Kalman Filter Made Easy

Terence Tong

October 12, 2005

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 . 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 adopt the same principle.

Now say you are "done" with her, and you are back to work. You need to know the a system state based on some noisy, perhaps redundant, sensors. 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?

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - h(\hat{x}_k^-)) \tag{1}$$

where $\hat{x}_k$ is the my new corrected state estimation; $\hat{x}_k^-$ is the estimation made from last time frame; K_k is the gain, which we will worry about it later. For now, it is just some constant; y_k is the measurement from an actual sensor; h is the function to convert a state to a measurement; $h(\hat{x}_k^-)$ is then the measurement prediction; and finally $(y_k - h(\hat{x}_k^-))$ is the measurement prediction error or the Innovation vector;

My next estimation is based on the new corrected estimation and the current sensor measurement. The estimation will be:

$$x_{k+1}^- = f(\hat{x}_k, u_k) \tag{2}$$

where u_k is the measurement from actual sensor; $\hat{x}_k$ is obtained by above equation; x_{k+1}^- is the new predicted state; f is the function related to the next state given old state and sensor input

To make the concept more concrete, an example of using Kalman filter to get rid of the notorious

gyroscope drift with be presented. The first question we want to know is what is the output and the input measurement. You want to know the angle displacement of the system, so the output or the state that we want to keep tract is simply the angle. We have only one gyro, so the input is a voltage measured by the sensor.

Let's take a look at these equations

$$\theta_{k+1} = \theta_k + \omega_k \delta \tag{3}$$

$$\theta_{k+1} = \theta_k + (y_k - bias_k)\delta \tag{4}$$

$$\omega_{k+1} = \omega_k \tag{5}$$

$$\omega_{k+1} = y_k - bias_k \tag{6}$$

$$bias_{k+1} = bias_k \tag{7}$$

where θ is my angle; ω is my angular velocity; δ is my sampling period; bias is the gyro bias in angular velocity; y is the gyro output

To convert state to measurement is usually the easiest part,

$$y_k^{gyro} = \omega_k + bias_k \tag{8}$$

Notice that I have two ways to model the theta and omega. However, the filter may not work, if you don't choose the right set of equations If we are not careful, the state can be diverge, meaning that the error, that is the difference between prediction and actual, can be small but the state is completely nonsense. By trying different combinations on matlab, I found that equation 6 and 4 works well. Diverging can be problematic, so if it is possible, try to keep the filter as simple as possible. It would not be a good idea to overdo in modeling the system. That will only give you a hard time. It is a good idea to avoid nonlinearity, although nonlinearlity can be fixed with Extended Kalman filter, which is not that different from the original Kalman filter, but it can exacerbate the diverging problem. Some people ask me what it mean to be linear. For a equation to be linear simply means that you can "pull" out a matrix out of the equation, such that you can do y = Ax. Also, people ask me what is the difference between y and u. The answer is that they are the same thing. Traditionally, it is written as two different symbols, just to be confusing.

With all these equations, we can now have a filter. By carefully guessing and trial error on the gain, we are able to get the filter tp work. However, getting the right gain can be tricky and time consuming. This is where Kalman filter comes in. Kalman takes the derivative of the gain with respect to the error to 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 some pseudo code instead of staring at a bunch of equations.

Start of the loop.

Get the sensor values y from sensors. Update H using the current state and input

$$K = PH'(HPH^T + VRV^T)^{-1} \tag{9}$$

$$P = (I - KH)P; \tag{10}$$

$$x = x + K(y - h(x)); \tag{11}$$

Save down the corrected x here for your control system. Next we are going to overwrite this x with the prediction of the next time frame. Update F here using the new x we have.

$$P = FPF^T + WQW^T; \tag{12}$$

$$x = f(x, y); \tag{13}$$

Another iteration.

Don't be overwhelmed by these equations. The dimensions of those variable can be confusing, so let's define the number of state variables be m and the number of sensors be n. K is the kalman gain, which has the dimension of m x n. That's the gain I have been talking about. It tells how much the state should be adjusted given the difference between the measurement and predicted measurement. Q is the state noise variance with dimension m x m. It basically tells you how much confidence you have on the state. If you are very confident, then you want a small value. How you obtain it is a mixture of trial error and calculation. Say you have w confidence on the acceleration, the value for velocity will be w x sample period, since a = vt. After you got that vector with dimension 1 x m, you multiply this vector by itself to obtain a m x m matrix. R is the measurement noise variance with dimension n x n. It is relatively easier to obtain this value since you can look it up at the datasheet for those sensor. People usually use the standard deviation squared times Identity(n) as your R. Of course, you can adjust it yourself if you feel like it. P is set to be Q before the iteration usually. Don't ask me why! H is the matrix to transform from your old state to measurement using the h function. If your equation is not linear, you simply take the partial derivative of the equation with respect to the measurement variables you have. It has a dimension of n x m. V is the partial derivative of the output function with respect to the output noise. Usually it is a Identity matrix of n x n. What the hell is the symbol I? This has been a mysterious variable for me, until I reliase that I is simply the identity matrix m x m. y is the output of your sensors with dimension of n x n, and it usually is the voltage level of your sensors. F is the matrix to transform to new state using old state. Again take the partial derivative of the f function with respect to each state variable, if your equation is not linear. W is the partial dervative of the state function f with respect to state noise, usually it is an identity matrix of m x m.

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.

With our models, what would be the values for those H, F, W, V....blah blah blah? To make things more interesting, let's introduce accelerometers which we can use to correct our drift.

$$x_k = \begin{bmatrix} \theta_k \\ \omega_k \\ bias_k^{gyro} \end{bmatrix} \tag{14}$$

$$y_k = \begin{bmatrix} y_k^{gyro} \\ y_k^{accel} \end{bmatrix} \tag{15}$$

$$f(x_k, u_k) = \begin{bmatrix} \theta_k + \delta(y_k^{gyro} - bias_k^{gyro}) \\ y_k^{gyro} - bias_k^{gyro} \\ bias_k^{gyro} \end{bmatrix} \tag{16}$$

$$F = \begin{bmatrix} 1 & 0 & -\delta \\ 0 & 0 & -1 \\ 0 & 0 & 1 \end{bmatrix} \tag{17}$$

$$h(x_k) = \begin{bmatrix} \omega_k + bias_k^{gyro} \\ \theta_k \end{bmatrix} \tag{18}$$

$$H = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \tag{19}$$

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{20}$$

$$W = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{21}$$

$$V = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{22}$$

$$R = \begin{bmatrix} sd_{gyro}^2 & 0 \\ 0 & sd_{accel}^2 \end{bmatrix} \tag{23}$$

$$Q = \begin{bmatrix} 0.2\delta \\ 0.2 \\ 0.1 \end{bmatrix} \begin{bmatrix} 0.2\delta & 0.2 & 0.1 \end{bmatrix} \tag{24}$$

The code is written up in octave, which is basically open source matlab. It can be found here.
http://www.ocf.berkeley.edu/ tmtong/howto/kalman/filter.m
The result of the filter can be found here
http://www.ocf.berkeley.edu/ tmtong/howto/kalman/output.pdf
Free free to go to my home page and write to me using the contact me page. Go to
http://omni.isr.ist.utl.pt/ mir/pub/kalman.pdf
for a more hardcore reference