ATTITUDE DETERMINATION WITH QUATERNION USING EXTENDED KALMAN FILTER

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)

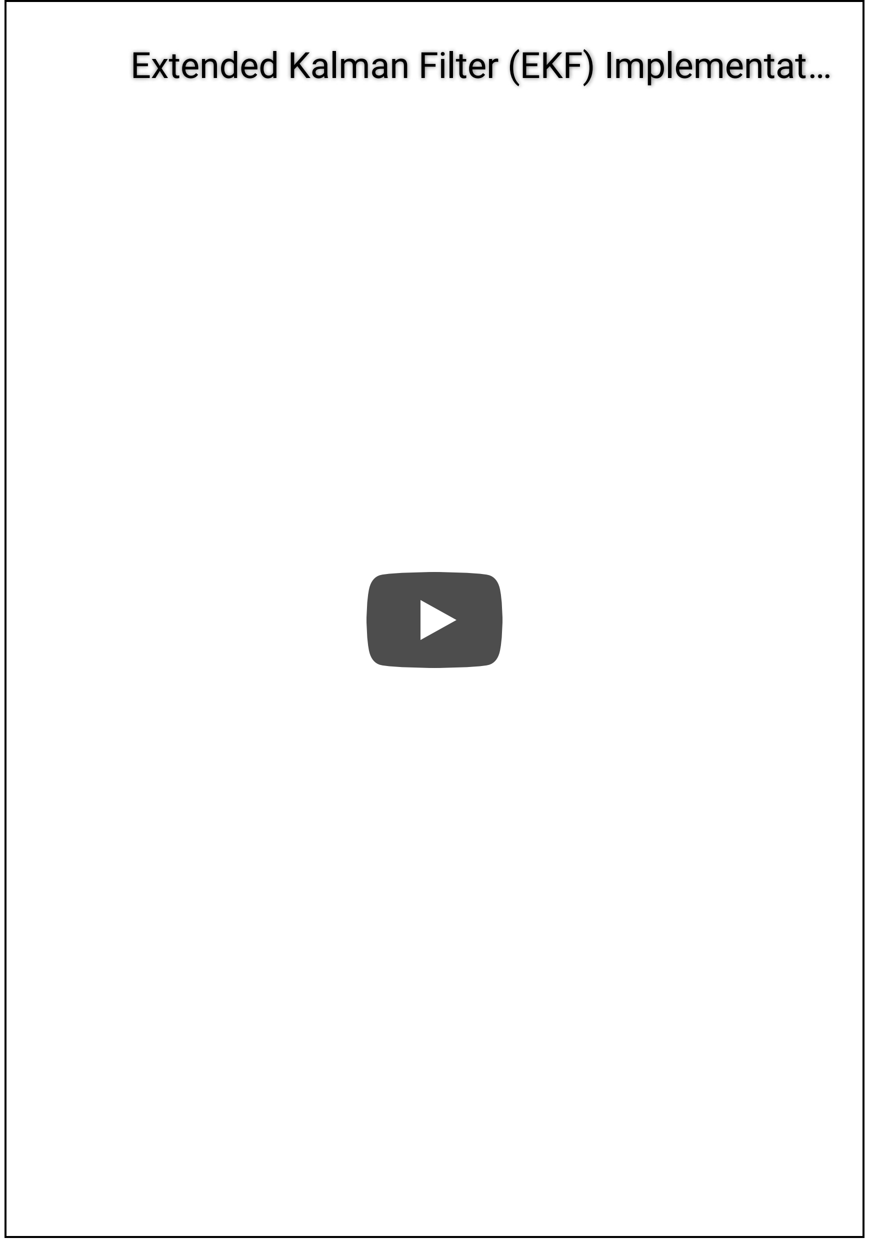
- Arduino Code
- Python Code (EKF implementation)

Categories

Attitude Determinat

Archives

September 2019 (3)
February 2019 (1)
January 2019 (1)
September 2018 (1)
July 2018 (4)
March 2018 (3)
February 2018 (5)
November 2017 (6)
July 2017 (2)
April 2017 (6)
March 2017 (4)
February 2017 (2)
January 2017 (9)
December 2016 (24)
November 2016 (1)



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

$$\dot{q} = \tfrac{1}{2}S(w)q = \tfrac{1}{2}S(q)w \quad ------- \quad (2)$$

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).

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

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,

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

Substituting equation (3) into equation (4),

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

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.

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

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

$$\begin{aligned} x_{k+1} &= \begin{bmatrix} I_{4\times4} & -\tfrac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_k x_k + \begin{bmatrix} \tfrac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_k w_k \\ \begin{bmatrix} q \\ b^g \end{bmatrix}_{k+1} &= \begin{bmatrix} I_{4\times4} & -\tfrac{T}{2}S(q) \\ 0_{3\times4} & I_{3\times3} \end{bmatrix}_k \begin{bmatrix} q \\ b^g \end{bmatrix}_k + \begin{bmatrix} \tfrac{T}{2}S(q) \\ 0_{3\times3} \end{bmatrix}_k w_k \\ & ------- \quad (7) \end{aligned}$$

*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.

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

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.

$$ {}^b R_w = {}^b R_w{}^T = R(q)^T = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}^T$$

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

——————- (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.

$$ \begin{bmatrix} {}^ba_x \\ {}^ba_y \\ {}^ba_z \end{bmatrix} = - \begin{bmatrix} 2(q_1q_3 - q_0q_2) \\ 2(q_2q_3+q_0q_1) \\ q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} + \begin{bmatrix} {}^bb_x \\ {}^bb_y \\ {}^bb_z \end{bmatrix} + \begin{bmatrix} {}^be_x \\ {}^be_y \\ {}^be_z \end{bmatrix} $$

——————- (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.

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

where

$({}^ba_m)_k$ 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), ${}^be_a + {}^b b_a$, 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.

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

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

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

$$ h_a(q_k)|_{q_k=q_{k-1}} = h_a(q_{k-1}) + h'_a(q_{k-1})(q_k - q_{k-1}) + \ldots $$

——————- (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).

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

———————— (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)

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

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,

$$r = \begin{bmatrix} {}^bm_1 \\ {}^bm_2 \\ {}^bm_3 \end{bmatrix}$$

$$r' = \begin{bmatrix} {}^wm_1 \\ {}^wm_2 \\ {}^wm_3 \end{bmatrix}$$

where

${}^bm_1, {}^bm_2, {}^bm_3$ is the calibrated magnetometer data in the body frame's x, y, z direction respectively.
${}^wm_1, {}^wm_2, {}^wm_3$ 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 ${}^wm_3 = 0$, 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).

$$^bm_m =^b R_w(^wm_r) +^b e_{mag} +^b b_{mag} \qquad \text{-------} \qquad (16)$$

where

^bm_m is the measured magnetic field in the body frame by the magnetometer
^bR_w is the rotation matrix for world frame to body frame
^wm_r is the reference magnetic North vector in the world frame, where in my case $^wm_r = \begin{bmatrix} 0 & -1 & 0 \end{bmatrix}^T$
$^be_{mag}$ is the magnetometer noise in the body frame
$^bb_{mag}$ is the magnetometer bias in the body frame

The rotation matrix ^bR_w is actually the same matrix as what was used in the accelerometer. Below is the exact same equation (9).

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

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

$$\begin{bmatrix} ^bm_x \\ ^bm_y \\ ^bm_z \end{bmatrix} = -\begin{bmatrix} 2(q_1q_2+q_0q_3) \\ q_0^2-q_1^2+q_2^2-q_3^2 \\ 2(q_2q_3-q_0q_1) \end{bmatrix} + \begin{bmatrix} ^bb_x \\ ^bb_y \\ ^bb_z \end{bmatrix} + \begin{bmatrix} ^be_x \\ ^be_y \\ ^be_z \end{bmatrix}$$
$$\text{-------} \qquad (17)$$

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 ^wm_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

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

$P_k^- = AP_{k-1}A^T + Q$ ——– (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

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

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

$P_k = (I - K_k C)P_k^-$ ——– (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

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

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

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))

```
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=
    tmp2 = np.concatenate((np.zeros((3, 4)), np.identity(3)), a
    self.A = np.concatenate((tmp1, tmp2), axis=0)
    self.B = np.concatenate((dt / 2 * Sq, np.zeros((3, 3))), ax
    self.xHatBar = np.matmul(self.A, self.xHat) + np.matmul(sel
    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.tra
```

Update

Equation 3k

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

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.

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

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

$\hat{x}_k = \hat{x}_k^- + K_k(y_k - C\hat{x}_k^-)$ ——– (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

$P_k = (I - K_kC)P_k^-$ ——– (5k)

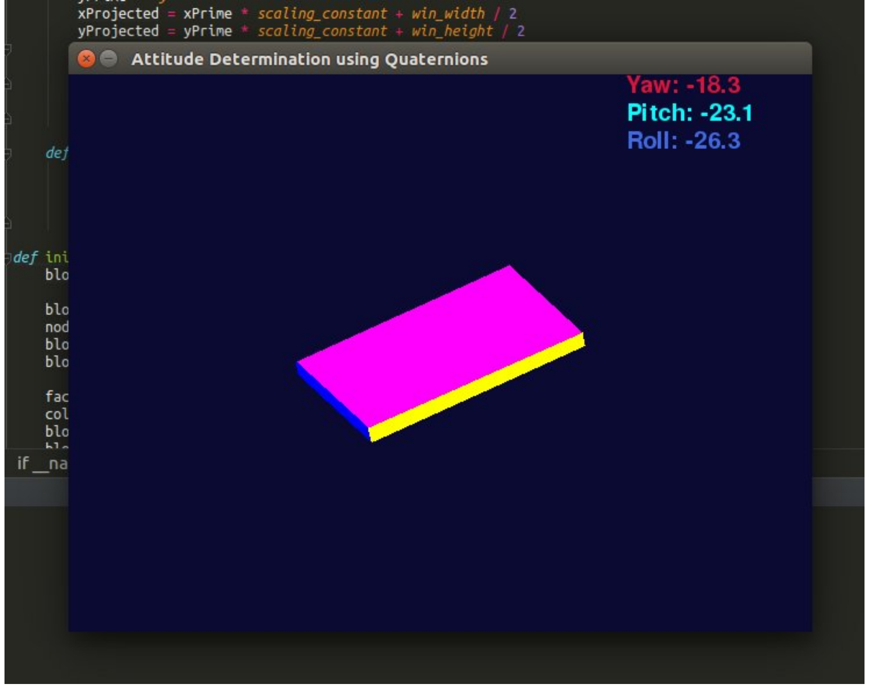
As for equation (5k), we already have all the variables ready so implementing it will be no problem at all. Equation (3k) to (5k) are all implemented within the `update()` method in the python code as shown below as well.

```
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 -
    self.xHat[0:4] = self.normalizeQuat(self.xHat[0:4])
    self.p = np.matmul(np.identity(7) - np.matmul(self.K, self.
```

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! (:



```
xProjected = xPrime * scaling_constant + win_width / 2
yProjected = yPrime * scaling_constant + win_height / 2
```

Attitude Determination using Quaternions

```
Yaw: -18.3
Pitch: -23.1
Roll: -26.3
```

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! (:

CALIBRATING THE MAGNETOMETER

Posted on July 28, 2018

Table of Contents

In order to remove the bias from the 3-axis gyroscope, we would require a reference for at least 2 of the axis for a fully defined solution. This can be visualized simply.First, position your fingers similar to that of Fleming's Right-hand Rule. The 3 fingers show the x (thumb), y (index), z (middle). You will realize that when you lock the direction of one finger in place, you could still rotate your hand in some way. However, if you lock the direction of any 2 fingers, you'll find that there is no way for you to rotate your hand anymore. This means that we will need at least 2 references for the axis to lock the 3 directional vectors in place.

In order to do so, we are going to use the accelerometer (gravity vector always points to the ground) and the magnetometer (Earth's magnetic field always points to the Magnetic North). In this post, I will talk about the calibration of the Magnetometer. Calibration of the accelerometer is not done because the raw data is already quite "clean".

What we want to get from the magnetometer is the North directional vector. However, there are all sorts of errors that influence the actual measurement thus there is a need to perform a calibration for the magnetometer before using it. The magnetometer module that I was using was the **LSM303D**. It came together with the **Pololu MinIMU-9 v3** which is now discontinued but the model is not really that important because the calibration method mentioned here should work for all magnetometers.

I got my study materials from a few journal papers, but this **website** was the most helpful because it actually explains the actually method concisely. In fact, what I am going to explain here will be almost similar to what is written in the **website** so you can just check out either one (I hope mine is easier to understand!).

Error Modelling

In this section, I will list down a few of the common errors found in a magnetometer together with its mathematical model.

Scale Factor

Alright, this is not really an error due to noise but we will require some form of scaling for the input in order to generate outputs that fall within our desired range. For example, we might want to normalize our 3 direction vectors to have a value between 0 to 1. This can be modeled mathematically with a diagonal matrix as

$$S = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

Misalignment Error

When setting up the sensors on module, the 3 axis will most probably be somewhat misaligned because we can never position it perfectly. This causes the measurement axis to be skewed such that it is not exactly orthogonal. This can be modeled via

$$N = \begin{bmatrix} n_{x,x} & n_{y,x} & n_{z,x} \\ n_{x,y} & n_{y,y} & n_{z,y} \\ n_{x,z} & n_{y,z} & n_{z,z} \end{bmatrix}$$

where each column represents the orientation of the respective axis with respect to the orthogonal axis.

Bias Error

The bias error is the error that shifts the measurement values away from the actual value by a fixed amount. If the mean of the actual measurement is 100, then having a bias of 10 would mean that the mean of the measurement reading will be 110. This can be modeled by a simple offset vector as shown below.

$$b_m = \begin{bmatrix} b_{m,x} \\ b_{m,y} \\ b_{m,z} \end{bmatrix}$$

Hard Iron Error

Hard iron errors appear due to the presence of permanent magnets and remanence of magnetized iron. The effect of hard iron errors is the same as having a bias (think of it as the permanent magnet pulling the North direction vector towards it) so the error model is similar as well.

$$b_{hi} = \begin{bmatrix} b_{hi,x} \\ b_{hi,y} \\ b_{hi,z} \end{bmatrix}$$

Soft Iron Error

Soft iron errors appear due to the presence of material that influences or distorts a magnetic field, but does not necessarily generate a magnetic field itself. The presence of iron and nickel for example, will generate a distortion in the measured magnetic field. While hard-iron distortion is constant regardless of orientation, the distortion produced by soft-iron materials is dependent upon the orientation of the material relative to the sensor and the magnetic field. Thus, in order to model this error, a more complicated matrix is required such as shown below.

$$A_{si} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

where each column represents the orientation and scale of the respective axis with respect to the orthogonal axis.

Measurement Model

Using the mathematical model of the above errors, we can then summarize the measurement model as such:

$$h_m = SN(A_{si}h + b_{hi}) + b_m$$

where

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

We can then combine some of the errors together to get a more simplified form:

$$h_m = Ah + b$$

where

$A = SNA_{si}$

$b = SNb_{hi} + b_m$

Calibration

With the above measurement model, we can now manipulate the equation to make h the subject instead.

$$h = A^{-1}(h_m - b) \quad \text{———————-} \quad (1)$$

where

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

Therefore, if we know the values of the A^{-1} and b matrix, we will be able to solve for the actual magnetic field. As such, this section will focus on how to estimate the values of the A^{-1} and b matrix using measurement data.

Firstly, there is actually another set of criteria that we have. That is, we want to set the magnitude of the actual magnetic field to be 1. In mathematical equation, that is

$$h^T h = 1 \quad ———————- \quad (2)$$

$$\begin{bmatrix} h_x & h_y & h_z \end{bmatrix} \begin{bmatrix} h_x \\ h_y \\ h_z \end{bmatrix} = 1$$

If we substitute equation (1) into equation (2), we will get the following equation.

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

Let $Q = (A^{-1})^T A^{-1}$. Therefore, Q must be a symmetric matrix since it is the product of a matrix multiplied by its transpose.

$$(h_m - b)^T Q (h_m - b) = 1$$

Expanding the above equation and simplifying,

$$h_m{}^T Q h_m – h_m{}^T Q b – b^T Q h_m + b^T Q b – 1 = 0 \quad ———————- \quad (3)$$

since Q is symmetric, $b^T Q h = h^T Q b$. Therefore, we can futher simplify equation (3) to

$$h_m{}^T Q h_m – 2 h_m{}^T Q b + b^T Q b – 1 = 0 \quad ———————- \quad (4)$$

Equation (4) can now be rewritten in **quadric form** (a quadric is a generalization of conic sections (ellipses, parabolas, and hyperbolas))

$$h_m{}^T Q h_m + h_m{}^T n + d = 0 \quad ———————- \quad (5)$$

where

$Q = (A^{-1})^T A^{-1}$

$n = -2Qb$

$d = b^T Q b – 1$

There are many different types of quadric surfaces and they can all be verified mathematically. However, we are not going to do that here because there is a more obvious answer. h is supposed to be a unit direction vector so it would only be natural that the set of all directions would form a unit circle. In other words, Equation (5) must be that of an ellipse (a circle is a special case of an ellipse). On a side note, you can find a list of all quadric surfaces and their definition from **this site**.

Anyway, what we need to do now is to collect some data so that we can use it for our calibration. In order to collect the data, we need the Arduino to send it to the computer, and Python on the computer to save the data (you can always use any other programs to achieve this). The Arduino program is exactly the same as the one in the **previous section** but I'll provide it below just in case. The python program simply collects the data for a period of time and saves the file in a csv format (Please don't mind the class name because I was too lazy to change it from my previous project). I will also provide the data that I have collected so that you can replicate the graph that I will show later.

- Arduino Code

- Python Data Collection Code
- Magnetometer Data

Alright, go ahead and collect your data if you happen to use the same sensor as me. Otherwise, you can still use part of the code for the communication between Arduino and Python.

Right now, if we were to simply plot our raw data, the data points will most probably appear on the surface of an ellipse which has a center that has an offset from the origin. However, after our calibration, if we do a scatter plot of our data taken in all directions, we would expect that the points will now lie close to the surface of a unit circle. That is the main objective of this calibration process. Before I begin, here is the calibration code. The algorithm is based on this **journal paper** (Least squares ellipsoid fitting) and this **journal paper** (scaling the solution).

- Python Magnetometer Calibration Code

Let's take a look at the raw data that I have gotten from the sensor. I must say that I did not rotate the magnetometer in all directions while collecting the data thus there are blank spaces in the graph as shown below. You would possible get better results if you were to capture the data more thoroughly.

UNCALIBRATED MAGNETOMETER DATA

If you take a careful look at the above graph, you'll see a small blue dot at (0, 0, 0). That is actually a unit circle centered at the origin so we want our data to be scaled down such that it lies on that blue circle. You should try to run the program on your computer because then you'll be able to rotate the 3D graph to get a more complete view. Anyway, I am not going to go through the least squares fitting algorithm from the **first paper** (because I don't really understand the mathematics behind it as well..) but after applying the fitting algorithm, we will be able to solve for the values of Q, n and d from equation (5).

Next, we will have to determine the values of A^{-1} and b so that we can apply equation (1) to perform our calibration. This is done via the following equations based on the **second paper**.

$$b = -Q^{-1}n$$

$$A^{-1} = \frac{1}{\sqrt{n^T M^{-1} n - d}} M^{1/2}$$

Now that we have the required variables to use equation (1), we are finally able to perform the calibration method. What we have to do now is simply apply equation (1) to all measurement points! For my data set, I was able to get the following graph.

CALIBRATED MAGNETOMETER DATA

And we are done! The magnetometer is now ready for use! On a side note, I do feel that there is still room for improvement in the above calibration process. For my magnetometer, the direction vector is slightly skewed to a particular direction, meaning that an actual 90 degrees show up as 80 degrees and a -90 degrees show up as -80 degrees. I suppose there are ways to correct these but I have not yet found the time to look things up. For now, let us use the above method and proceed to the next section!

Go to Next Section

QUATERNION ROTATION

Posted on July 10, 2018

Table of Contents

In this post, I will go through some of the quaternion basics, and provide a simple implementation in python to visual the rotations. However, we will not be doing any derivations from first principles as it is quite mathematical. Instead, we will use existing formulae to build our code.

Quaternion basics

Quaternion provides us with a way for rotating a point around a specified axis by a specified angle. If you are just starting out in the topic of 3d rotations, you will often hear people saying “use quaternion because it will have any gimbal lock problems”. This is true, but the same applies to rotation matrices well. Rotation matrices do not experience gimbal lock problems. In fact, it does not make sense to say that at all. The gimbal lock problem happens when you use Euler Angles, which are simply a set of 3 elemental rotations to allow you to describe any orientation in a 3D space. In attitude determination, we often visualize a 3D rotation as a combination of yaw, pitch and roll. These are Euler angles thus they are susceptible to the gimbal lock problem, regardless of whether you use quaternion or not.

A quaternion is named as such because there are 4 components in total. If q is a quaternion, then

$$q = q_0 + q_1\,\tilde{\imath} + q_2\tilde{\jmath} + q_3\tilde{k}$$

You can think of quaternion as an extension of complex number where instead of 1 real and 1 imaginary number, you now have 1 real and 3 imaginary numbers. Another way of notating a quaternion is a as such:

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

However, note that some authors write the imaginary part first before the real part. For example,

$$q = q_0\,\tilde{\imath} + q_1\tilde{\jmath} + q_2\tilde{k} + q_3 = \begin{bmatrix} \tilde{\mathbf{q}} \\ q_3 \end{bmatrix}$$

You can usually tell which ordering they use by checking which of the variables are bolded (or italicized). The marked variable is usually the imaginary component and the unmarked one (usually with a number subscript) is the real part. For this tutorial, we will be using the first definition where the real part comes first.

The conjugate of a quaternion, $q = q_0 + q_1\,\tilde{\imath} + q_2\tilde{\jmath} + q_3\tilde{k}$, is defined as follows.

$$q^* = q_0 - q_1\,\tilde{\imath} - q_2\tilde{\jmath} - q_3\tilde{k}$$

Don’t worry about what it is used for yet because you’ll find out soon below.

Nerxt, the magnitude of a quaternion is defined in a similar fashion to vectors.

$$|q| = q_0^2 + q_1^2 + q_2^2 + q_3^2$$

Therefore, a unit quaternion can then be defined in the following manner.

$$Uq = \frac{q}{|q|} = \frac{q_0}{|q|} + \frac{q_1}{|q|}\,\tilde{\imath} + \frac{q_2}{|q|}\,\tilde{\jmath} + \frac{q_3}{|q|}\,\tilde{k}$$

where Uq refers to a unit quaternion.

Moving on, below is the definition for quaternion addition and subtraction (simply replace the “+” with “-“)

$$p + q = \begin{bmatrix} p_0 + q_0 \\ p_1 + q_1 \\ p_2 + q_2 \\ p_3 + q_3 \end{bmatrix}$$

Next comes the definitions of the imaginary part

$$\tilde{\imath}\,\tilde{\imath} = \tilde{\jmath}\tilde{\jmath} = \tilde{k}\tilde{k} = -1$$

$$\tilde{\imath}\,\tilde{\jmath} = \tilde{k} = -\tilde{\jmath}\,\tilde{\imath}$$

$$\tilde{\jmath}\tilde{k} = \tilde{\imath} = -\tilde{k}\tilde{\jmath}$$

$$\tilde{k}\tilde{\imath} = \tilde{\jmath} = -\tilde{\imath}\tilde{k}$$

With this, you will then be able to do quaternion multiplications . This is done the same way as when you multiply a 4 variable algebra by another 4 variable algebra. The derivation gets quite long so I am going to skip it and just post the answer.

$$q \otimes p = \begin{bmatrix} q_0p_0 - q_1p_1 - q_2p_2 - q_3p_3 \\ q_1p_0 + q_0p_1 - q_3p_2 + q_2p_3 \\ q_2p_0 + q_3p_1 + q_0p_2 - q_1p_3 \\ q_3p_0 - q_2p_1 + q_1p_2 + q_0p_3 \end{bmatrix}$$

The above multiplication can also be written in the form of a matrix multiplication as follows:

$$q \otimes p = Qp = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \\ q_1 & q_0 & -q_3 & q_2 \\ q_2 & q_3 & q_0 & -q_1 \\ q_3 & -q_2 & q_1 & q_0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix}$$

Now, we are finally ready to talk about rotations. If we define a quaternion in the following manner:

$$q = \begin{bmatrix} cos(\theta/2) \\ \tilde{\mathbf{u}} sin(\theta/2) \end{bmatrix}$$

Then,

$$r' = q \otimes r \otimes q^*$$

refers to a rotation of the vector r, θ degrees about the vector $\tilde{\mathbf{u}}$. r' is thus the rotated vector. The above can once again be written as a matrix multiplication instead of a quaternion multiplication. The math is tedious so I am just going to post the result once again. You can always try deriving it yourself if you don't believe (: Anyway, the matrix equivalent of the above is

$$r' = Cr \quad \text{————––} \quad (1)$$

where

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

This matrix form is important because it allows us to make a comparison with the rotation matrix derived from Euler Angles in order to determine the attitude (yaw, pitch, roll) of the object.

Last is the most exciting equation of all.

$$\dot{q} = \tfrac{1}{2} q \otimes w$$

where

$$w = 0 + w_1\tilde{i} + w_2\tilde{j} + w_3\tilde{k}$$

w_1, w_2, w_3 is the angular velocity in the x, y and z direction respectively. This equation gives us a way to use the values directly from our gyrometer to transform it into a rotation! Once again, it is more intuitive to work in matrix so we are going to convert the above into matrix from. Notice that the same equation can be expressed in 2 different ways here.

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

where

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

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

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

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

Now, let us test out this method of rotation with the Pygame rectangular block that we created in the **previous section**.

Testing Quaternion Rotation in Pygame

Here are the sample code for this section. It is built based on the previous post so if you are looking to understand the code, you should read up the **previous post**.

- Python Simple Quaternion Rotation Code

The BoardDisplay code references the Wireframe code, and the Wireframe code references the Quaternion code. In order to let the Pycharm know where it can find all the relevant files, you will need to mark the folder containing the all the files as the sources root. You can do this by right clicking the folder in Pycharm, the select “Mark Directory as”, then “sources root”. If you do this right, the folder should be marked blue in Pycharm.

The aim of this section is to convert angular velocity into a rotation of the object. To test things out, we are going to use a constant angular velocity as an input, and see how an object will rotate in Pygame. We can use equation (1) and (2) from above to achieve this but they are in continuous time form so we have to discretize it first.

In order to discretize equation (2), we are going to take the first order Taylor Series Transformation. This can be done rather simply as shown:

$$\dot{q} = \tfrac{1}{2}S(w)q$$

$$(q_{k+1} - q_k)/dt = \tfrac{1}{2}S(w_k)q_k$$

$$q_{k+1} = \tfrac{dt}{2}S(w_k)q_k + q_k$$

where dt is the time between sample $k+1$ and k.

The same can be applied to the other form in equation (2) as well. Doing so will yield this result:

$$q_{k+1} = \tfrac{dt}{2}S(q_k)w_k + q_k$$

You can see that in the Quaternion code, this is actually the equation that I was implementing in the rotate method.

```
class Quaternion:
    def __init__(self):
        self.q = np.array([1, 0, 0, 0])  # Initial state of the

    def rotate(self, w, dt):
        q = self.q
        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]]])
        self.q = np.matmul(dt/2 * Sq, np.array(w).transpose())
```

Now I had some problems understanding what q actually represented because sometimes they can take on the form of a vector (scalar part is 0). In here, q represents the quaternion that maps the original orientation of the object with the current orientation of the object. As such, when I initialized q as [1, 0, 0, 0], I was actually implying that the starting orientation of the object is the orientation of the object in Pygame. We don't really have a physical object to test with right now so it sounds a little weird but once we get to the next section, it should become clearer. It just means that the object in real life is in the same orientation as the object shown in Pygame at time 0.

Determining the Euler Angles

Next, we need a way for us to determine the Euler Angles because they are more intuitive to deal with. Equation (1) provides us with a rotation matrix using quaternion, but we can actually determine the same rotation matrix using Euler angles as well (except for the singularity points encountered during Gimbal Lock). We are going to use the Yaw, Pitch, Roll convention so take note that if you use any other order, you will get a different result.

A positive yaw is defined as counter-clockwise rotation about the z-axis. The rotation matrix for a yaw angle of α is:

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A positive pitch is defined as counter-clockwise rotation about the y-axis.The rotation matrix for a pitch angle of β is:

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix}$$

A positive roll is defined as counter-clockwise rotation about the x-axis.The rotation matrix for a roll angle of γ is:

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix}$$

Any rotations can be decomposed into these 3 elemental rotations so a roll, pitch, yaw rotation (note the order of rotation here) can be described by the following matrix:

$$\begin{aligned} R(\alpha,\beta,\gamma) &= R_z(\alpha)R_y(\beta)R_x(\gamma) \\ &= \begin{bmatrix} \cos(\alpha)\cos(\beta) & \cos(\alpha)\sin(\beta)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\sin(\beta)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ \sin(\alpha)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -\sin(\beta) & \cos(\beta)\sin(\gamma) & \cos(\beta)\cos(\gamma) \end{bmatrix} \end{aligned}$$

Take note that the above equation only applies for a rotation that performs the roll first, the pitch, then yaw last. If the order is changed, you will end up with a different rotation matrix. Now, if the quaternion rotation matrix is correct, which of course it is since it has been used by so many people around the world, then we can use the above roll, pitch, yaw rotation matrix to get the corresponding Euler angles from the quaternion rotation matrix.

From equation (1), lets give each cell in matrix C an index:

$$C = \begin{bmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{bmatrix}$$

It is then possible to solve for the roll, pitch, yaw angles!

$$\begin{aligned} C_{20} &= -\sin(\beta) \\ \beta &= \sin^{-1}(-C_{20}) \qquad -------(3) \end{aligned}$$

Thus pitch is given by the above equation (3).

CASE 1

If $\cos(\beta) \neq 0$,

$$\begin{aligned} \frac{C_{10}}{C_{00}} &= \frac{\sin(\alpha)\cos(\beta)}{\cos(\alpha)\cos(\beta)} = \tan(\alpha) \\ \alpha &= \tan^{-1}\left(\frac{C_{10}}{C_{00}}\right) \qquad -------(4) \end{aligned}$$

$$\begin{aligned} \frac{C_{21}}{C_{22}} &= \frac{\cos(\beta)\sin(\gamma)}{\cos(\beta)\cos(\gamma)} = \tan(\gamma) \\ \gamma &= \tan^{-1}\left(\frac{C_{21}}{C_{22}}\right) \qquad -------(5) \end{aligned}$$

Thus the yaw angle is given by the equation (4), and the roll angle is given by equation (5).

$$\begin{aligned} yaw &= \tan^{-1}\left(\frac{C_{10}}{C_{00}}\right) \\ pitch &= \sin^{-1}(-C_{20}) \\ roll &= \tan^{-1}\left(\frac{C_{21}}{C_{22}}\right) \end{aligned}$$

However, if $\cos(\beta) = 0$, the problem of Gimbal lock occurs. With our coordinate system, this occurs when the pitch angle is 90 or -90 degrees.

Case 2

For the case of a pitch angle of 90 degrees, the rotation matrix simplifies to:

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & \cos(\alpha)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & \cos(\alpha)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ 0 & \sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & \sin(\alpha)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ -1 & 0 & 0 \end{bmatrix}$$

Apply some trigonometric identities and we can get the following:

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & \sin(\gamma-\alpha) & \cos(\alpha-\gamma) \\ 0 & \cos(\alpha-\gamma) & \sin(\alpha-\gamma) \\ -1 & 0 & 0 \end{bmatrix}$$

In this situation, both yaw and roll refers to the same rotation so we usually assign one of it to be 0, and calculate the other. For me, I assigned yaw to be 0, and calculated roll but you can always do it the other way round. Therefore,

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & \sin(\gamma) & \cos(-\gamma) \\ 0 & \cos(-\gamma) & \sin(-\gamma) \\ -1 & 0 & 0 \end{bmatrix}$$

Now we can determine the roll angle with:

$$roll = \tan^{-1}\left(\frac{C_{01}}{C_{02}}\right)$$

If you are asking why did I use $C_{01} and C_{02}$ when it is possible to get the answer with just any one of the cells, it is because I wanted to rely on the arctan2 function which automatically finds the quadrant for me.

CASE 3

For the case of a pitch angle of -90 degrees, the rotation matrix simplifies to:

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & -\cos(\alpha)\sin(\gamma) - \sin(\alpha)\cos(\gamma) & -\cos(\alpha)\cos(\gamma) + \sin(\alpha)\sin(\gamma) \\ 0 & -\sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\gamma) - \cos(\alpha)\sin(\gamma) \\ 1 & 0 & 0 \end{bmatrix}$$

Once again, simplifying with trigonometric identities,

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & -\sin(\alpha+\gamma) & -\cos(\alpha+\gamma) \\ 0 & -\cos(\alpha+\gamma) & -\sin(\alpha+\gamma) \\ 1 & 0 & 0 \end{bmatrix}$$

I assigned yaw to be 0 so the above equation further simplifies to:

$$R(\alpha,\beta,\gamma) = \begin{bmatrix} 0 & -\sin(\gamma) & -\cos(\gamma) \\ 0 & -\cos(\gamma) & -\sin(\gamma) \\ 1 & 0 & 0 \end{bmatrix}$$

Now we can determine the roll angle with:

$$roll = \tan^{-1}\left(\frac{-C_{01}}{-C_{02}}\right)$$

And finally we are done! All the above can be written in python code as follows:

```
def getRotMat(q):
    c00 = q[0] ** 2 + q[1] ** 2 - q[2] ** 2 - q[3] ** 2
    c01 = 2 * (q[1] * q[2] - q[0] * q[3])
    c02 = 2 * (q[1] * q[3] + q[0] * q[2])
    c10 = 2 * (q[1] * q[2] + q[0] * q[3])
    c11 = q[0] ** 2 - q[1] ** 2 + q[2] ** 2 - q[3] ** 2
    c12 = 2 * (q[2] * q[3] - q[0] * q[1])
    c20 = 2 * (q[1] * q[3] - q[0] * q[2])
    c21 = 2 * (q[2] * q[3] + q[0] * q[1])
    c22 = q[0] ** 2 - q[1] ** 2 - q[2] ** 2 + q[3] ** 2

    rotMat = np.array([[c00, c01, c02], [c10, c11, c12], [c20, 
    return rotMat

def getEulerAngles(q):
    m = getRotMat(q)
    test = -m[2, 0]
    if test > 0.99999:
        yaw = 0
        pitch = np.pi / 2
        roll = np.arctan2(m[0, 1], m[0, 2])
    elif test < -0.99999:
        yaw = 0
        pitch = -np.pi / 2
        roll = np.arctan2(-m[0, 1], -m[0, 2])
    else:
        yaw = np.arctan2(m[1, 0], m[0, 0])
        pitch = np.arcsin(-m[2, 0])
        roll = np.arctan2(m[2, 1], m[2, 2])

    yaw = rad2deg(yaw)
    pitch = rad2deg(pitch)
    roll = rad2deg(roll)

    return yaw, pitch, roll
```

Alright, what's left is the interface between the Quaternion class and the actual code which is not that interesting so I am going to skip it. If you are wondering about the convertToComputerFrame method, it is actually for the next part and I was just lazy to erase it. Basically, the axis of the physical sensor is in a different orientation from the axis in Pygame so a conversion is required such that the rotation of the sensor can be reflected correctly in Pygame. If you have any other questions, feel free to write in the comments section below!

Below is how the program should look like when it runs.

Implementing Rotations with MEMS Gyrometer Data

We are now finally ready to interact with our Gyro sensor. I used the Pololu MinIMU-9 v3 sensor for the gyro (which is discontinued), and the Arduino UNO to read the data from the sensor and pass it to the computer. Here are the source code for the relevant components.

- Arduino Code
- Python Code

The “AccelGyro” folder in the Arduino Code is actually a library for the pololu MinIMU-9 v3 sensor so you need to place it in your Arduino’s libraries folder for it to work. If you have the exact same model, you can use the code as it is. If your sensor is of another model, or you are using another type of micro-controller, all you have to do is follow the format in which the data is sent to the computer and the Python Code should still work fine. Here’s the format which I used:

```
sendToPC(&gyroData.X, &gyroData.Y, &gyroData.Z,
         &accelData.X, &accelData.Y, &accelData.Z,
         &magData.X, &magData.Y, &magData.Z);

-------------------------------------------------------------

void sendToPC(int* data1, int* data2, int* data3,
              int* data4, int* data5, int* data6,
              int* data7, int* data8, int* data9)
{
  byte* byteData1 = (byte*)(data1);
  byte* byteData2 = (byte*)(data2);
  byte* byteData3 = (byte*)(data3);
  byte* byteData4 = (byte*)(data4);
  byte* byteData5 = (byte*)(data5);
  byte* byteData6 = (byte*)(data6);
  byte* byteData7 = (byte*)(data7);
  byte* byteData8 = (byte*)(data8);
  byte* byteData9 = (byte*)(data9);
  byte buf[18] = {byteData1[0], byteData1[1],
                 byteData2[0], byteData2[1],
                 byteData3[0], byteData3[1],
                 byteData4[0], byteData4[1],
                 byteData5[0], byteData5[1],
                 byteData6[0], byteData6[1],
                 byteData7[0], byteData7[1],
                 byteData8[0], byteData8[1],
                 byteData9[0], byteData9[1]};
  Serial.write(buf, 18);
}
```

Basically, the data is sent in the following order: gyro_X, gyro_Y, gyro_Z, accelerometer_X, accelerometer_Y, accelerometer_Z, magnetometer_X, magnetometer_Y, magnetometer_Z, and each variable is a 2 byte integer.

For the Python Code, you have to mark the directory containing all the files as "Sources Root" so that the IDE knows where to find the files. You can do this by right clicking the folder in Pycharm, the select "Mark Directory as", then "sources root". If you do this right, the folder should be marked blue in Pycharm. The readSensor_naive is an adaptation from one of my previous project so if you want to understand it, please take a look at **this post.** I did change a few of the parameters (and the parameter names too) but it should still be roughly the same. If you have any questions about it, don't hesitate to write in the comments section below!

One last point though, in the getSerialData method for the python code, you will see some random constants as from the extract below.

```
def getSerialData(self):
    privateData = self.rawData[:]
    for i in range(self.numParams):
        data = privateData[(i*self.dataNumBytes):(self.dataNumBy
        value,  = struct.unpack(self.dataType, data)
        if i == 0:
            value = ((value * 0.00875) - 0.464874541896) / 180.
        elif i == 1:
            value = ((value * 0.00875) - 9.04805461852) / 180.0
        elif i == 2:
            value = ((value * 0.00875) + 0.23642053973) / 180.0
        elif i == 3:
            value = (value * 0.061) - 48.9882695319
        elif i == 4:
            value = (value * 0.061) - 58.9882695319
        elif i == 5:
            value = (value * 0.061) - 75.9732905214
        elif i == 6:
            value = value * 0.080
        elif i == 7:
            value = value * 0.080
        elif i == 8:
            value = value * 0.080
        self.data[i] = value
    return self.data
```

These are actually the sensor bias value which I have determined through collecting the data while the sensor is stationary. It will most likely be different with your sensor so be sure to calibrate it first before using. If you look carefully at the readSensor_naive.py code, there is actually code for you to export the sensor readings to CSV. You can use those to get a list of values of sensor output to determine its bias value.

On a side note, the reason why I labeled the code as "naive" is because it does not take into account the noise of the gyrometer. This causes the calculated orientation to gradually drift away with no means of identifying the actually orientation. It'll be easier to understand if you watch the demo video below.

*does anyone have an easy solution to the graphics problem when using painter's algorithm with perspective view? If you switch the projection method in BoardDisplay.py to `projectOnePointPerspective`, you will see that the painter's algorithm does not work correctly. I suspect that it is due to the depth calculation but I have no idea how to go about correcting it. Will appreciate any help that I can get!

Go to Next Section

PREPARING THE GRAPHICS

Posted on July 10, 2018

Table of Contents

In this section, we are going to create the visualization tool so as to understand the problems and result more intuitively. Actually, it's just to make things look a little cooler. I mean, who would like to be looking at numbers when you can have an animation! Here are the full source code for this section for your reference:

1. Python BoardDisplay Code
2. Python Wireframe Code

Oh, before I forget, the BoardDisplay code references the Wireframe code so you will need to mark the folder containing the Wireframe.py as the source root. You can do this by right clicking the folder in Pycharm, the select "Mark Directory as", then "sources root". If you do this right, the folder should be marked blue.

Creating the Wireframe

In order to display an object in Pygame, we will first need to fully define its coordinates. To make things simple, I am just going to display a rectangular block to mimic the board which I am using so all we need to do is to define the 6 faces of the block. We are going to do this in 3 steps. First, we will have a class called "Node" to define the xyz coordinates of a point, then we will have another class called "Face" which takes in 4 "Nodes" to define a face of the rectangular block. We will then have 1 last class called "Wireframe" which stored the information of the cube, and also acts as an interface between the Pygame displaying code, and the back-end calculation code.

The back-end calculation part will come in the 3rd section when we start talking about quaternions so for now, we will only concentrate on displaying the block in Pygame. Here's the code for the 3 classes stated above.

```
# Node stores each point of the block
class Node:
    def __init__(self, coordinates, color):
        self.x = coordinates[0]
        self.y = coordinates[1]
        self.z = coordinates[2]
        self.color = color

# Face stores 4 nodes that make up a face of the block
class Face:
    def __init__(self, nodes, color):
        self.nodeIndexes = nodes
        self.color = color

# Wireframe stores the details of the block
class Wireframe:
    def __init__(self):
        self.nodes = []
        self.edges = []
        self.faces = []

    def addNodes(self, nodeList, colorList):
        for node, color in zip(nodeList, colorList):
            self.nodes.append(Node(node, color))

    def addFaces(self, faceList, colorList):
        for indexes, color in zip(faceList, colorList):
            self.faces.append(Face(indexes, color))

    def outputNodes(self):
        print("\n --- Nodes --- ")
        for i, node in enumerate(self.nodes):
            print(" %d: (%.2f, %.2f, %.2f) \t Color: (%d, %d, %
                 (i, node.x, node.y, node.z, node.color[0], node

    def outputFaces(self):
        print("\n --- Faces --- ")
        for i, face in enumerate(self.faces):
            print("Face %d:" % i)
            print("Color: (%d, %d, %d)" % (face.color[0], face.
            for nodeIndex in face.nodeIndexes:
                print("\tNode %d" % nodeIndex)
```

I have also created some convenience functions for logging purposes. This helps to ensure that we inserted the information correctly into the class. We will be using this class in the main "BoardDisplay" code. Here is an example of how you can use the class to initialize parameters for the block.

```
def initializeBlock():
    block = wf.Wireframe()

    block_nodes = [(x, y, z) for x in (-1.5, 1.5) for y in (-1,
    node_colors = [(255, 255, 255)] * len(block_nodes)
    block.addNodes(block_nodes, node_colors)
    block.outputNodes()

    faces = [(0, 2, 6, 4), (0, 1, 3, 2), (1, 3, 7, 5), (4, 5, 7,
    colors = [(255, 0, 255), (255, 0, 0), (0, 255, 0), (0, 0, 25
    block.addFaces(faces, colors)
    block.outputFaces()

    return block
```

If you run the above code, you should get a print out of all the nodes and faces that you have created. If you used exactly what I have written above, this is the output that you should get:

```
— Nodes —
0: (-1.50, -1.00, -0.10) Color: (255, 255, 255)
1: (-1.50, -1.00, 0.10) Color: (255, 255, 255)
2: (-1.50, 1.00, -0.10) Color: (255, 255, 255)
3: (-1.50, 1.00, 0.10) Color: (255, 255, 255)
4: (1.50, -1.00, -0.10) Color: (255, 255, 255)
5: (1.50, -1.00, 0.10) Color: (255, 255, 255)
6: (1.50, 1.00, -0.10) Color: (255, 255, 255)
7: (1.50, 1.00, 0.10) Color: (255, 255, 255)

— Faces —
Face 0:
Color: (255, 0, 255)
Node 0
Node 2
Node 6
Node 4
Face 1:
…
…
…
```

I only copied the print out for Face 0 but you should get print outs of up till Face 5 since we inserted information for 6 faces in total. Let us first begin with the Nodes. We have inserted information for 8 nodes, and as you can see in the print out, they are being indexed from 0 to 7. The color information is redundant for this project because we will be showing the colors of each face instead of the color for the nodes. However, the index of the Nodes are very important because the face object refers to this information. For example, I specified Face 0 to be the face that is enclosed by Node 0, 2, 6 and 4. Below is a diagram which shows the ordering of indexes that I used. The center of the block is located at the origin of the axes.

WIREFRAME NODE INDEX

With this, we have now a fully defined block. The next step is to create the view that you see on top in Pygame.

Creating a Perspective View

Your computer screen shows a 2 dimensional image. There should be no depth because it is just a flat piece of screen. So why does the diagram in the previous section look like a 3D figure to our eyes? This is actually due to the lines that make up the object (and also the axis) which gives an illusion of depth. **Here** is an article on the topic of depth perception which I though was really interesting so feel free to read through it if you are interested. Creating "depth" on the computer is not that difficult but if you do not draw the object with a perspective view, for example an orthographic view, it will look distorted to our eyes even though it is actually correct.

In this section, we will only talk about a simple implementation of the one vanishing point perspective view. You can look up **wikipedia** for more details on other perspective views. Below is the diagram of the perspective view that I am going to create for my viewer.

ONE VANISHING POINT PERSPECTIVE VIEW

Firstly, I am going to make the origin the vanishing point, so I will have to shift the object out of the origin. In order to do this, I will shift the center of the object by "P" in the z direction so the new center of the object will now be at (0, 0, P) as shown in the diagram. Next, I am going to put my "screen" at a distance "S" on the z-axis. This screen is the Pygame screen which is 2 dimensional. What we need to do now is to project all the 8 points on the cube onto the screen. If you look closely at the projection, you'll realize that this is simply a linear scaling of the x and y coordinates based on the distance away from the screen.

Let us take Node 0 with coordinates (-1.50, -1.00, -0.10) as a example. When we shift our object out of the origin, the new coordinate of this Node will be (-1.50, -1.00, -0.10+P). In order to project this point onto the screen, we can scale it based on similar triangles. When z=0, we know that the point vanishes so x and y both becomes zero. This serves as a reference point for our calculations. If the screen is located at z=S=5, then our new projected x-coordinate will be:

$$x' = \frac{S}{z}(x) = \frac{5}{-0.10+P}(-1.50)$$

and our new projected y-coordinate will be:

$$y' = \frac{S}{z}(y) = \frac{5}{-0.10+P}(-1.00)$$

The projected depth is a little more complicated so I will not go through it here since this is not the main topic of this series of posts. If you would like more information on it, you can look up **here** for more information. Actually, on this point, I learnt quite a bit about computer graphics, and it is by no means an easy topic. I should have just went ahead to use the openGL libraries for a better demonstration but I chose to implement everything from scratch. As a result of this, it took me much longer to get things done, and there were some unresolved issues as well. Anyway, I hope that through this post, you will get to know more about computer graphics, and the multitude of problems that follows.

Alright, back to the point, the above equations are implemented in my python **BoardDisplay** code. Below is an extract of the code:

```
def projectOnePointPerspective(self, x, y, z, win_width, win_hei
    # In Pygame, the y axis is downward pointing.
    # In order to make y point upwards, a rotation around x axis
    # This will result in y' = -y and z' = -z
    xPrime = x
    yPrime = -y
    zPrime = -z
    xProjected = xPrime * (S / (zPrime + P)) * scaling_constant
    yProjected = yPrime * (S / (zPrime + P)) * scaling_constant
    pvDepth.append(1 / (zPrime + P))
    return (round(xProjected), round(yProjected))
```

We are now ready to draw the points onto our Pygame screen so let us move to the next section

Drawing in Pygame

In order to display the object, I used Pygame's draw polygon method. Here's the **documentation** for it if you want to check it out. Basically, we can use the `pygame.draw.polygon` method to draw a polygon by supplying it with the a list of points. Since our block is simple a six sided rectangle, we have to call this function with a list of 4 points for a total of 6 times, one time for each face, in total. However, it is impossible to show all 6 sides of the block because at any one time, we would be able to see only 3 of the faces. So how do we determine which faces are visible? We can do this using the simple painter's algorithm.

When painting landscapes, we usually start painting from the furthest landscapes, then gradually paint over those landscapes to draw objects that are closer. This is what the painter's algorithm does as well. All we have to do is to order the 6 faces in terms of its depth (at the center of the face), then draw the polygon starting from the furthest face. This is implemented in the code as follows:

```
# Calculate the average Z values of each face.
avg_z = []
for face in self.wireframe.faces:
    n = pvDepth
    z = (n[face.nodeIndexes[0]] + n[face.nodeIndexes[1]] +
         n[face.nodeIndexes[2]] + n[face.nodeIndexes[3]]) / 4.0
    avg_z.append(z)
# Draw the faces using the Painter's algorithm:
for idx, val in sorted(enumerate(avg_z), key=itemgetter(1)):
    face = self.wireframe.faces[idx]
    pointList = [pvNodes[face.nodeIndexes[0]],
                 pvNodes[face.nodeIndexes[1]],
                 pvNodes[face.nodeIndexes[2]],
                 pvNodes[face.nodeIndexes[3]]]
    print(pointList)
    pygame.draw.polygon(self.screen, face.color, pointList)
```

First, we determine the depth of the center of the face (average depth of the nodes of the face), then sort it in increasing order and draw them in order. That's it. Simple isn't it?

However, the painter's algorithm will not work for many situations. For example, you will see later that the painter's algorithm does not work perfectly in the perspective view as it is. I do not yet have a solution for this problem so if you any suggestions, please write it down in the comments section. I'll appreciate any help that I can get.

Anyway, for now, your block is stationary so you cannot see anything except for a green rectangle like this:

SIMPLE PYGAME BLOCK DISPLAY

In the next section, we will be starting on the topic of quaternion rotations so you will be able to make the stationary block above rotate.

Go to Next Section

ATTITUDE DETERMINATION WITH QUATERNION USING EXTENDED KALMAN FILTER

Posted on July 10, 2018

Table of Contents

Foreword

It has been around 3 months since I last posted something on my blog, and the reason for that is THIS! Well, part of it is because I had to work in the day and could only continue this project in the night and during weekends, but another part of it is due to the inaccurate information from sites that come out from google search. Once you get into a more difficult topic, you will realize that the only sources of information that you can rely on are journal papers. Implementations that are readily available online and look simple are really attractive when you are starting on a new topic, but they usually prove to be misleading and wrong so always be critical of what you read online. Just because it "looks" like it is working, doesn't necessarily mean that it is working correctly.

I did a 2D implementation for sensor fusion in **this project** but moving on to a 3D implementation was really a huge jump in my opinion. I took a long time to get things to work, and I hope that this tutorial will help those who are starting out in the world of 3D navigation systems to understand the concepts better. To this end, I will provide a sample program to illustrate a simple (relatively) Extended Kalman Filter Implementation which fuses 3 sensors, namely the gyrometer, accelerometer and the magnetometer.

Introduction

When you search for articles regarding navigation systems or attitude determination systems, you will most likely, if not surely, encounter the Kalman Filter as well. Specifically, the Extended Kalman Filter (EKF) or the Unscented Kalman Filter (UKF). In this project, I will demonstrate an implementation of attitude determination with quaternions using the EKF, and also explain the need to fuse several sensor data together in order to get the system to work.

The sensors that will be used are the gyrometer, accelerometer and the magnetometer. The Arduino is used to read data from the sensor, but the data processing will be done in python. In addition to this, I created a simple display using Pygame in order to visualize the results better. I will also provide a simple method for calibrating the magnetometer since their readings vary greatly depending on the surroundings.

This series of post will assume that you are familiar with Python, and have some basic knowledge on matrix operators and calculus. Don't worry if you are not so familiar with the mathematics because I will try to explain them as simply as I can. For starters, here's a video of what you will get out of this project.

So without further ado, let us begin by preparing the required visualization tool in Pygame.

Go to Next Section