In Part 1, we left after deriving basic equations for a Kalman filter algorithm. Here they are stated again for easy reference.

1. Predict:
   1. X = A \* X + B \* u
   2. P = A \* P \* AT \* Q
2. Measurement
   1. Y = Z – H \* X
   2. K = ( P \* HT ) / ( ( H \* P \* HT ) + R )
3. Update
   1. X = X + K \* Y
   2. P = ( I – K \* H ) \* P

As our sensor readings, we will use a text file which is freely available at Udacity’s github handle. This text file (*obj\_pose-laser-radar-synthetic-input.txt*) has sensor readings from Laser and Radar along with reading’s timestamp and ground truth. Till this point, we have only covered basic Kalman filter algorithm, hence in this coding exercise we will only be using Laser readings from the above stated input file. Once we cover ‘Extended Kalman Filter’ in future post, we will start using Radar readings too. But with our current understanding of Kalman Filter equations, just using Laser readings will serve as a perfect example to cement our concept with help of coding.

Sensor readings captured in input text file are in below format

For a row containing radar data, the columns are: sensor\_type (R), rho\_measured, phi\_measured, rhodot\_measured, timestamp, x\_groundtruth, y\_groundtruth, vx\_groundtruth, vy\_groundtruth, yaw\_groundtruth, yawrate\_groundtruth.

For a row containing lidar data, the columns are: sensor\_type (L), x\_measured, y\_measured, timestamp, x\_groundtruth, y\_groundtruth, vx\_groundtruth, vy\_groundtruth, yaw\_groundtruth, yawrate\_groundtruth.

With all this information at our fingertip, let’s begin coding. As with any Python file, let’s import all required libraries first

import numpy as np

import pandas as pd

from numpy.linalg import inv

Next, read input text file.

measurements = pd.read\_csv('obj\_pose-laser-radar-synthetic-input.txt', header=None, delim\_whitespace = True, skiprows=1)

I have made ‘skiprows’ parameter as 1 because, when we begin implementing KF algorithm, we don’t have any prior knowledge about state of vehicle. In such cases usually we default our state to first reading from sensor output. And this is exactly what we will be doing. In below shown code, we will initialize our state ***X*** with reading from first row of input file. And as this reading has already been used, we simply skip it while reading input file. Along with initial values of state vector ***X***, we will also use first *timestamp* from input file as our ***previous time****.* As explained in previous post, we need current and previous timestamps to calculate ***delta\_t****.* Timestamps provided are in unit microseconds, which we will divide by 10^6. This has two reasons, first, a smaller number is easier to maintain. Second, the velocity groundtruth readings (and hence velocity values in our code) are in units of *seconds.*

prv\_time = 1477010443000000/1000000.0

x = np.array([

[0. 312242],

[0. 5803398],

[0],

[0]

])

Next, we Initialize variables to store ground truth and RMSE values. RMSE (Root Mean Square Error) is used to judge performance of our algorithm against ground truth values.

ground\_truth = np.zeros([4, 1])

rmse = np.zeros([4, 1])

We initialize matrix ***P*** and ***A***. For details about structure of ***P*** and ***A*** matrices, refer to Part 1 where its explained in more depth. Basically matrix ***A*** is used to implement kinematics equations of distance, speed and time, and matrix ***P*** is State Covariance Matrix having variances in *x, y, vx* and *vy* as its diagonal elements. What this initial ***P*** values mean is that we have high confidence in our positional values (which makes sense as we have taken it from actual sensor readings), indicated by relatively low variance value, and low confidence in velocity values (which again makes sense as we have no idea about velocity), indicated by relatively large variance value.

P = np.array([

[1, 0, 0, 0],

[0, 1, 0, 0],

[0, 0, 1000, 0],

[0, 0, 0, 1000]

])

A = np.array([

[1.0, 0, 1.0, 0],

[0, 1.0, 0, 1.0],

[0, 0, 1.0, 0],

[0, 0, 0, 1.0]

])

Next we define ***H*** and ***I*** matrices, which, as I explained in last post, will be 4 x 2 and 4 x 4 matrices respectively. We define vector ***Z***, which, as our Lidar readings will consist of 2 positional readings (x and y), will be a 2 x 1 vector.

H = np.array([

[1.0, 0, 0, 0],

[0, 1.0, 0, 0]

])

I = np.identity(4)

We define Measurement Covariance matrix ***R***, which again, as per last post will be a 2 x 2 matrix. We will talk more about how to get values for ***R*** matrix and ***noise\_ax*** and ***noise\_ay*** in future articles.

R = np.array([

[0.0225, 0],

[0, 0.0225]

])

Next we define ***noise\_ax***, ***noise\_ay*** and matrix ***Q***.

noise\_ax = 5

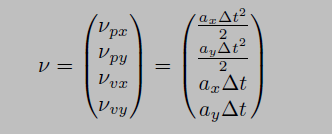
noise\_ay = 5

Q = np.zeros([4, 4])

Let’s take few moments to understand them. If we revisit kinematics equations defined in 1st part of this series, you can see that there is an acceleration factor in positional and velocity terms. They have been rewritten here for ease.

1. Px(t+1) = Px + delta\_t \* vx + 0.5 \* ax \* delta\_t 2
2. Py(t+1) = Py + delta\_t \* vy + 0.5 \* ay \* delta\_t 2
3. Vx(t+1) = Vx + ax \* delta\_t
4. Vy(t+1) = Vy + ay \* delta\_t

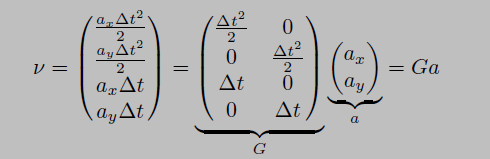
Since the acceleration is unknown we can add it to the noise component, and this random noise would be expressed analytically as the last terms in the equation derived above. So, we have a random acceleration vector *v* in this form:



which is described by a zero mean and a covariance matrix ***Q***.



The vector ***v*** can be decomposed into two components a 4 by 2 matrix G which does not contain random variables and a 2 by 1 matrix *a* which contains the random acceleration components:



delta\_t is calculated at each iteration of Kalman Filter, and as we don’t have any acceleration data, we define acceleration ***a***as random vector with zero mean and standard deviations noise\_ax and noise\_ay.

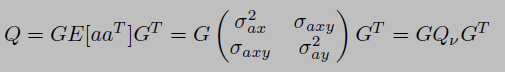
Based on our noise vector we can define now the new covariance matrix Q. The covariance

matrix is defined as the expectation value of the noise vector *v* times the noise vector *v* transpose.

So let’s write this down:

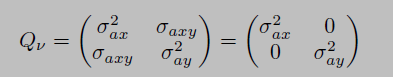


To know more about ‘Expected Value’, watch this video from Khan Academy. As G does not contain random variables, we can put it outside the expectation calculation.

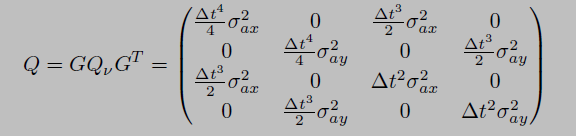


ax and ay are assumed to be uncorrelated noise processes. This means that the covariance sigma\_axy in

Q\_ is zero:



So after combining everything in one matrix we obtain our 4 by 4 Q matrix:



At each iteration of Kalman Filter, we will be calculating matrix Q as per above formula.

With all our variables defined, let’s begin with iterating through sensor data and applying Kalman Filter on them. Running a for loop till length of measurements, reading measurement line, checking if it’s a Lidar (‘L’) reading.

for i in range (len(measurements)):

new\_measurement = measurements.iloc[i, :].values

if new\_measurement[0] == 'L':

Get timestamp from current reading, calculate change in time by comparing it with previous timestamp and then replace current timestamp as previous timestamp for next iteration.

cur\_time = new\_measurement[3]/1000000.0

dt = cur\_time - prv\_time

prv\_time = cur\_time

Calculate delta\_t’s (‘dt’ in code) square, cube , 4th power of delta\_t which are required to calculate Q matrix.

dt\_2 = dt \* dt

dt\_3 = dt\_2 \* dt

dt\_4 = dt\_3 \* dt

Updating matrix A with delta\_t value. Delta\_t will be multiplied by velocity to come up with positional values.

A[0][2] = dt

A[1][3] = dt

Updating Q matrix. If you look back at above derived equations for Q matrix, you can easil corelate below provided lines of code with that.

Q[0][0] = dt\_4/4\*noise\_ax

Q[0][2] = dt\_3/2\*noise\_ax

Q[1][1] = dt\_4/4\*noise\_ay

Q[1][3] = dt\_3/2\*noise\_ay

Q[2][0] = dt\_3/2\*noise\_ax

Q[2][2] = dt\_2\*noise\_ax

Q[3][1] = dt\_3/2\*noise\_ay

Q[3][3] = dt\_2\*noise\_ay

Updating sensor readings.

z\_lidar[0][0] = new\_measurement[1]

z\_lidar[1][0] = new\_measurement[2]

Collecting ground truths

ground\_truth[0] = new\_measurement[4]

ground\_truth[1] = new\_measurement[5]

ground\_truth[2] = new\_measurement[6]

ground\_truth[3] = new\_measurement[7]

And finally call Predict and Update functions.

predict()

update(z\_lidar)

Now lets have a look at our predict() function which would very much similar to the following predict equations that we been using in this series. Not much to explain in the code section, its really just a direct replica of derived formulas.

1. Predict:
   1. X = A \* X + B \* u
   2. P = A \* P \* AT \* Q

def predict():

# Predict Step

global x, P, Q

x = np.matmul(A, x)

At = np.transpose(A)

P = np.add(np.matmul(A, np.matmul(P, At)), Q)

Moving ahead to define update function(). We will be implementing both ‘measurement’ and ‘update’ steps in this function

1. Measurement
   1. Y = Z – H \* X
   2. K = ( P \* HT ) / ( ( H \* P \* HT ) + R )
2. Update
   1. X = X + K \* Y
   2. P = ( I – K \* H ) \* P

def update(z):

global x, P

# Measurement update step

Y = np.subtract(z\_lidar, np.matmul(H, x))

Ht = np.transpose(H)

S = np.add(np.matmul(H, np.matmul(P, Ht)), R)

K = np.matmul(P, Ht)

Si = inv(S)

K = np.matmul(K, Si)

# New state

x = np.add(x, np.matmul(K, Y))

P = np.matmul(np.subtract(I ,np.matmul(K, H)), P)

…and with that, you have gone through complete coding for a Kalman Filter algorithm. Even though it might look like a small step, this is the foundational algorithm for many of the advanced versions used for Sensor fusion technology. As stated earlier, all variants of Kalman Filter consists of same Predict, Measurement and Update states that we have defined in this series so far. The only difference in more advanced versions is the different kinematics and sensor equations they use. We will go through them too step by step in this article. But at this moment, lets have a high five for finishing our foundation step of a classic Kalman Filter Algorithm. You can find complete code along with input file at my github repo here.

As usual, if you liked my article, show your appreciation with likes and comments. You can also find me and my other articles at twitter

Till next time, cheers!!