}}$$

For this example, the measurements are taken every 10 seconds, so $\Delta T = 0.1\ s$.

The Kalman Filter computes Kalman Gain which is used to determine how much the input measurements influence the system state estimate. The Kalman gain is the weight given to the measurements and current state estimate. To clarify, when a noisy measurement is taken to update the system state, the Kalman Gain will trust its current state estimate more than this new inaccurate information in order to form an optimal estimate.

$$\boldsymbol{K = P_p H^T (HP_p H^T + R)^{-1}}$$

Lastly, the Kalman Gain, K, is used to compute the new state, x, and state covariance estimate, P.

$$\boldsymbol{x = x_p + K(z - Hx_p)} \qquad\qquad \boldsymbol{P = P_p - KHP_p}$$

If P_p is large, it means that the state is estimated to change more frequently, so you need to be able to change your estimates with new measurements. As a result, the Kalman Gain is higher.

Conversely, if P_p is small, then you know that your state does not vary that much, so you don't want to alter your estimates too much at every time instant. Therefore, if $P_p \rightarrow 0$, then $K \rightarrow 0$.

The implementation of Kalman filter is shown in the picture below.

```
def filter(z, updateNumber):
    dt = 0.1
    # Initialize State
    if updateNumber == 1:
        filter.x = np.array([[0],
                             [20]])
        filter.P = np.array([[5, 0],
                             [0, 5]])

        filter.A = np.array([[1, dt],
                             [0, 1]])
        filter.H = np.array([[1, 0]])
        filter.R = 10
        filter.Q = np.array([[1, 0],
                             [0, 3]])

    # Predict State Forward
    x_p = filter.A.dot(filter.x)
    # Predict Covariance Forward
    P_p = filter.A.dot(filter.P).dot(filter.A.transpose()) + filter.Q
    # Compute Kalman Gain
    S = filter.H.dot(P_p).dot(filter.H.transpose()) + filter.R
    K = P_p.dot(filter.H.transpose())*(1/S)

    # Estimate State
    residual = z - filter.H.dot(x_p)
    filter.x = x_p + K*residual

    # Estimate Covariance
    filter.P = P_p - K.dot(filter.H).dot(P_p)

    return [filter.x[0], filter.x[1], filter.P];
```

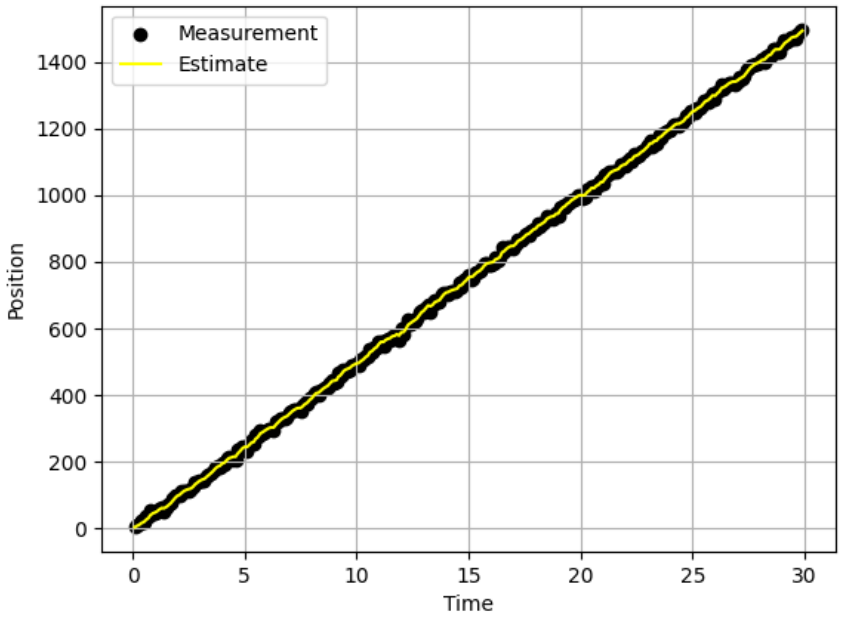
Test Kalman Filter

Now, it is time to test how the Kalman Filter works using the input measurements.

```
def testFilter():
    dt = 0.1
    t = np.linspace(0, 10, num=300)
    Number_of_measurements = len(t)

    Time_measurement = []
    Position_measurement = []
    Measurement_Diff_Position = []
    Estimate_Diff_position = []
    Estimate_position = []
    Estimate_velocity = []

    for k in range(1,Number_of_measurements):
        z = input(k)
        # Call Filter and return new State
        f = filter(z[0], k)
        # Save off that state so that it could be plotted
        Time_measurement.append(k)
        Position_measurement.append(z[0])
        Measurement_Diff_Position.append(z[0]-z[1])
        Estimate_Diff_position.append(f[0]-z[1])
        Estimate_position.append(f[0])
        Estimate_velocity.append(f[1])

    return [Time_measurement, Position_measurement, Estimate_position, Estimate_velocity, Measurement_Diff_Position, Estimate_Diff_position];
```

Plot Kalman Filter Results

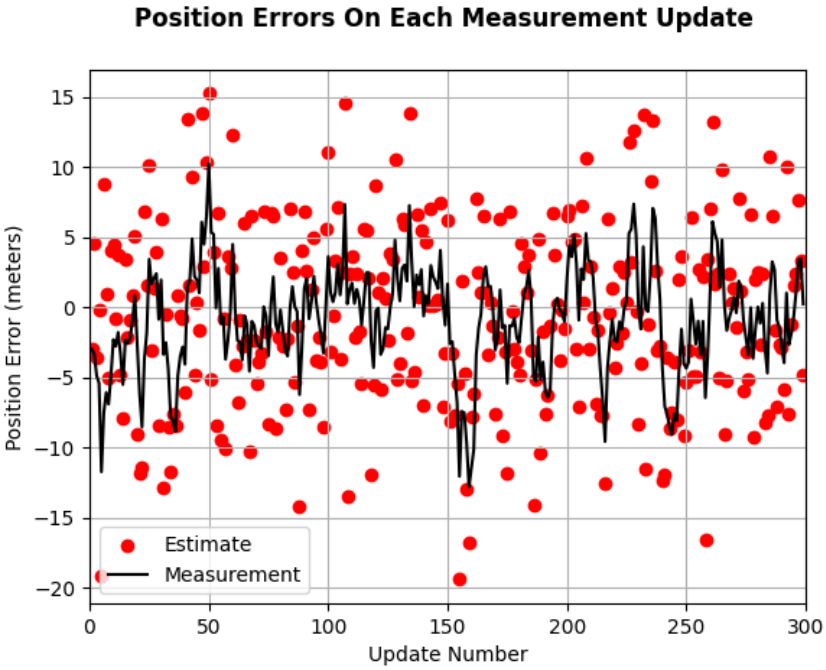
The first figure displays the measurement position and estimate position with respect to time.



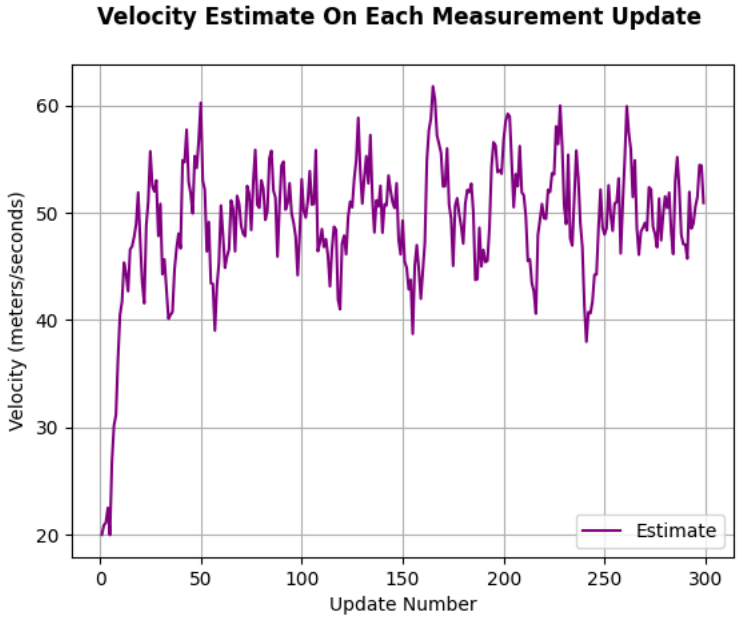
RMSE stands for Root Mean Square Error which is a measure of how spread out these residuals are. It shows how far estimates fall from measured actual values using Euclidean distance. Using RMSE, we can clearly judge the efficiency of the model and evaluate the quality of predictions.

```
Root Mean Square Error:
5.179791425279855
```

The second figure displays the position measurement error and estimate error relative to the actual car position. It illustrates how the Kalman Filter smooths the input measurements and reduces the positional error.



The third plot shows the velocity estimate after each measurement update. It can be observed that the Filter starts to maintain constant velocity at 50 m/s.



The code below is written to plot Kalman filter results.

```
t = testFilter()
dt = 0.1

plot1 = plt.figure(1)
plt.scatter([element * dt for element in t[0]], t[1], color = 'black')
plt.plot([element * dt for element in t[0]], t[2], color = 'yellow')
plt.ylabel('Position')
plt.xlabel('Time')
plt.grid(True)

plot2 = plt.figure(2)
plt.plot(t[0], t[3], color = 'purple')
plt.ylabel('Velocity (meters/seconds)')
plt.xlabel('Update Number')
plt.title('Velocity Estimate On Each Measurement Update \n', fontweight="bold")
plt.legend(['Estimate'])
plt.grid(True)

plot3 = plt.figure(3)
plt.scatter(t[0], t[4], color = 'red')
plt.plot(t[0], t[5], color = 'black')
plt.legend(['Estimate', 'Measurement'])
plt.title('Position Errors On Each Measurement Update \n', fontweight="bold")
plt.ylabel('Position Error (meters)')
plt.xlabel('Update Number')
plt.grid(True)
plt.xlim([0, 300])
plt.show()
```

The code below is written to calculate RMSE of the model.

```
MSE = np.square(np.subtract(t[1],t[2])).mean()
RMSE = math.sqrt(MSE)
print("\nRoot Mean Square Error: ")
print(RMSE)
print("\n")
```

References

Duckworth, D. (2012). *pykalman*. Retrieved from pykalman: https://pykalman.github.io/

Franklin, W. (2020). *Kalman Filter Explained Simply.* Retrieved from https://thekalmanfilter.com/kalman-filter-explained-simply/

Labbe, R. R. (2016). *KalmanFilter*. Retrieved from KalmanFilter: https://filterpy.readthedocs.io/en/latest/kalman/KalmanFilter.html