Kalman Filter

Skylar Koroluk

May 27, 2018

Abstract

This document serves as both general documentation of the KalmanFilterProcess2.py program implemented within rover-processes for the University of Saskatchewan Space Design Team and as a general explanation of how a basic Kalman Filter works.

Contents

1 What is a Kalman Filter

According to Wikipedia:

> Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe.

An english translation by me:

> A user who is implementing GPS to track something such as a vehicle or aircraft knows the limitations of the movements of that craft and therefore can ignore any erratic motion and can also help predict the next GPS location based on the previous reading and other measurements of the motion of the craft such as accelerometer readings etc. For example, if you are tracking a car driving 100km/h north and all of a sudden your GPS reading is 5m south of the previous location, you as the user knows that is impossible. For our implementation with the rover we take GPS readings and accelerometer data and "slam it together" to get an accurate position of the rover.

2 Variables Used

Note: Short single letter variable names were chosen to match popular convention. This also makes it easier to follow the mathematics that the code is heavily reliant on.

2.1 Main Variables

Q Process noise covariance matrix: A matrix containing the variances and covariances of the noise in the different parameters of the state model.

R Observation noise covariance matrix: A matrix containing the variances and covariances of the noise in the different parameters of the observation measurements.[1]

v Observation noise model: The noise in the observation.[2]

F State transition model: Applied to the previous state in order to calculate the current state. This should never be changed in KalmanFIlterProcess2.py.

B Control input model: Applied to the control inputs vector **u** to properly use the sensor data in the model. This should never be changed in KalmanFIlterProcess2.py.

H Observation model: Maps the state vector to the matrix state space dealt with in the calculations. This should never be changed in KalmanFIlterProcess2.py.

P_pp Previous (predicted) process error covariance matrix: The previous "true" estimated accuracy of the state.

x_pp Previous (predicted) state vector: The previous "true" state.

u Control inputs vector: The control inputs describing the change of the state.[3]

P_cp Current (predicted) process error covariance matrix: Updated process error covariance matrix based on the state transition model, control input model, and control inputs (**F**, **B**, **u**).

x_cp Current (predicted) state vector: Updates state based on the state transition model, control input model, and control inputs (**F**, **B**, **u**).

[1] In KalmanFilterProcess2.py this is the variances and covariances of the noise in the GPS measurements. The values for this matrix were calculated in RoverStatistics.py

[2] For KalmanFilterProcess2.py this was created with more of an iterative process rather than any mathematical process. This is one of the areas that I plan to look into more but for now it is fine.

[3] In KalmanFilterProcess2.py this is accelerometer readings.

z Observation Vector: Observational measurements of location, usually from GPS based measurements. [4]

K Kalman gain matrix: A value between 0 and 1 that gives a "weighting of trust" to either the the mathematical model or the observation, depending on the relative errors of both. A Kalman gain of zero means that only the mathematical model is used whereas a gain of 1 means that only the observation is used.

P_cc Current (calculated/updated) process error covariance matrix: Final "true" process covariance matrix.

x_cc Current (calculated/updated) state vector: Final "true" state.

2.2 Other Variables

zone_number/zone_letter In the UTM coordinate system the earth is split up into different zones that are distinguishable by number and letter. These values are needed when converting from UTM to GCS and are found during the conversion from GCS to UTM using the UTM package.

I Identity matrix: Common in linear algebra it is a matrix with 1's across the diagonal and zeros elsewhere.

S N/A: Intermediate value often used to simplify calculations.

L N/A: Intermediate value used to simplify calculations.

y_p N/A: Intermediate value used to simplify calculations.

3 The Mathematics

3.1 Setup

Firstly all the different static variables regarding the specific application of the filter must be created. Examples will be shown for filtering the location of a 2D restricted vehicle such as the USST mars rover.

F:

$$\begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

[4] In the KalmanFilterProcess2.py program they are Geographic Coordinate System (latitude/longitude) measurements converted to Universal Transverse Mercator coordinates (northing/easting), using the UTM package in python

B:

$$\begin{bmatrix} \frac{(\Delta t)^2}{2} & 0 \\ 0 & \frac{(\Delta t)^2}{2} \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}$$

H:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

3.2 Predict Stage

In the prediction stage of the calculation the filter takes the previous calculated state and predicts a current location according to the control inputs[5] and the state transition model.

$$\mathbf{x}_{cp} = \mathbf{F}\mathbf{x}_{pp} + \mathbf{B}\mathbf{u} \tag{1}$$

The same prediction/update is done for the process error covariance matrix.

$$\mathbf{P}_{cp} = \mathbf{F}\mathbf{P}_{pp}\mathbf{F}^T + \mathbf{Q} \tag{2}$$

3.3 Update Stage

Next in the update stage the filter uses the observed location and the predicted location and calculates a weighted average where the "weight" is the Kalman Gain. Firstly the Kalman Gain is calculated using the current predicted process error covariance matrix and the observation noise covariance matrix.

$$\mathbf{K} = \frac{\mathbf{P}_{cp}\mathbf{H}^T}{\mathbf{R} + \mathbf{H}\mathbf{P}_{cp}\mathbf{H}^T} \tag{3}$$

Using the Kalman Gain and the observed state the final calculated "true" state and process error covariance matrix can be calculated.

$$\mathbf{x}_{cc} = \mathbf{x}_{cp} + \mathbf{K}[\mathbf{z} - \mathbf{H}\mathbf{x}_{cp}] \tag{4}$$

$$\mathbf{P}_{cc} = [\mathbf{I} - \mathbf{K}\mathbf{H}]\mathbf{P}_{cp}[\mathbf{I} - \mathbf{K}\mathbf{H}]^T + \mathbf{K}\mathbf{R}\mathbf{K}^T \tag{5}$$

In KalmanFilterProcess2.py these last three calculations are simplified with intermediate values as follows

$$\mathbf{S} = \mathbf{R} + \mathbf{H}\mathbf{P}_{cp}\mathbf{H}^T \tag{6}$$

$$\mathbf{L} = \mathbf{I} - \mathbf{K}\mathbf{H} \tag{7}$$

$$\mathbf{y}_p = \mathbf{z} - \mathbf{H}\mathbf{x}_{cp} \tag{8}$$

This simplifies equations (3)-(5) to

$$\mathbf{K} = \mathbf{P}_{cp}\mathbf{H}^T\mathbf{S}^{-1} \tag{9}$$

$$\mathbf{x}_{cc} = \mathbf{x}_{cp} + \mathbf{K}\mathbf{y}_p \tag{10}$$

[5] ex. for the rover the control inputs consist of accelerometer data during the last time step.

$$\mathbf{P}_{cc} = \mathbf{L}\mathbf{P}_{cp}\mathbf{L}^T + \mathbf{K}\mathbf{R}\mathbf{K}^T \tag{11}$$

The state vector $\mathbf{x}_{cc}$ is the "true" state that is returned by the filter. In KalmanFilterProcess2.py this vector contains the x and y (East and North) positions and velocities. The positions are then passed through the UTM converter to convert the UTM positions back into GCS GPS coordinates

4 Full Code

```
import numpy as np
import utm as utm
import statistics as stats
from robocluster import Device
import random as random
import matplotlib.pyplot as plt
import time

KalmanFilter = Device('KalmanFilter', 'rover')

DummyGPS = Device('DummyGPS', 'rover')
# @DummyGPS.every('0.1s')
async def dummy():
    #await DummyGPS.publish('singlePointGPS',
        [51.00000+(random.randrange(400, 600)/1000000),
        110.00000+(random.randrange(600, 800)/1000000)])
    await DummyGPS.publish('singlePointGPS', [random.gauss(52.13255308,
        0.0000699173), random.gauss(-106.6279284, 0.0000460515)])


def create_covariance_of_process_noise(num_dimensions):
    # this is currently completely manual
    Q = np.eye(2 * num_dimensions)*0.005 # TODO: Figure out what this
        matrix actually is
    return Q

def create_covariance_of_observation_noise(num_dimensions):
    # this is currently completely manual
    # these values are taken from RoverGPSstatistics
    R = np.zeros((2 * num_dimensions, 2 * num_dimensions))
    R[0, 0] = 60.125
    R[1, 1] = 60.125#10.308
    R[0, 1] = 8.783
    R[1, 0] = 8.783
    return R

def create_observation_noise(num_dimensions):
    '''this is currently completely manual
    drawn from zero mean white noise gaussian with covariance, R:v ~
        N(0,R)'''
    v = np.zeros((2 * num_dimensions, 1)) # TODO: Figure out what this
        vector/matrix is
    v[0, 0] = 0.001
    v[1, 0] = 0.001
    return v
```

```
def create_state_transition_model(time_step, num_dimensions):
    '''takes a time step value and a number of dimensions and returns a
        state_transition model of the form
    pos1, pos2, ..., posn, vel1, vel2, ..., veln. This is opposed to a
        model of the form pos1, vel1, pos2, vel2...'''
    F = np.eye(2 * num_dimensions)
    for index in range(num_dimensions):
        F[index, index + 2] = time_step
    return F

def create_control_input_model(time_step, num_dimensions):
    '''takes a time step value and a number of dimensions and returns a
        control input model of the form
    pos1, pos2, ..., posn, vel1, vel2, ..., veln. This is opposed to a
        model of the form pos1, vel1, pos2, vel2...'''
    B = np.zeros((2 * num_dimensions, num_dimensions))
    for index in range(num_dimensions):
        B[index, index] = 0.5 * (time_step ** 2)
        B[index + 2, index] = time_step
    return B

def create_observation_model(num_dimensions):
    '''takes the number of dimensions and creates the observation matrix
        used to map the true state to the observed
    space. For our uses this is simply an identity matrix.'''
    H = np.eye(2 * num_dimensions)
    for index in range(num_dimensions):
        H[index + 2, index + 2] = 0
    return H

def create_initial_process_error_covariance_matrix(num_dimensions,
    variance=0):
    '''takes in the number of dimensions and the variable variance
        values and creates an initial process error
    covariance matrix. If initial positions are known to a decent level
        of accuracy this is just a zero matrix
    hence the default variance of 0.
    If initial positions are not known then variance should be a list of
        variances of the form
    pos1, pos2, ..., posn, vel1, vel2, ..., veln. This is opposed to a
        model of the form pos1, vel1, pos2, vel2...'''

    P_pp = np.zeros((2 * num_dimensions, 2 * num_dimensions))
    if variance != 0:
        for index in range(2 * num_dimensions):
            P_pp[index, index] = variance[index]
    return P_pp

def create_initial_state_vector(num_dimensions, initial_conditions = (0,
    0)):
    '''takes in the number of dimensions and the initial conditions and
        creates an initial state vector.
    If initial positions are chosen to be entered then they should be in
        a list of the form
    pos1, pos2, ..., posn, vel1, vel2, ..., veln. This is opposed to a
        model of the form pos1, vel1, pos2, vel2...
```

```
    Initial positions should be entered but are optional. Ideally if no
        initial conditions are not entered then
    the variance in the initial process error covariance matrix should
        have npne default variance parameters.
    Don't believe this is actually too important though'''
    x_pp = np.zeros((2 * num_dimensions, 1))
    if initial_conditions != (0, 0):
        for index in range(num_dimensions):
            x_pp[index, 0] = initial_conditions[index]
    return x_pp

def create_control_inputs_vector(ax = 0, ay = 0):
    '''takes accelerometer data and turns it into acceleration in
        northing easting and then creates a vector'''
    u = np.array([[ax],[ay]])
    #u = np.zeros((2, 1)) # TODO: actually calculate acceleration in
        northing easting and create vector
    return u

def predict(F, x_pp, B, u, P_pp, Q):
    '''takes the previous (a posteriori) state estimate and previous (a
        posteriori) process error covariance matrix
    and creates the predicted (a priori) state estimate and the
        predicted (a priori) estimate covariance'''
    x_cp = np.dot(F, x_pp) + np.dot(B, u)
    P_cp = np.dot(F, np.dot(P_pp, F.T)) + Q
    return x_cp, P_cp

def observation(gps_latitude, gps_longitude, H, v):
    '''currently only for 2d system
    takes GPS measurements and converts them to northing and easting
        values.'''
    [x_meas, y_meas, zone_number, zone_letter] =
        utm.from_latlon(gps_latitude, gps_longitude)
    x_m = np.mat([[x_meas], [y_meas], [0], [0]])
    #cart_pos = [x_meas, y_meas]
    zone_info = [zone_number, zone_letter]
    z = np.dot(H, x_m) + v
    return z, zone_info

def update(num_dimensions, x_cp, P_cp, z, H, R):
    '''takes in the (a priori) estimates along with the observations and
        appropriate matrices to calculate the true
    state.'''
    I = np.eye(len(x_cp)) # Identity matrix needed for calculations
    y_p = z - np.dot(H, x_cp) # measurement pre-fit residual
    S = R + np.dot(H, np.dot(P_cp, H.T)) # pre_fit residual covariance
    #S[S == 0] = 1
    K = np.dot(P_cp, H.T) / S # Optimal Kalman gain
    K[np.isnan(K)] = 0
    K[np.isinf(K)] = 0

    x_cc = x_cp + np.dot(K, y_p) # updated (a posteriori) state estimate
    L = (I - np.dot(K, H)) # intermediate calculation needed for later
    P_cc = np.dot(L, np.dot(P_cp, L.T)) + np.dot(K, np.dot(R, K.T)) #
        updated (a posteriori) estimate covariance
    P_cc[np.isnan(P_cc)] = 0
```

```
    y_c = z - np.dot(H, x_cc) # measurement post-fit residual, we will
        probably never use this
    return x_cc, P_cc


@KalmanFilter.task
def start_up_kalman_filter():
    #User defined variables
    #-----------------------------------------------------------------------------------------------------------
    dt = 0.1 # time step (can be determined by rate that GPS readings
        come in)
    num_dimensions = 2 # number of dimensions in model
    ax = 0
    ay = 0
    initial = [52.132653, -106.628012]
    initial_cart = utm.from_latlon(initial[0], initial[1])
    KalmanFilter.storage.initial_cart = initial_cart
    KalmanFilter.storage.dt = dt
    KalmanFilter.storage.F = create_state_transition_model(dt,
        num_dimensions)
    KalmanFilter.storage.B = create_control_input_model(dt,
        num_dimensions)
    KalmanFilter.storage.H = create_observation_model(num_dimensions)
    KalmanFilter.storage.x_pp =
        create_initial_state_vector(num_dimensions, initial)
    KalmanFilter.storage.P_pp =
        create_initial_process_error_covariance_matrix(num_dimensions)
    KalmanFilter.storage.Q =
        create_covariance_of_process_noise(num_dimensions)
    KalmanFilter.storage.R =
        create_covariance_of_observation_noise(num_dimensions)
    KalmanFilter.storage.v = create_observation_noise(num_dimensions)
    KalmanFilter.storage.u = create_control_inputs_vector(ax, ay)

@KalmanFilter.on('*/Acceleration')
async def update_accel(event, data):
    print(data)
    KalmanFilter.storage.u = data


@KalmanFilter.on('*/singlePointGPS')
async def kalman_filter(event, data):
    num_dimensions = 2
    F = KalmanFilter.storage.F
    B = KalmanFilter.storage.B
    H = KalmanFilter.storage.H
    x_pp = KalmanFilter.storage.x_pp
    P_pp = KalmanFilter.storage.P_pp
    Q = KalmanFilter.storage.Q
    R = KalmanFilter.storage.R
    v = KalmanFilter.storage.v
    u = KalmanFilter.storage.u
    initial_cart = KalmanFilter.storage.initial_cart

    #u = create_control_inputs_vector(ax, ay) #is where the
        accelerometer values go
    gps_latitude = data[0] # Is this causing issues?
```

```
    gps_longitude = data[1]
    [x_cp, P_cp] = predict(F, x_pp, B, u, P_pp, Q)
    [z, zone_info] = observation(gps_latitude, gps_longitude, H, v) #
        This is where gps values are inputted
    [x_cc, P_cc] = update(num_dimensions, x_cp, P_cp, z, H, R)
    true_gps = utm.to_latlon(x_cc[0, 0], x_cc[1, 0], zone_info[0],
        zone_info[1], strict=False) # this has the value we
    cart_pos_final = [x_cc[0,0], x_cc[1,0]]
    print("z", z)
    print('raw: {}'.format(data))
    print("                                        filtered:
        {}".format(true_gps))

    ''' pres = 0.00001
    dif_lat = abs(true_gps[0] - 52.132653)
    dif_long = abs(true_gps[1] + 106.628012)
    if dif_lat < pres:
        print('Accurate Lat : ', dif_lat)
    if dif_long < pres:
        print('Accurate Long : ', dif_long)'''



    await KalmanFilter.publish('FilteredGPS', true_gps)
    KalmanFilter.storage.x_pp = x_cc
    KalmanFilter.storage.P_pp = P_cc


try:
    KalmanFilter.start()
    KalmanFilter.wait()
    for i in range(0,350):
        print("{},{}".format(KalmanFilter.storage.filtered_x[i],
            KalmanFilter.storage.filtered_y[i]))
    print("New Data\n\n")
    for i in range(0,350):
        print("{},{}".format(KalmanFilter.storage.noise_x[i],
            KalmanFilter.storage.noise_y[i]))

except KeyboardInterrupt:
    KalmanFilter.stop()
```