ARK Task Round Documentation

Advance Tasks

4.1 Tracking Multiple Objects

I. Introduction

In this task you had to implement Kalman filter. You had to find the trajectory of the drone. There are 6 ground stations and they have provided with the coordinates of the drone with respect to them but the coordinates are noisy.

I had read about Kalman filter but did not have the time to properly implement it. I have just done the very basics of Kalman filter.

II. Problem Statement

For a drone, it is very important to know the coordinates of the drone and how fast it is moving at all times with high precision. So, we get a state estimate with the help of the kinematics equations and then we compare our calculations with the measured data. But in practice the calculated results do not always match the actual ground measurements.

To overcome this problem, Kalman filter is used. It is an algorithm that uses a series of time-variant measurements containing various uncertainties and produces estimates of the unknown variables. It uses a joint probability distribution over the variables for each timeframe. Each step is considered to be operating on a linear system. The filter algorithm uses a recursive approach.

In this task we are supposed to implement the Kalman filter to predict the trajectory of the drone based on the data given to us by the 6 ground stations.

III. Related Works

I tried to implement Kalman filter using OpenCV and NumPy.

I was not able to make a working code though.

Algorithm

The Kalman filter algorithm can be divided into four parts:

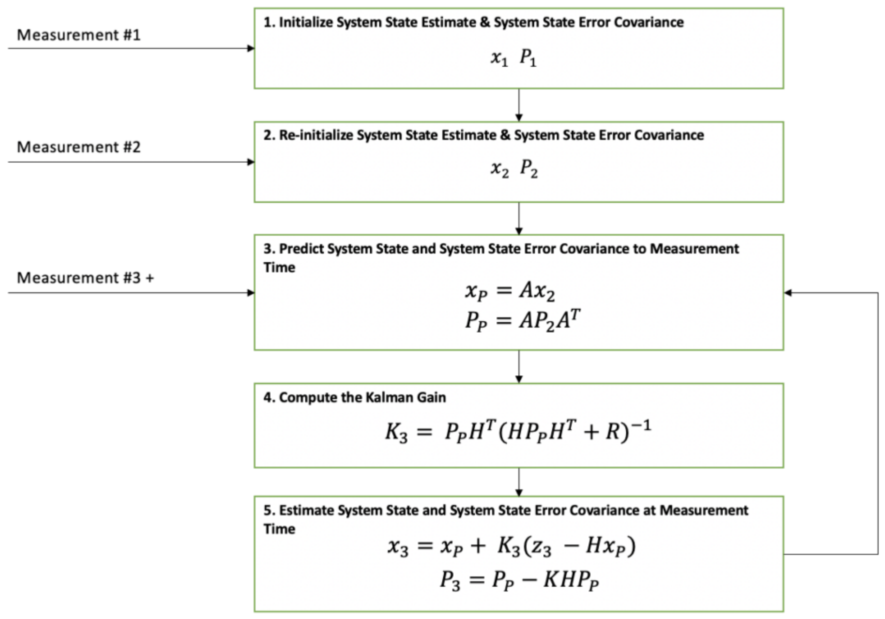
1. The initial state is recorded at the time when the sensors begin to give the readings.

2. Prediction- Based on previous knowledge of a vehicle position and kinematic equations, the position of vehicle is predicted.

3. Measurement- Ground measurements of the position and the velocity of vehicle is obtained and further compared with the predicted state obtained in step1.

4. Update - Finally the knowledge about the position (or state) of vehicle is updated based on the prediction and sensor readings.

Again, repeat through the step 2.


Measurement #1
Measurement #2
Measurement #3 +
1. Initialize System State Estimate & System State Error Covariance
x_1 P_1
2. Re-initialize System State Estimate & System State Error Covariance
x_2 P_2
3. Predict System State and System State Error Covariance to Measurement Time
$x_P = Ax_2$
$P_P = AP_2A^T$
4. Compute the Kalman Gain
$K_3 = P_P H^T (HP_P H^T + R)^{-1}$
5. Estimate System State and System State Error Covariance at Measurement Time
$x_3 = x_P + K_3(z_3 - Hx_P)$
$P_3 = P_P - KHP_P$

IV. Initial Attempts

The first thing that I had tried to do was have a very small input data...that is like 3 to 4 samples of the coordinates and the velocities and then apply Kalman filter on it.

```
import numpy as np
from numpy.linalg import inv

x_observations = np.array([320, 330, 340])
v_observations = np.array([50, 100, 150])

z = np.c_[x_observations, v_observations]

# Initial Conditions
a = 1  # Acceleration
v = 80
```

```python
t = 1  # Difference in time


# Process / Estimation Errors
error_est_x = 10
error_est_v = 4


# Observation Errors
error_obs_x = 25  # Uncertainty in the measurement
error_obs_v = 8


def prediction2d(x, v, t, a):
    A = np.array([[1, t],
                  [0, 1]])
    X = np.array([[x],
                  [v]])
    B = np.array([[0.5 * t ** 2],
                  [t]])
    X_prime = A.dot(X) + B.dot(a)
    return X_prime



def covariance(s1, s2):
    cov1 = s1 * s2
    cov2 = s2 * s1
    cov_matrix = np.array([[s1 ** 2, cov1],
                           [cov2, s2 ** 2]])
    return np.diag(np.diag(cov_matrix))
```

```
# Initial Estimation Covariance Matrix
P = covariance(error_est_x, error_est_v)
A = np.array([[1, t],
              [0, 1]])


# Initial State Matrix
X = np.array([[z[0][0]],
              [v]])
n = len(z[0])


for data in z[1:]:
    X = prediction2d(X[0][0], X[1][0], t, a)
    # To simplify the problem, professor
    # set off-diagonal terms to 0.
    P = np.diag(np.diag(A.dot(P).dot(A.T)))


    # Calculating the Kalman Gain
    H = np.identity(n)
    R = covariance(error_obs_x, error_obs_v)
    S = H.dot(P).dot(H.T) + R
    K = P.dot(H).dot(inv(S))


    # Reshape the new data into the measurement space.
    Y = H.dot(data).reshape(n, -1)


    # Update the State Matrix
    # Combination of the predicted state, measured values, covariance matrix
and Kalman Gain
```

```
    X = X + K.dot(Y - H.dot(X))


    # Update Process Covariance Matrix
    P = (np.identity(len(K)) - K.dot(H)).dot(P)


print("Kalman Filter Matrix is:\n", X)
```

V. Final Attempt

In the final attempt I created snippets of code but was not able to work it ot completely.

This is to create matrices and perform the filters operations..

```
def create_A(m):
calc_m=m
calc_m[0,0]=calc_m[0,0]+m[1,0]*t
calc_m[1,0]=calc_m[1,0]+m[1,1]*t
return calc_m

def calc_perror(p_measured,p_prev):
 return abs(p_measured-p_prev)
def calc_perror2(p_measured,p_prev):
 p_max_error=abs(p_measured-p_prev)
 return 1/(2*(np.pi)*(p_max_error**2))
```

R is the co-varaince matrix of measurement

Variance is calculated about the predicted path point

```
def create_R(prev,curr):
    var_px=calc_perror2(curr[0][0],prev[0][0])
    var_py=calc_perror2(curr[0][1],prev[0][1])
    var_vx=calc_verror2(curr[1][0],prev[1][0])
    var_vy=calc_verror2(curr[1][1],prev[1][1])
    R=np.diag([var_px,var_py,var_vx,var_vy])
#R=np.diag([0.8,0.8,0.1,0.1])
    return R
```

Calculate Kalman Gain

```
def calc_K(P,R):
    return (np.dot(P,np.linalg.inv(np.add(P,R))))
```

This is for the update step

```
def update_X(prev,curr,P,R): # update X
    t=np.subtract(curr,prev)
    K=calc_K(P,R)
    t[0][0]=K[0][0]*t[0][0]
    t[0][1]=K[1][1]*t[0][1]
    t[1][0]=K[2][2]*t[1][0]
    t[1][1]=K[3][3]*t[1][1]
    t=np.add(t,curr)
    return t
def update_P(P,R): # update P
    K=calc_K(P,R)
    I=np.identity(4,dtype=float)

    return np.dot(np.subtract(I,K),P)
```

VII. Future Work

Another filter like particle filter, histogram filter could be try to be implemented.

VIII. Conclusion

The overall problem was about to get an accurate trajectory of the drone using the Kalman filter. This would be helpful to find the position, velocity and acceleration the drone at any point of time.

Kalman filters are extensively used in the autonomous world. A common application is in navigation and control of vehicles, particularly aircraft, spacecraft and dynamically controlled ships. The Kalman filter stands out of other techniques because of its dynamic structure. It considers errors, noise, external influence unlike other techniques.

References

https://docs.ufpr.br/~danielsantos/ProbabilisticRobotics.pdf

https://thekalmanfilter.com/kalman-filter-explained-simply/

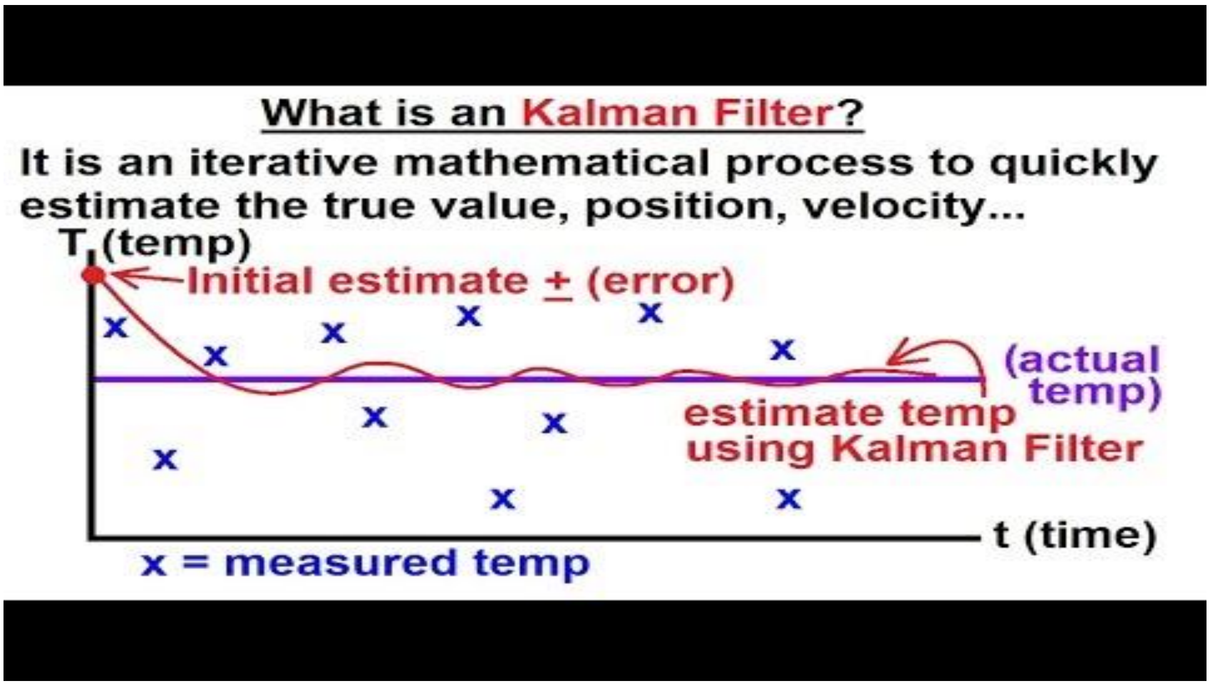
http://web.mit.edu/kirtley/kirtley/binlustuff/literature/control/Kalman%20filter.pdf

https://towardsdatascience.com/sensor-fusion-part-1-kalman-filter-basics-4692a653a74c

https://towardsdatascience.com/sensor-fusion-part-2-kalman-filter-code-78b82c63dcd

https://www.intechopen.com/books/introduction-and-implementations-of-the-kalman-filter/introduction-to-kalman-filter-and-its-applications

Special Topics - The Kalman Filter (1 of 55) What is a Kalman Filter?



What is an Kalman Filter?

It is an iterative mathematical process to quickly estimate the true value, position, velocity...