EXTENDED KALMAN FILTER IMPLEMENTATION

Posted on September 10, 2018

Table of Contents

In this section, we will be finally implementing the extended kalman filter. This implementation is based on the following dissertation: Extended Kalman Filter for Robust UAV Attitude Estimation, Martin Pettersson. However, I have added in some other stuffs by myself as well, and the coding was done from scratch without referring to the pseudocode in the paper too.

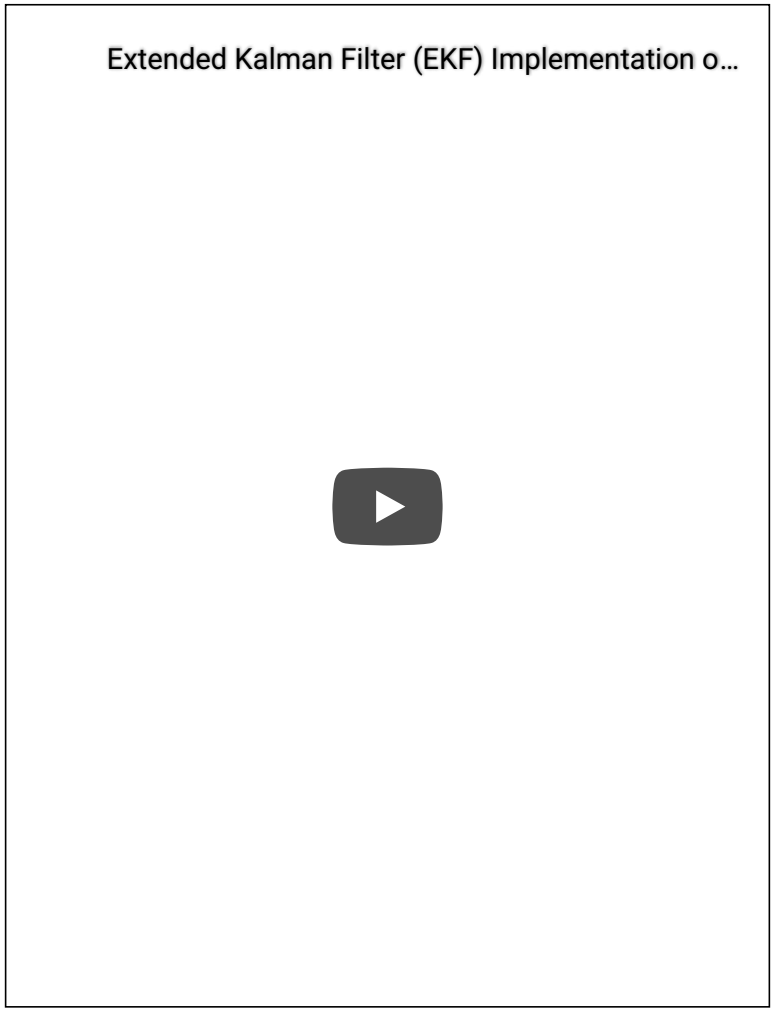
Let me first give a basic overview of this section before we go into the details. As we can see from section 3, using the gyrometer data alone is insufficient because it will eventually drift away from its original reference point over time. Furthermore, there is no way for the gyrometer to know the actual orientation after a certain period of time due to a lack of reference points. In order to alleviate these problems, we are going to use the accelerometer to provide a reference vector that is pointing downwards (gravity), and a magnetometer to provide another vector pointing in the magnetic north direction. With these 2 reference vectors, the orientation of the sensor will be fully defined thus we can use them as a reference to counter the drift of the gyrometer.

Another way of looking at this problem is that we will need at least 2 reference vectors in order to fully define a 3 dimensional orientation. As such, we will need at least 2 reference vectors for this method to work. You can have more to

improve the accuracy but you will definitely need to have at least 2.

Below is a video which shows the extended kalman filter implementation, and here are the files that I used in the video (and also for the section below)

- Arduino Code
- Python Code (EKF implementation)



Kalman Filter States

In order to use the Kalman Filter, we first have to define the states that we want to use. This is why there are so many different kalman filter implementations out there. Every author out there is saying that using their chosen states, you will be able to achieve a better result. However, for the purpose of this tutorial, we are just going to implement a really simple (in comparison to other implementations) one which is similar to the one implemented in my other post for determining a 1 dimensional angle using kalman filter.

Although I said that this is simple relatively, it is actually not that simple. Moving from 2D to 3D is really a big jump as you have to start using vectors and matrices, but that's not something to be afraid of so let us press on.

The states that we will be using for this implementation is given as follows:

$$x = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 & b^g{}_1 & b^g{}_2 & b^g{}_3 \end{bmatrix}^T$$

where

q_0 is the scalar term in the quarternion
q_1, q_2, q_3 is the vector term in the quarternion
$b^g{}_1, b^g{}_2, b^g{}_3$ is the bias of the gyrometer in the x, y, z direction respectively (units is the same as angular velocity).

The bias term refers to how much the gyrometer would have drifted per unit time. For more information, you can read up my other post for a 1 dimensional kalman filter implementation.

For this implementation, we can write the states in a more concise and compact manner as shown below.

$$x = \begin{bmatrix} \tilde{q} \\ \tilde{b^g} \end{bmatrix} \quad \text{————————} \quad (1)$$

where

$\tilde{q}$ is the quaternion
$\tilde{b^g}$ is the bias vector

Now, we need to form an equation for the system dynamics. From section 3, we know that

$$\dot{q} = \frac{1}{2} S(w) q = \frac{1}{2} S(q) w \quad \text{————————} \quad (2)$$

where

$$S(w) = \begin{bmatrix} 0 & -w_1 & -w_2 & -w_3 \\ w_1 & 0 & w_3 & -w_2 \\ w_2 & -w_3 & 0 & w_1 \\ w_3 & w_2 & -w_1 & 0 \end{bmatrix}$$

$$q = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}$$

$$S(q) = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix}$$

$$w = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}$$

In order to compensate for the gyrometer bias, we will need to add in the bias term into equation (2).

$$
\begin{aligned}
\dot{q} &= \frac{1}{2}S(w - b^g)q \\
&= \frac{1}{2}S(w)q - \frac{1}{2}S(b^g)q \\
&= \frac{1}{2}\begin{bmatrix} 0 & -w_1 & -w_2 & -w_3 \\ w_1 & 0 & w_3 & -w_2 \\ w_2 & -w_3 & 0 & w_1 \\ w_3 & w_2 & -w_1 & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} + \frac{1}{2}\begin{bmatrix} 0 & -b^g{}_1 & -b^g{}_2 & -b^g{}_3 \\ b^g{}_1 & 0 & b^g{}_3 & -b^g{}_2 \\ b^g{}_2 & -b^g{}_3 & 0 & b^g{}_1 \\ b^g{}_3 & b^g{}_2 & -b^g{}_1 & 0 \end{bmatrix}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}
\end{aligned}
\quad \text{———————} \quad (3)
$$

Next, we will need to discretize the above equation so that we can implement it in our program that runs in discrete time. Here, we are simply going to use the first order linearized model to make things simpler. Therefore,

$$\dot{q}(k) = [q(k+1) - q(k)]/T$$

where

T is the time taken between sample $k+1$ and k

Rearranging the equation,

$$q(k+1) = T\dot{q}(k) + q(k) \quad \text{———————} \quad (4)$$

Substituting equation (3) into equation (4),

$$q(k+1) = \tfrac{T}{2}S(w)q(k) - \tfrac{T}{2}S(b^g)q(k) + q(k) \quad \text{———————} \quad (5)$$

We can actually write equation (4) in a different manner as shown below such that it will be more meaningful when written in matrix form later.

$$q(k+1) = \tfrac{T}{2}S(q(k))w - \tfrac{T}{2}S(q(k))b^g + q(k) \quad \text{———————} \quad (6)$$

Thus, the system state equation can be written as follows.

$$
\begin{aligned}
x_{k+1} &= \begin{bmatrix} I_{4\times4} & -\frac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_k x_k + \begin{bmatrix} \frac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_k w_k \\
\begin{bmatrix} q \\ b^g \end{bmatrix}_{k+1} &= \begin{bmatrix} I_{4\times4} & -\frac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_k \begin{bmatrix} q \\ b^g \end{bmatrix}_k + \begin{bmatrix} \frac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_k w_k
\end{aligned}
\quad \text{———————}
$$

(7)

*I removed the tilde sign above the variables to make the equation look cleaner but take note that some of the parameters above are vectors while others are scalars (such as T).

If you expand equation (7), you will find that the first row will resolve into equation (6), and the second row simply says that the bias at time $k+1$ is the same as the bias at time k. This may look silly at first because we are just saying that the bias is constant. However, the Kalman Filter alters the states of the equation thus the value of the bias actually changes over time when we actually implement the algorithm.

For the sake of clarity, I am going to expand equation (7) so that there will be no confusion. Below is the expanded form of equation (7).

$$\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ b^g{}_x \\ b^g{}_y \\ b^g{}_z \end{bmatrix}_{k+1} = \begin{bmatrix} 1 & 0 & 0 & 0 & -\frac{T}{2}(-q_1) & -\frac{T}{2}(-q_2) & -\frac{T}{2}(-q_3) \\ 0 & 1 & 0 & 0 & -\frac{T}{2}(q_0) & -\frac{T}{2}(-q_3) & -\frac{T}{2}(q_2) \\ 0 & 0 & 1 & 0 & -\frac{T}{2}(q_3) & -\frac{T}{2}(q_0) & -\frac{T}{2}(-q_1) \\ 0 & 0 & 0 & 1 & -\frac{T}{2}(-q_2) & -\frac{T}{2}(q_1) & -\frac{T}{2}(q_0) \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ b^g{}_x \\ b^g{}_y \\ b^g{}_z \end{bmatrix}_k + \frac{T}{2} \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}_k \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_k$$

Before we move on to the actual implementation, let us first take a look at how we can use the accelerometer and magnetometer data in order get the reference vectors.

Accelerometer Data

The acceleration measured by the accelerometer module can be calculated if we know the acceleration of the body in the world frame, and the orientation of the body. Our reference vector is going to be the gravity vector, and we know that this vector will always point downwards in the world frame. Therefore, if we know the orientation of the body, we can predict the acceleration that the accelerometer is going to measure. This is all based on the assumption that external forces are negligible thus the only force that acts on the body is the gravitational force. As a result of this assumption, you will find that once you move the body hectically, the algorithm will not perform well. There are of course ways to alleviate this problem, but we will not go into there for the purpose of this tutorial.

The above paragraph can be summarized into the following equation.

$$^b a_m = {}^b R_w(-g) + {}^b e_a + {}^b b_a \quad \text{————————} \quad (8)$$

where

$^b a_m$ is the measured acceleration in the body frame by the accelerometer
$^b R_w$ is the rotation matrix for world frame to body frame
g is the gravitational vector in the world frame, $g = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$
$^b e_a$ is the accelerometer noise in the body frame
$^b b_a$ is the accelerometer bias in the body frame

Since we can determine all the variables on the right hand side (except for the noise term), it is possible to predict the measured acceleration. We can then use this prediction to compare with the actual measured acceleration to determine the error in our orientation. The Extended Kalman Filter (EKF) will then automatically help us convert this error into the bias term of the gyrometer. This is what I call the magic of the EKF because I do not need to know how the conversion is done. EKF handles it all as you will see in the later section.

For now, let us define some of the variables above. From the quaternion, it is possible to derive the rotation matrix with the following equation. Do read up on my previous post if you need more information on this.

$$ {}^{b}R_{w} = {}^{b}R_{w}{}^{T} = R(q)^{T} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}^{T} $$

$$ {}^{b}R_{w} = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} $$

———————– (9)

Since we know that $g = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^{T}$, we can substitute equation (9) into equation (8) and then simplify equation (8) to get the following equation.

$$ \begin{bmatrix} {}^{b}a_x \\ {}^{b}a_y \\ {}^{b}a_z \end{bmatrix} = -\begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3 + q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} + \begin{bmatrix} {}^{b}b_x \\ {}^{b}b_y \\ {}^{b}b_z \end{bmatrix} + \begin{bmatrix} {}^{b}e_x \\ {}^{b}e_y \\ {}^{b}e_z \end{bmatrix} \quad \text{———————–} \quad (10) $$

*take note that in the above equation (10), g is a scalar.

Notice in equation (10) that the matrix with quaternion terms is non-linear. In order to use the Kalman Filter, we have to write equation (10) in the form of $y = Cx + D$ where x is the state matrix as shown in equation (1) and y is the term on the left hand side of equation (10). You will realize that this is not possible because of the non-linearity. One possible solution is that we can linearize equation (10) near its “operating point”. This is exactly what the Extended Kalman Filter is doing, and also why you need to use the extended form of the Kalman Filter when working with rotation matrices because they are mostly non-linear.

So, how do we go about linearizing equation (10)? We can rewrite equation (8), the unexpanded form of equation (10), evaluated at time t as shown below.

$$ ({}^{b}a_m)_k = -gh_a(q_k) + C_k \quad \text{———————–} \quad (11) $$

where

$({}^{b}a_m)_k$ is the measured acceleration in the body frame by the accelerometer at time step k
g is the gravitational constant (take note that g is a scalar here)
$h_a(q_k)$ represents a non-linear function in q_k, where q_k is the quaternion at time step k
C_k represents the other terms in equation (8), ${}^{b}e_a + {}^{b}\ b_a$, evaluated at time step k.

Just for clarity, I am going to write the values of $h_a(q_k)$ below so that there will be no misunderstandings.

$$ h_a(q_k) = \begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3 + q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}_k $$

With this, what we need to do now becomes much clearer. We have to linearize the non-linear function $h_a(q_k)$, and one way of doing so is to find its Jacobian (gradient) at time $k - 1$. In doing so, we are assuming that the non-linear

function is approximately linear between 2 points adjacent in time. This also means that the approximation will be better if you have a shorter time-step between your iterations.

In order to linearize a function, we will call upon the mighty Taylor Expansion as shown below.

$$f(x)|_{x=a} = f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \dots$$

where $f(x)$ is a non-linear function in x.

Applying the above to our non-linear function, we will get the following equation.

$$h_a(q_k)|_{q_k=q_{k-1}} = h_a(q_{k-1}) + h'_a(q_{k-1})(q_k - q_{k-1}) + \dots \quad \text{————————-} \quad (12)$$

where

$$h'_a(q_{k-1}) = \frac{\delta h_a(q)}{\delta q}\Big|_{q=q_{k-1}} = \begin{bmatrix} \frac{\delta h_{a1}}{\delta q_0} & \frac{\delta h_{a1}}{\delta q_1} & \frac{\delta h_{a1}}{\delta q_2} & \frac{\delta h_{a1}}{\delta q_3} \\ \frac{\delta h_{a2}}{\delta q_0} & \frac{\delta h_{a2}}{\delta q_1} & \frac{\delta h_{a2}}{\delta q_2} & \frac{\delta h_{a2}}{\delta q_3} \\ \frac{\delta h_{a3}}{\delta q_0} & \frac{\delta h_{a3}}{\delta q_1} & \frac{\delta h_{a3}}{\delta q_2} & \frac{\delta h_{a3}}{\delta q_3} \end{bmatrix}_{k-1}$$

$$h'_a(q_{k-1}) = 2\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1} \quad \text{————————-} \quad (13)$$

And there we have it! Let us now substitute equation (13) into equation (12).

$$h_a(q_k)|_{q_k=q_{k-1}} = \begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3 + q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}_{k-1} + 2\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1} \left(\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_k - \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_{k-1} \right)$$

————————- (14)

Now that we have the linearized form of the non-linear equation, let us write out the whole linearized solution of equation (10).

$$\begin{bmatrix} {}^b a_x \\ {}^b a_y \\ {}^b a_z \end{bmatrix}_k = -g\left[\begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3 + q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}_{k-1} + 2\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1} \left(\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_k - \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_{k-1} \right) \right] + \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix}_k$$

————————- (15)

Take note that g is a scalar term here. In the Python source code, you will notice that the C_k matrix is missing. This is because I calibrated the accelerometer at first to deal with the bias. In other words, I was working with a 0 bias accelerometer value so the C_k term becomes 0.

Let us now write equation (15) in a more compact matrix form so that it will be clearer.

$$y = Cx + D$$

where

$$y = ({}^{b}a_m)_k = \begin{bmatrix} {}^{b}a_x \\ {}^{b}a_y \\ {}^{b}a_z \end{bmatrix}_k$$

$$C = -g[2h'_a(q_{k-1})] = -2g\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1}$$

$$x = q_k = \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_k$$ (here, the bias terms in equation (1) are treated as zeros)

$$D = -g[h_a(q_{k-1}) - 2h'_a(q_{k-1})(q_{k-1}) + C_k] = -g\left(\begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3 + q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}_{k-1} - 2\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1}\begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix}_k\right)$$

We now have a system of linear equations, albeit being a little ugly. Actually, in the kalman filter implementation, we are only going to use matrix C (the Jacobian matrix) thus the rest of the terms are actually not needed. There are some mathematical proofs for this, but that is beyond the scope of this tutorial. We are now finally done with the accelerometer section.

Magnetometer Data

The magnetometer provides us with a vector that is always pointing to the magnetic North. This vector changes (elevation of the vector changes depending on the altitude) over the surface of the Earth thus we will need a map of the vectors if we want a reference vector that works across the globe. However, in this project, we will be assuming that the change in the North reference vector is negligible since the sensor will not be moving around much (probably only within the room).

The magnetometer actually gives a 3 directional reference vector instead of a simple North heading in 2 dimensions. Due to this, there are some complications that arise. In this project, I will make the assumption that the accelerometer is accurate in providing the reference vector in the vertical plane, and the magnetometer data is accurate in providing the reference vector in the horizontal place. As a result of this, we do not want the magnetometer data to affect the accelerometer data and vice versa. Therefore, we are going to remove 1 dimension from the magnetometer reference vector.

Firstly, we will calibrate our magnetometer based on my previous post so that we can get a 3 dimensional unit vector from the raw magnetometer data. If you follow the method in the previous post, you should be able to determine the values of A^{-1} and b so you can apply the calibration equation as follows.

$$h = A^{-1}(h_m - b)$$

where

h is the actual magnetic field
h_m is the measurement reading from the magnetometer with errors

Next, in order to remove 1 dimension (the vertical plane) from the magnetometer vector, we have to transform the coordinates from the body frame (measurements are done in the body frame) to the world frame first (because the vertical plane that we want to remove exist in the world frame). Therefore, we have to get the rotation matrix that converts the body frame to the world frame from our quaternion. This can be done through equation (1) from my previous post which is shown below as well.

$$r' = Cr$$

where

$$C = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 - q_0q_3) & 2(q_1q_3 + q_0q_2) \\ 2(q_1q_2 + q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 - q_0q_1) \\ 2(q_1q_3 - q_0q_2) & 2(q_2q_3 + q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

In this case, r will be the unit vector calibrated magnetometer data in the body frame. Therefore,

$$r = \begin{bmatrix} {}^{b}m_1 \\ {}^{b}m_2 \\ {}^{b}m_3 \end{bmatrix}$$

$$r' = \begin{bmatrix} {}^{w}m_1 \\ {}^{w}m_2 \\ {}^{w}m_3 \end{bmatrix}$$

where

${}^{b}m_1, {}^{b}m_2, {}^{b}m_3$ is the calibrated magnetometer data in the body frame's x, y, z direction respectively.
${}^{w}m_1, {}^{w}m_2, {}^{w}m_3$ is the calibrated magnetometer data in the world frame's x, y, z direction respectively.

Now, we can safely remove the z-axis from our data by letting ${}^{w}m_3 = 0$, then re-normalizing the vector such that it stays as a unit vector but only in 2 dimensions now. We will then rotate it back to the body frame and use the resulting vector (in place of the actual calibrated measured data) for our kalman filter update section later on. On a side note, you will find that even without doing all these, the kalman filter algorithm will work. However, whether it works better with or without is another story to be told another day.

In the accelerometer section, we have the gravity vector as our reference vector. On the other hand, we have the North vector as a reference for our magnetometer. For me, since the North vector points exactly in the direction of the negative y-axis when I placed the sensor parallel to my table, I simply took the negative y-axis (in the world frame) as my reference vector. A better approach would be perhaps to have a period of time for the initialization process to set the current North vector as the reference vector. In this way, all we have to do is place the object in the desired direction for it to use the direction as a reference for calculation.

As a result of setting the y-axis to be the reference vector, in place of $g = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}^T$, I have $m = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$. (in the python code, i added in the negative sign into the gravity reference vector)

Moving on, once again, we need a linear equation for the output of our system in order for us to use the kalman filter. The output that we want to get here is the predicted accelerometer and magnetometer data from our kalman filter states (quaternion). One way of doing so is through the rotation matrix which can be derived from a quaternion. Similar to the accelerometer, we can use the following equation to convert the magnetometer reading from the world frame to the body frame (so that we can compare it to the actual measured magnetometer value).

$$^{b}m_m = ^{b}R_w(^{w}m_r) + ^{b}e_{mag} + ^{b}b_{mag} \quad \text{————————} \quad (16)$$

where

$^{b}m_m$ is the measured magnetic field in the body frame by the magnetometer
$^{b}R_w$ is the rotation matrix for world frame to body frame
$^{w}m_r$ is the reference magnetic North vector in the world frame, where in my case $^{w}m_r = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$
$^{b}e_{mag}$ is the magnetometer noise in the body frame
$^{b}b_{mag}$ is the magnetometer bias in the body frame

The rotation matrix $^{b}R_w$ is actually the same matrix as what was used in the accelerometer. Below is the exact same equation (9).

$$^{b}R_w = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1q_2 + q_0q_3) & 2(q_1q_3 - q_0q_2) \\ 2(q_1q_2 - q_0q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2q_3 + q_0q_1) \\ 2(q_1q_3 + q_0q_2) & 2(q_2q_3 - q_0q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$
———————— (9)

Since we know that $^{w}m_r = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$, we can substitute equation (9) into equation (16) and then simplify equation (16) to get the following equation.

$$\begin{bmatrix} ^{b}m_x \\ ^{b}m_y \\ ^{b}m_z \end{bmatrix} = -\begin{bmatrix} 2(q_1q_2 + q_0q_3) \\ q_0^2 - q_1^2 + q_2^2 - q_3^2 \\ 2(q_2q_3 - q_0q_1) \end{bmatrix} + \begin{bmatrix} ^{b}b_x \\ ^{b}b_y \\ ^{b}b_z \end{bmatrix} + \begin{bmatrix} ^{b}e_x \\ ^{b}e_y \\ ^{b}e_z \end{bmatrix} \quad \text{————————} \quad (17)$$

From here, you can see that in order to obtain the Jacobian matrix for use in the Extended Kalman Filter, the steps are exactly the same as that for the accelerometer. As such, I am going to skip the derivations and just write down the value of the C matrix below here.

$$C = -2\begin{bmatrix} q_3 & q_2 & q_1 & q_0 \\ q_0 & -q_1 & q_2 & -q_3 \\ -q_1 & -q_0 & q_3 & q_2 \end{bmatrix}_{k-1}$$

Here, notice that if we did not simplify the equations with the values of $^{w}m_r$ for equation (16), we will get the exact same generalized Jacobian matrix from equation (8) (without the values of the vector g being substituted) as well. I will leave this as homework to you guys but the answer can be found in the python code anyway. I was lazy to write 2 different implementations for the accelerometer and the magnetometer so I combined them both. Anyway, we are now ready to get on with the EKF implementation finally!

Quaternion EKF Implementation

Now, if you have no experience with the Kalman Filter at all, I would strongly recommend that you read one of my earlier post on kalman filter to get an idea of it first. The extended kalman filter is simply replacing one of the the matrix in the original original kalman filter with that of the Jacobian matrix since the system is now non-linear.

The equations that we are going to implement are exactly the same as that for the kalman filter as shown below. (This is taken directly from my earlier post except that i added k to the numbering to symbolize that it is the kalman filter equation)

Prediction

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \quad \text{———-} \quad (1k)$$

$$P_k^- = AP_{k-1}A^T + Q \quad \text{———-} \quad (2k)$$

where
$\hat{x}_k^-$ is the priori estimate of x at time step k
P_k^- is the priori estimate of the error at time step k
Q is the process variance

Update

$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R} \quad \text{———-} \quad (3k)$$

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-) \quad \text{———-} \quad (4k)$$

$$P_k = (I - K_k C)P_k^- \quad \text{———-} \quad (5k)$$

where
y_k is the actual measured state
$\hat{x}_k$ is the posteri estimate of x at time step k
P_k is the posteri estimate of the error at time step k
K_k is the kalman gain at time step k
R is the measurement variance

※The other parameters in equation 1 to 5 (such as the constants A, B and C) will be described in detail in below.

Let us go through the equations in order.

Prediction

Equation 1k

$$\hat{x}_k^- = A\hat{x}_{k-1} + Bu_k \quad \text{———-} \quad (1k)$$

If you know the kalman filter well, you would have realized that we have derived this equation already in the earlier part of this post. Below is a copy of the equation (7) from the kalman filter states section above except that I replace k with $k-1$.

$$\begin{bmatrix} q \\ b^g \end{bmatrix}_k = \begin{bmatrix} I_{4\times4} & -\frac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_{k-1} \begin{bmatrix} q \\ b^g \end{bmatrix}_{k-1} + \begin{bmatrix} \frac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_{k-1} w_{k-1}$$

From the above equation, we can determine our A and B matrices.

$$A = \begin{bmatrix} I_{4\times4} & -\frac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_{k-1}$$

$$B = \begin{bmatrix} \frac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_{k-1}$$

where

$$S(q) = \begin{bmatrix} -q_1 & -q_2 & -q_3 \\ q_0 & -q_3 & q_2 \\ q_3 & q_0 & -q_1 \\ -q_2 & q_1 & q_0 \end{bmatrix}$$

With all these information, we can now determine $\hat{x}_k^-$ in equation (1k). Once this is done, we have to normalize the quaternion in our kalman filter states dues inaccuracies in the discretized calculation. Next up is equation (2k).

Equation 2k

$$P_k^- = AP_{k-1}A^T + Q \quad \text{———-} \quad \text{(2k)}$$

This equation is pretty straightforward. However, we have to decide an initial value for the P matrix and the Q matrix. You should play around with the values in the code to see how this affects the overall performance of the algorithm. You will realize that if you start with a P matrix with huge values, it may not converge to give you a coherent result. On the other hand, if you start with a P matrix with small values, it may take quite a while for the P matrix to converge to the actual solution. The Q matrix is the process variance, and it represents the inaccuracies of the model that we are using.

In the code, equation (1k) and (2k) are both done within the `predict()` method as shown below. (predictAccelMag() will be explained under equation (3k))

Kalman Filter Prediction Step — Python

```
def predict(self, w, dt):
    q = self.xHat[0:4]
    Sq = np.array([[-q[1], -q[2], -q[3]],
                   [ q[0], -q[3],  q[2]],
                   [ q[3],  q[0], -q[1]],
                   [-q[2],  q[1],  q[0]]])
    tmp1 = np.concatenate((np.identity(4), -dt / 2 * Sq), axis=1
    tmp2 = np.concatenate((np.zeros((3, 4)), np.identity(3)), ax
    self.A = np.concatenate((tmp1, tmp2), axis=0)
    self.B = np.concatenate((dt / 2 * Sq, np.zeros((3, 3))), axi
    self.xHatBar = np.matmul(self.A, self.xHat) + np.matmul(self
    self.xHatBar[0:4] = self.normalizeQuat(self.xHatBar[0:4])
    self.xHatPrev = self.xHat

    self.yHatBar = self.predictAccelMag()
    self.pBar = np.matmul(np.matmul(self.A, self.p), self.A.tran
```

Update

Equation 3k

$$K_k = \frac{P_k^- C^T}{CP_k^- C^T + R} \quad \text{———-} \quad (3k)$$

In order to implement equation (3k), we first have to figure out what is our C matrix here. If you take a look at my previous post explaining the kalman filter using the pendulum example, you will know that the C matrix is a matrix to convert our kalman filter states to the measured variables. In the pendulum example, it just so happens that the measured variables are the same as the kalman filters states thus the C matrix is the identity matrix. However, in our case here, our measured variables are the direction of the gravity and North vector while our kalman filter states are the quartenion and the gyro bias. They are essentially different parameters thus we have to figure out a C matrix which allows us to convert out state variables into the measured variables.

Now, if you had diligently read through everything in this post, you should realize that we have already derived our C matrix in the accelerometer and magnetometer section.

For the Accelerometer,

$$C_a = -2\begin{bmatrix} -q_2 & q_3 & -q_0 & q_1 \\ q_1 & q_0 & q_3 & q_2 \\ q_0 & -q_1 & -q_2 & q_3 \end{bmatrix}_{k-1}$$

Notice here that I removed the multiple g because the accelerometer data that I will be using is in units of g.

For the Magnetometer,

$$C_m = -2\begin{bmatrix} q_3 & q_2 & q_1 & q_0 \\ q_0 & -q_1 & q_2 & -q_3 \\ -q_1 & -q_0 & q_3 & q_2 \end{bmatrix}_{k-1}$$

In order to determine the C matrix, we would require the quaternion state from the previous iteration. Since the conversion of quaternion to rotation matrix from the world frame to the body frame is the same for the accelerometer and the magnetometer, there is actually a more generalized form of the C equation that

will work for both the accelerometer and the magnetometer. I left that as a homework in the above section but the answer is actually already within the code under the method `getJacobianMatrix()`.

For the purpose of clarity, I am once again going to write out the exact equation that is implemented within the python code.

$$\hat{y}_k^- = C\hat{x}_k^-$$
$$\begin{bmatrix} \hat{a}_m \\ \hat{m}_m \end{bmatrix}_k = \begin{bmatrix} C_a & 0_{3\times3} \\ C_m & 0_{3\times3} \end{bmatrix} \begin{bmatrix} q \\ b^g \end{bmatrix}_k$$

The last term that we have to determine in order to implement equation (3k) is the R matrix. This is the measurement variance and it represents how accurate our measurement data is. Similar to the Q matrix in equation (2k), this term is important yet difficult to determine. The values of these terms are usually determine through simulations or actual tests so in this tutorial, I am just going to use an arbitrary value which does not have any special meaning.

Equation 4k

$$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-) \quad \text{———-} \quad (4k)$$

Next up is equation (4k) which by now we should have all the terms except for y_k. This is the actual measurement values that we obtain from the sensor data. In my python code, this is expressed as shown below.

$$y_k = \begin{bmatrix} a_m \\ m_m \end{bmatrix}$$

Since we are making an addition to the quaternion, we have to normalize it again after this step to maintain a unit quaternion during the calculations.

Equation 5k

$$P_k = (I - K_k C)P_k^- \quad \text{———-} \quad (5k)$$

As for equation (5k), we already have all the variables ready so implementing it will be no problem at all. Equation (3k) to (5k) are all implemented within the `update()` method in the python code as shown below as well.

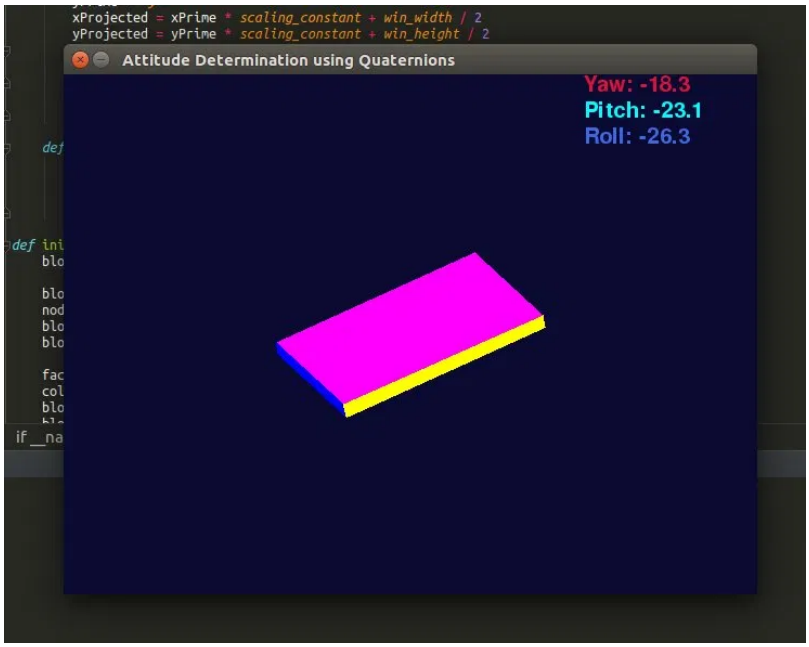
Kalman Filter Update Implementation — Python

```python
def update(self, a, m):
    tmp1 = np.linalg.inv(np.matmul(np.matmul(self.C, self.pBar),
    self.K = np.matmul(np.matmul(self.pBar, self.C.transpose()),

    magGuass_B = self.getMagVector(m)
    accel_B = self.getAccelVector(a)

    measurement = np.concatenate((accel_B, magGuass_B), axis=0)
    self.xHat = self.xHatBar + np.matmul(self.K, measurement - s
    self.xHat[0:4] = self.normalizeQuat(self.xHat[0:4])
    self.p = np.matmul(np.identity(7) - np.matmul(self.K, self.C
```

We are now done with the implentation! Go ahead and run the program to check out how awesome it is! Here's a picture of how it the program should look like when it is run! (:



PYTHON IMPLEMENTATION DISPLAY

Conclusion

This project took about 2 months for me to complete, partly because of my lack of knowledge, and also partly due to the inaccurate sources that I was reading up in the internet. Now that I am finally at the conclusion, I must say that I have learnt a lot, and there are actually a lot more that can be done to improve the program/algorithm.

One thing that I felt was weird was the equation (4K) where we added the difference between the estimated values and actual measured values to our quaternion. For most parameters, this is perfectly normal. For example, you can add to accelerations or velocities to say that it is accelerating or moving faster. However, additions to quaternion do not have any symbolic meaning. In fact, what we wanted to achieve is to add a small rotation to correct for the orientation thus we should not be adding to the quaternion, but to the rotation angle in my opinion. However, when implemented, the above works so I'm not really sure what is going on with the mathematics in there. There are journal papers which does what I described but the complexity is on a whole new level so I am not going to introduce it. If you are interested, they are called "Multiplicative Extended Kalman Filter (MEKF)" and a simple search in google should show many related articles.

Another thing that I felt could be done better (but I have not done it yet) is the calibration of the magnetometer. We have successfully mapped the data points on a unit sphere, but the points may still be skewed to one direction. There must be a way for us to take references from different angles and spread the points evenly between the 360 degree azimuth. Also, if you take a careful look at the implementation of equation (3k), you will realize that we are actually inverting a 6×6 matrix. Numpy in python knows how to do it, but not me! Also, inverting huge matrices are often very computationally costly so we should find ways to reduce the dimension of the matrix being inverted as much as possible. There is actually another form of Kalman Filter for this called the Iterated Kalman Filter. I have not yet read it through but it seems to be able to solve the problem that I just mentioned.

There are clearly much more work that we can do to improve the logic but for now, lets run the program and enjoy the fruits of our labour! (:

PREVIOUS POST
Calibrating the Magnetometer

NEXT POST
Arduino Real Time Frequency Plot with Python

POSTED IN ATTITUDE DETERMINATION WITH QUATERNION USING EXTENDED KALMAN FILTER

36 comments on "*Extended Kalman Filter Implementation*"

Vincenzo Junior Forte
September 18, 2018 at 12:49 am **Reply**

Hi, I would like first of all to congratulate you for the good work you have done. I'm following your tutorial step by step but using other sensors (MPU9250 and MetaMotionC Board).
By using theirs API, I receive data with the correct units (g for the accelerometer, dps for the gyroscope and uT for the magnetometer), so, how should I modify the EKF filter in these cases? For example, how should I define 'mag_Ainv' and 'mag_b'? And also, why you normalize in 'getAccelVector()' and 'getMagVector()'?

Riki
September 18, 2018 at 3:37 pm **Reply**

Hi Forte,

Thanks for reading my post.
I am currently away on a business trip this week so i have no access to my setup so i can only answer simple questions.

For mag_Ainv and mag_b, you need to follow my previous post on calibrating the magnetometer. Its in the same series of tutorial so you can reach the page from the list of contents at the top of the page. To summarize it, you have to capture data and perform the calibration for the magnetometer.

As for the normalization, i wanted to work work with unit vectors so that it is much easier to debug and simpler to work with.

Hope this helps you!

shengkai
October 4, 2018 at 7:51 am **Reply**

hi, this is a wonderful post. I have questioned how to make sure the actual direction(heading) when the sensor move? I have lots questions want to discuss with you. If you have free time. Please touch with me.

rikisenia.L
October 4, 2018 at 11:15 am **Reply**

Hi Shengkai,

Thanks for going through my post.
I will be grateful if you could post your questions here so that others can benefit from it as well.
I will reply as soon as possible to all comments posted on my blog.

shengkai
October 5, 2018 at 5:18 am **Reply**

I have the imu data, I want to use gyro and accel data to make sure the vehicle displacement and drawing the trajectory. But the direction is difficult to combine with displacement. Do you know how to solve it?

rikisenia.L
October 5, 2018 at 9:27 am **Reply**

When you have an acceleration, the model that I presented above no longer works because I assumed that acceleration is negligible. Thus if you simply apply the model presented in the post, it will not give accurate orientation results.

This problem, however, can be solved if you have another sensor that is capable of measuring the speed (GPS or pitot tube etc) or the displacement (odometer etc) so that you can determine the acceleration of the vehicle from another source. Using this acceleration, we would be able able to determine the actual direction of the gravity vector (since the measurement from the accelerometer is the vector sum of all accelerations that are acting on the body).

When you are able to provide an accurate gravity vector, the method in the post will work again so you will be able to determine the heading of the vehicle. All that is left is to apply the displacement at every calculation iteration in the direction of the heading and you should get the result that you want.

Of course, the method that I just presented is not the only method available so if you have any other innovative ideas, please feel free to share them! I hope this will help you in your work!

shengkai
October 8, 2018 at 9:56 am

how about ultrasonic sensor to get the displacement

between two sample time?

rikisenia.L
October 8, 2018 at 2:10 pm **Reply**

Hi Shengkai,

I am not too sure how your system is like but an ultrasonic sensor to get the displacement would possible work. In fact, any displacement, velocity, acceleration sensor other than an accelerometer would work.
However, you would need to adapt the code to include the sensor data as well.

shengkai
October 9, 2018 at 3:21 am **Reply**

Hi Risisenia,

I am confused with how to remove gravity vector in x y z-axis , if I get velocity data, is that useful to improve the accuracy of the accelerometer? What the next step to get trajectory? You know, there has lots algorithm, so, I am confused. For my system, I just want to use IMU to get a trajectory for my vehicle displacement. For the straight path, my system works. But when I want to get a circle path, the trajectory was confusion. what did you think?? thank you for your time to answer my question.

rikisenia.L
October 10, 2018 at 12:41 am **Reply**

Hi Shengkai,

It is actually opposite of what you wrote. We want another sensor data for displacement/velocity/acceleration to remove the vehicle acceleration from the accelerometer reading. In this manner, we will be able to determine the direction of the gravity vector. The gravity vector acts as a reference point for our gyrometer which determines the orientation of the vehicle. Without a reference vector, the noise from the gyrometer will cause the orientation measurements to drift away from the true value over time.

A straight path is a simple case because you do not have to consider the orientation (heading) of your vehicle. When you have a circular path, you need additional sensors to measure the orientation of the vehicle in order to draw the correct trajectory. Furthermore, if you are using an accelerometer, you will need to consider the centripetal acceleration as well.

Implementing what I just said would require more than just modifying some variables in the code provided in the post. You will need to formulate new equations for the system to accommodate the new sensor measurements, and then reflect those changes within the code.

shengkai
October 10, 2018 at 2:56 am **Reply**

Ok, I get it. I will try to do that. thanks for your answer.

shengkai
October 10, 2018 at 7:06 am **Reply**

YES. I just buy another sensor to determine the velocity and displacement. If I also have a problem, could I discuss with you?

rikisenia.L
October 10, 2018 at 10:45 pm **Reply**

Yeah, sure. Just leave a comment and I'll reply as soon as possible.

orhan
October 31, 2018 at 5:05 pm **Reply**

it was helpful
could you put the best references you used, please

rikisenia.L
October 31, 2018 at 9:36 pm **Reply**

You are right. I thought I had referenced it at the start of the tutorial but apparently I hadn't.
I have added in the link to the reference at the start of this post.

Anyway, this implementation is based on the following dissertation: Extended Kalman Filter for Robust UAV Attitude Estimation, Martin Pettersson

However, I would not say that it is 100% alike because I tweaked it in places where I think it would make more sense if I changed it. Furthermore, the coding was all done from scratch so I did not follow the pseudocode in the paper as well.

Sed

April 26, 2019 at 9:10 am **Reply**

As you mentioned once we move the body hectically, the algorithm will not perform well. Do you have any idea to include the linear acceleration a in the measurement model?

rikisenia.L

May 8, 2019 at 11:33 pm **Reply**

Hi Sed,

Apologies for the late reply. I was on a holiday.
One of the reference vector that I used is the gravity vector which is determined through the use of the accelerometer. As such, when huge linear accelerations are present, it becomes difficult to correctly identify the direction of the gravity vector.

For example, when the accelerometer is at free fall, we will have no reliable reference gravity vector anymore thus the algorithm will not work at all. Now, imagine accelerating the accelerometer downwards by 2g. This will result in a reading of -g by the accelerometer which is the same reading as when you turn the accelerometer upside down. This means that we cannot differentiate between the 2 scenarios (unless you keep track of the past details).

If you can find a way to separate the linear acceleration from the gravitational acceleration, perhaps it will aid in improving the performance of the algorithm.
One method which is often used is to add a GPS to track the movement of the object to predict the acceleration of the body. This will then allow us to determine the gravity vector more accurately by subtracting away the linear acceleration of the body from the measured values.

Sed
May 23, 2019 at 2:30 pm **Reply**

The output of predictAccelMag () is supposed to be $Cxhat^-$ which is basically $yhat^-$. But in your code the output of this function is different. Could you please explain that? Thanks

rikisenia.L
May 25, 2019 at 12:24 am **Reply**

Ahh, you caught on a subtle point.
My thinking fell along this line of reasoning.
"yHatBar" is the predicted accelerometer and magnetometer values.
If we calculate the values using the Jacobian (C matrix), it is essentially a linearized estimate. Furthermore, if you take a look at the linearized equation (15), you can see that there is actually a constant "D" that we should be adding into our calculations. If we simply multiply "C" with "xHatBar", we are actually ignoring the constant which will result in an erroneous linearization.

Now, there is actually a different way altogether to predict the accelerometer and magnetometer values. We can simply multiply the rotational matrix with the gravity vector as shown in equation (8) (ignore the bias and noise term). This is actually how I predicted the accelerometer and magnetometer values in my code, given by these 2 lines:

accelBar = np.matmul(getRotMat(self.xHatBar).transpose(), self.accelReference)
magBar = np.matmul(getRotMat(self.xHatBar).transpose(), self.magReference)

Sorry for the confusion. I should have written about this in my article. Hope this clears things up!

Sed
June 10, 2019 at 9:10 am **Reply**

I have a question about the reference magnetic vector. Here you said I assume the magnetometer data is accurate in providing the reference vector in the horizontal plane and you chose m_r = [0, -1, 0].
But I am seeing in some papers (like Madgwick's paper) they used m_r = [mx, 0, my]. I would greatly appreciate if explain what is the difference and why they choose such a reference vector.

Sed
June 10, 2019 at 9:12 am **Reply**

Correction! m_r = [mx, 0, mz]

rikisenia.L
June 11, 2019 at 11:21 pm **Reply**

Hi Sed,

I think it will easier to understand if we talk about the gravity reference vector first.
In my case, I used [0, 0, -1] to show that the reference gravity vector is pointing in the negative z direction. Assume that the accelerometer is now vertical and we get a reading of [1, 0, 0] after normalizing the values. We know that this means that the accelerometer's x-axis is pointing downwards. The reason we know this is because we know that gravity acts in the downward direction (regardless of how we turn our sensor) and since the x-axis shows a reading of 1, it must be pointing in the downward direction. This is the importance of the reference vector. It helps to tell the sensor its orientation based on some information that we know beforehand.

Likewise for the magnetometer's reference vector, we need to know where the actual reference vector is pointing in the world frame, so that we can use it as a reference to get the orientation of our sensors. One thing you need to take note about magnetometers is that the magnetic field of the Earth varies with the location, particularly with places of different latitudes. I did my experiment in Japan while Madgwick did his somewhere else on the globe so I won't be surprised that his choice of reference vector is different from mine. In addition, it also has to do with your placement of the sensor. For example, if I were to rotate my sensor 180 degrees in the vertical axis, my reference vector will turn into [0, 1, 0] instead.

To put it simply, in the world frame, there is a vector pointing in a certain direction (for example, the gravity vector points downwards). The reference vector is how this unchanging vector (for example, the gravity vector) in the world frame is represented in the local frame of the sensor when it is in its default orientation.

Sed
June 12, 2019 at 12:40 pm **Reply**

Thanks so much for your explanation

SensorNovice
June 13, 2019 at 4:04 am **Reply**

HI,

I would like to say thank you so much for creating such a great tutorial for sensor fusion algorithm. I am using MPU9250 as the sensor to get data. I am currently trying to implement your "Attitude Determination with Quaternion using Extended Kalman Filter" for my sensor.

While looking at your code I see some hard coded numbers used. So, back-tracked to your tutorials to see how you got them, but I am unclear in certain places. I would really appreciate if you could give me any help.

1> In readSensor_EKF, I assumed data[0:2] = magnetometer, data[3:5] = gyroscope and data[6:8] = accelerometer. Can you please tell me how you got the scaling factor that you used ?
For example, 0.00875 – gyro, 0.061 – mag, and 0.080 – accel. Because I thought, this is how you find the scale, for example, accel +/- 4g (the register map for

LSM6DS33 shows +/- 4g to be default and it is a 16 bit number): 8 / 2 ^ 16 = 0.000122.

2> Magnetometer Calibration, the script that you wrote that calculates the mag_Ainv and mag_b, needs a csv file called the magnetometer.csv. I read your article as well as looked into the csv file, the csv files has 9 rows, but doesnt it need only 3 rows for x,y and z? To understand the csv file, I looked into the "Python Data Collection Code". I tried to understand what it does, but I am not familiar with serial communication. Could you please elaborate what does the 9 rows represent? So, that I can modify the "Python Data Collection Code" for MPU9250?

3> in the "Python Magnetometer Calibration Code", you have: magX = data[:, 6] * 0.080. Should not it be agX = data[:, 6] * 0.00875? If the scale for mag is taken to be correct in the readSensor_EKF.py?

Thank you , again for your help.

SensorNovice
June 13, 2019 at 4:21 am **Reply**

Correction Q1 & 3: while carefully re-reading your code, I realized that the data elements were different than what I understood. So, Q3 is redundant as I misunderstood the sensor values. 1> In readSensor_EKF, I assumed data[0:2] = gyroscope, data[3:5] = accelerometer and data[6:8] = magnetometer .

But, could please explain how you got the conversion factor and Q2?

Thanks.

rikisenia.L
June 13, 2019 at 10:01 pm **Reply**

Hi SensorNovice,

You are right.
The first 3 values are the gyroscope reading, followed by accelerometer reading and finally magnetometer reading.
Do take a look at the arduino code too for the settings which I used for the sensors.

Conversion Factor:
For the gyroscope, if you take a look at the data sheet for L3GD20H, you'll find that there is a [Sensitivity] setting under the [Mechanical Characteristics] section. Since I used the 245 degrees per second [Measurement Range] setting, this corresponds to a sensitivity of 8.75 milli-degrees per second, per digit. This simply means that an increment of 1 in the raw sensor data is equal to an increment of 8.75 milli-degrees. As such, I had to multiply the raw data by 0.00875 to convert it into units of degrees.
Similarly, for the magnetometer, I used a +/- 2 gauss setting which gives a sensitivity of 0.080 milli-gauss per digit. Hence a factor of 0.080 was used on the raw data to convert it into units of milli-gauss.

Q2:
The 9 columns of data are 3 columns of gyroscope, accelerometer and magnetometer readings respectively. You are right that we only need 3 columns of magnetometer data for the calibration so you can just insert 0s in the first 6 columns. Otherwise, you can change the “Python Magnetometer Calibration Code” to adapt to your data format.
I was simply reusing the arduino data transfer code so I adapted the python side to match with the format. Apologies for the messiness. If you take a look at the arduino code, you might understand more about what is happening.

Hope this answers your questions!

SensorNovice
June 14, 2019 at 12:41 am **Reply**

Thank you, for such a detailed reply and the effort you put into this project.

Yang
September 20, 2019 at 1:50 pm **Reply**

Hi, rikisenia.L
I have a very simple question, your device can directly get the quaternion? I’m using a 6 DOF imu from st company which can directly get accelerator and gyroscope data from device. Right now,I’m trying to use kalman filter to remove the error and calculate the eular angle through the corrected data of gyro. then calculate the quaternion value through eular angle, and rotate the accelerate direction.
The thing is, I read many papers, you all get quaternion first then calculate eular angle. For me, the only data I have are accelerator and gyro. I don’t have either quaternion or eular angle directly. Is that a right way to apply my method?
Thank you and waiting for your reply.

rikisenia.L
September 21, 2019 at 3:43 pm **Reply**

Hi Yang,

I am also using just accelerometer and gyrometer data.
The quaternion can be calculated from angular velocities through equation 2.
Please take a look at this section (https://www.thepoorengineer.com/en/quaternion/#quaternion).
I think it will answer most of your questions.

francis

September 30, 2019 at 10:15 pm **Reply**

Hello, this is a super well documented material on the topic. First, let me thank you for creating such helpful tutorial.
I have also a question that i find difficult to understand:

I understood that the magnetometer gives a 3D vector that points to the mag. North. It is also OK, that acc. is used for vertical plane calculations, while mag. is used in the horizontal plane.

But i don't really get, why do you remove the z-axis of the transformed mag. data? You also mention, that even without this step the EKF works, but still, can you explain in detail why this step is necessary? Why should we remove wm3 (wm3=0)? What do we get by removing it?

Thanks in advance.

rikisenia.L

October 4, 2019 at 9:57 pm **Reply**

Hi Francis,

I am glad that this post helped you in some way.

The step of making wm3=0 is optional. The idea behind this is so that the magnetometer reference vector does not affect the direction of the gravity (vertical) reference vector. In this implementation, the accelerometer takes care of the vertical reference vector, and the magnetometer takes care of the horizontal vector, and they both should not affect each other. This is the only reason and you are totally right to say that "wont the results be better if we use more information?" If you have the time to test out both methods and compare their results, do update me on which one is better!

Anne

October 29, 2019 at 12:51 am **Reply**

Hello,

Thank you for the article, it was very usefull. I want to use only the accelerometer and gyrometer, do you know what do you know what do I need to change?

Thanks!

Fred

November 27, 2019 at 1:50 am **Reply**

really amazing job, thanks for sharing!!!

Deep Shah

December 4, 2019 at 3:30 pm **Reply**

I am using 6 axis imu(acc and gyro) to move object from A to B (horizontal) direction but it is not happening,,

what should i do?

Mayank

December 16, 2019 at 4:23 pm **Reply**

Thank you for the tutorial. I have implemented your code for the NXP-9DoF Precision sensor.
The roll and pitch values are not independent of each other but the yaw values are very stable and precise. I am not understanding why is the code behaving in that manner. And also, can you further explain how can we derive R and Q values from sensors' datasheets?

Varun Bachalli

April 15, 2020 at 2:47 am **Reply**

Hi thanks for your tutorial.

Are you sure you can divide the state space equations into A and B the way you have?

Don't you need to calculate the Jacobian for these matrices since it is nonlinear in the controls and states?

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Website

Notify me of follow-up comments by email.

Notify me of new posts by email.

POST COMMENT

This site uses Akismet to reduce spam. Learn how your comment data is processed.

Subscribe via Email!

Enter your email address to receive notifications of new posts by ThePoorEngineer through email.

Email Address

SUBSCRIBE

Categories

Select Category

Archives

September 2019 (3)

February 2019 (1)

January 2019 (1)