Kalman Filter Made Easy

STILL WORKING ON THIS DOCUMENT ...

Kalman Filter - Da Theory

You may happen to come across a fancy technical term called Kalman Filter, but because of all those complicated math, you may be too scared to get into it. This article provides a not-too-math-intensive tutorial for you and also me because I do forget stuff from time to time. Hopefully you will gain a better understanding on using Kalman filter.

If you ever design an embedded system, you will very likely to come across with some noisy sensors. To deal with these shity sensors, Kalman filter comes to rescue. Kalman filter has the the ability to fuse multiple sensor readings together, taking advantages of their individual strength, while gives readings with a balance of noise cancelation and adaptability. How wonderful!

Let's suppose you just meet a new girl and you have no idea how punctual she will be. (Girls are, in fact, not too punctual based on my personal experencie.) Based on her history, you has an estimation to when she will arrive. You don't want to come early, but at the same time, you don't want to be yelled for being late, so you want to come at exactly the right time. Now the girl comes and she is 30 mintues late (and that's how she interpret the meaning of "to the hour"), you now correct your own estimation for this current time frame, and use this new estimation to predict when she will come next time you meet her. A filter is exactly that; they use the same principle.

Now say you are back to work. You need to know the a system state based on some noisy, perhaps a few redundant, sensors. Now read this carefully: given a sensor readings, the system is going to correct its own predicted states made by last step, and use these corrected estimations to make new predictions for the next time step. Doesn't sound too complicated, huh?

We can then write the above concept mathematically

```
corrected_state = prediction + kalman_gain ( sensoroutput - state2output(prediction) );
```

where corrected_state is the new corrected state estimation; prediction is the estimation state made from last time frame; kalman_gain is the gain, which we will worry about later. For now, it is just some constant; sensoroutput is the measurement from an actual sensor, aka voltage; state2output is a function to convert a state to a measurement; state2output(state) is then the measurement prediction based on the state; and finally sensoroutput - state2output(prediction) is the measurement prediction error also known as the Innovation vector.

If we need to control some robot arm or whatever. corrected_state is what you want. Next, I will then use the sensor input for next prediction. The estimation will be

```
prediction = predict(corrected_state, sensoroutput);
```

where prediction is the prediction for next time frame, predict is a function that output a prediction based on corrected_state and sensoroutput. Yup, I did use same sensoroutput two times for each time frame. I use it to correct the prediction made from last time frame, and use it again for prediction of the next time frame

Kalman Filter - General Strategy

The general strategy to have a good filter is to sample as fast as possible. That's because sensors are noisy. And by averaging a lot of data, we are able to reduce the noise and have more adaptivity. However, it may not be possible to do so with a 10 MHz computer considering all those complicated computations; thus, do use fixed point instead of floating point to do multiplication, division, summation, trigonometry and integration. It will SIGNIFICANTLY speed up the processing time. If you decide using fixed point, DO NOT use floating point for ANYTHING AT ALL (not even a single floating point declaration) because your declaration or insturction will likly propagate, so do make sure you skim through the assembly code to see if there is any floating point instruction. And finally, you may want to probe around in matlab for testing purpose, but after you are done, please do implement it in C.

It is also important to eliminate noise in the hardware. That means reducing resistance (the length of the wire) will help (noise is proportional to reistance and temperature), and applying your layout magic to remove crosstalk. Adding a tunable hardware low pass, high pass filter may give you a much better time.

Application - Sensor Choice

WARNING! WORK STILL IN PROGRESS

Let's say we we want to find the orientation of the system in order to control our robot, plane, missle, or whatever. Everyone knows that a gyroscope will come into handy because of its sensitivity. With three angles provided by the gyroscope, we can easily define a 3d space precisely, but unfortunately, as time goes on, error will gradually build up. Therefore, we need to have other sensor to compensate for such error accumulation. Accelerometer will be a good choce to derive angles because of its non drifting quality, but it only gives you ONE 3d vector pointing to the ground. To define a 3d space precisely, we need to have two 3d vectors. In other words, we need one more sensor. Perhaps a GPS will be nice, but it is expensive, power hungry and also heavy in weight. I have considered a lot of alternatives, but using a 3d compass seems to be unavoidable. 3d compass is notorious for its difficulty in calibration, and its inaccuracy due to electromagnetic interference, especially if you have a motor or servo somewhere in your robot. Yet, no matter how reluntant I am towards using a compass, there are no other alternatives that can give you a cheap and non drifting readings. I guess people have been using a compass since stone age, so why not give it a try.

Application - Coordinate Syste,

To avoid confusion, we will use a right hand coordinate system with forward as y direction, right as our x direction, up as our z direction.

Once we defined the coordinate system, we can then find out what we are trying to find here.

Notice that with a acclerometer, which gives the vector sum of gravity and body acceleration, is not able to detect rotation on the z axis. It will have the same [0 0 -9.8] not matter how you rotate on the z axis; therefore with an accelerometer we can only obtain a yz and xz rotations. Also, I understand that if our robot accelerate, the sensor output is no long the gravity anymore, therefore, we are only able to obtain a gravity sample only when the magnitude of acceleration is 9.8, or somewhere close to it. Thus, we are going to use a gyroscope for short term state estimation and an sporadic acclerometer values for long term offset correction. Futhermore, I am not going to bother with rotation around the z axis since I believe I am missing one degree of freedom. If I do want to find out the rotation around the z axis, I may need a 2d compass or something. I don't think it is a good idea to rely on gyroscope alone for xy rotation without any offset compensation. (As a side not, it turns out that for digital compass, there is a twist to it. If the measurement is not sampled while the sensor is lying flat, it is not going to be correct. It needs to be compensated. Search google for compensated compass for more infomation)

Now back to equations. My state is

```
state = [angle_y velocity_y gyrooffset_y angle_x velocity_x gyrooffset_x]';
```

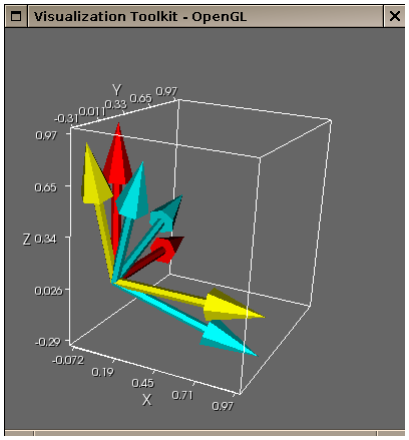
There are two things I need. I need to correct the current state based on sensor output using some equations, and I need to predict the next state based on again sensor output.

To do prediction, assume that the corrected state is correct and assume that our sensor is noisyless. What do we do if the world is so perfect? This is what I would do

```
function prediction = predict(corrected_state, sensoroutput);


predict_angle_y = corrected_angle_y + (gyro_y - corrected_gyrooffset_y) * sample_period;
predict_angle_x = corrected_angle_x + (gyro_x - corrected_gyrooffset_x) * sample_period;
predict_velocity_y = (gyro_y - corrected_gyrooffset_y);
predict_velocity_x = (gyro_x - corrected_gyrooffset_x);
predict_gyrooffset_x = corrected_gyrooffset_x;
predict_gyrooffset_y = corrected_gyrooffset_y;
```

For the acceloerometer, we need to convert the acceleration reading to angles, (rotation on y axis and rotation on x axis) before we can use it. It is hard to think in three dimensional, let's take a look at some pictures




```
framex = [1 0 0]'; framey = [0 1 0]'; framez = [0 0 1]';
frame(:, 1) = framex; frame(:, 2) = framey; frame(:, 3) = framez;

drawframe(frame, colorred);
drawframe(rotateX(20) * frame, coloryellow);
drawframe(rotateX(20) * rotateY(20) * frame, colorcyan);
```

The red arrow represents the neutral orientation. We first do a rotation on the x axis tilting it backwards and we have a orientation represented by the yellow arrows. We then do a rotation on its "y axis" we get the cyan color arrows. The z axis for the cyan arrows is what the accelerometer gives you. Well not really, the pictures here is showing a vector pointing up in the z direction, so gravity should be pointing down, but the calculation here is the same; they will end up having the same transformations. Now given this accelerometer value we need to work backward to find out what rotation we did.

```
sensor = rotateX(aX) rotateY(aY) gravity;
// working backward we need to find out aY, aX, but how?
gravity = rotateY(-aY) rotateX(-aX) sensor;
// do one step at a time
intermediateframe = rotateX(-aX) sensor;

// you will find that the y of the intermediate frame is 0
d   1    0        0        sensorx
0 = 0 cos(-aX) -sin(-aX) x sensory
e   0 sin(-aX)  cos(-aX)   sensorz

// after expand the middle equation,
sensory cos(-aX) - sensorz sin(-aX) = 0
aX = -atan(sensory / sensorz)

// expand the first equation
d = sensorx
// we know that the magnitude of intermediateframe is g
10^2 = d^2 + e^2
// e is then
e = sqrt(10^2 - d^2)

// gravity = rotateY(-aY) intermediateframe
    0       cos(-aY)  0 sin(-aY)    d
    0    =     0      1    0      x 0
  -10       -sing(-aY) 0  cos(-aY)   e

// expand the first equation
d cos(-aY) + e sin(-aY) = 0
aY = atan(-d / e)
```

In short, angle_x, angle_y are

```
function [angle_x, angle_y] = accel2angle(sensor)
anglex = rad2deg(-atan(sensory / sensorz))
e = sqrt(100 - sensorx * sensorx);
angley = rad2deg(atan(-sensorx / e))
```

Here are some tests to prove the math works

```
gravity = [0 0 -10]';
sensor = rotateX(20) * rotateY(25) * gravity = [-4.2262 3.0998 -8.5165]'
anglex = rad2deg(-atan(sensory / sensorz)) = 20.0
angley = rad2deg(atan(-sensorx / sqrt(100 - sensorx * sensorx))) = 25.000
// another example
sensor = rotateX(20) * rotateY(30) * gravity = [-5.0 2.9620 -8.1380]'
anglex = rad2deg(-atan(sensory / sensorz)) = 20.0
angley = rad2deg(atan(-sensorx / sqrt(100 - sensorx * sensorx))) = 30.000
```

Fortunately, this is not too complicated. Now that we have just finished part of the sensoroutput mentioned in the theory section, for the rest of sensoroutput ...

```
velocity_x = from gyro sensor
velocity_y = from gyro sensor
[angle_x, angle_y] = accel2angle(accelerometer)
sensoroutput = [velocity_x, velocity_y, angle_x, angle_y]';
```

Let's move on to state2output, which is fortunately damn simple

```
function [velocity_x, velocity_y, angle_x angle_y] = state2output(state)
velocity_x = state.velocity_x + state.gryooffset_x
velocity_y = state.velocity_y + state.gryooffset_y
angle_x = state.angle_x
angle_y = state.angle_y
```

We now have all the ingredients to ...

```
corrected_state = prediction + kalman_gain ( sensoroutput - state2output(prediction) );
```

We have everything now, but the gain. Ideally, we can carefully pick a gain manually, but that is no easy task. This is where the dude Kalman comes in. This dude took the derivative of the gain with respect to the error, set it to zero, and give a gain such that error can be minimized. He supplied you with a bunch of equations to calculate such gain. Let's stare at more pseudo code instead of staring at a bunch of equations. Before going to equations, lets find out the values for ...

Let's define H as the partial dervative of state2output function for each state. What the hell is that mean? state2output return [velocity_x velocity_y angle_x angle_y], the entry (column = 1, row = 1) is the partial derivative of velocity_x in terms of angle_x, the entry(column = 2, row = 1) is the partial derivative of velocity_x in terms of velocity_x.. etc.. You get the point and we will, in the end, have a matrix of size(sensoroutput) x size(state).

Let's define F as the partial deriviative of predict function for each state, just like what we did to H, but it should be a matrix of size(state) x size(state).

Let's define R as the measurement noise variance matrix of size(sensoroutput) x size(sensoroutput). This matrix is very similar to the identity matrix mostly filled with zero. The diagonal, however, is the standard deviation square, or the variance of each sensor.

I as the idenitity matrix of size(state) x size(state)

And finally Q as the state noise variance of size(state) x size(state). What I would do is define a vector of estimated standard deviation of the state. o = [a b c d ...] and I do Q = o' x o. You should make sure the standard deviation are proportional to each other. The displacement should be velocity times sample period, for example.

```
% set covariance to Q initially

% start to looping forever
% read sensor value from adc or whatever you read from
% may wanna update H, F, R, Q because it may depends on sensor values
kalman_gain = covariance H inverse(transpose(H) covariance H + R)
covariance = (I - H gain) covariance
% do our correction
state = state + kalman_gain ( sensoroutput - state2output(state) );
% output this state to our control system
covariance = F * corvariance * transpose(F) + Q
state = predict(state, sensoroutput);
% end of loop
```

For more information on the equations, go to page 33 of this dude's thesis.

It is important to apply those equations in exact order. Given an output, you do correction of the current state and then use that corrected state to make new prediction. You may ask whether these equations are the original version or the extended kalman filter. To tell you the truth, both! They have the exact same equations, but for extended kalman filter we just have H and F as a partial derivative matrix that change with the variables, whereas for original kalman filter it is just a plain old constant matrix. Another difference is that the gain of the original kalman filter will converge to some value, while the gain for extended kalman filter will vary in time.

One problem comes up quite often is that the state keeps on diverging but error remains small. Say the the real velocity is 3, the state velocity is 5, offset is 3. In next time frame, the velocity is again 3, but the state velocity is 1000, the offset is 997, for example. Diverging can be quite painful to debug, and requires deep understanding of what is going on.

All right, I am out. If you want to talk about this, you are very welcomed to send me an email through the contact page. Thanks for stopping by. I will be happy to hear from you on what I wrote on this page.

Written by Terence Tong
Statistics:
Go back to Home