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

$$q = q_0 + q_1 \tilde{i} + q_2 \tilde{j} + q_3 \tilde{k}$$

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,

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

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, $q = q_0 + q_1\,\tilde{i} + q_2\tilde{j} + q_3\tilde{k}$, is defined as follows.

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

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.

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

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

$$\tilde{i}\,\tilde{i} = \tilde{j}\tilde{j} = \tilde{k}\tilde{k} = -1$$

$$\tilde{i}\,\tilde{j} = \tilde{k} = -\tilde{j}\,\tilde{i}$$

$$\tilde{j}\tilde{k} = \tilde{i} = -\tilde{k}\tilde{j}$$

$$\tilde{k}\tilde{i} = \tilde{j} = -\tilde{i}\,\tilde{k}$$

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

$$C = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix}$$

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:

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

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

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

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:

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

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 quaternion

    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()) + q
```

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!

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

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

CASE 1

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

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

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

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

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

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, c21, c22]])
    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.dataNumBytes + i*self.dataNumBytes)]
        value,  = struct.unpack(self.dataType, data)
        if i == 0:
            value = ((value * 0.00875) - 0.464874541896) / 180.0 * np.pi
        elif i == 1:
            value = ((value * 0.00875) - 9.04805461852) / 180.0 * np.pi
        elif i == 2:
            value = ((value * 0.00875) + 0.23642053973) / 180.0 * np.pi
        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

PREVIOUS POST
Preparing the Graphics

NEXT POST
Calibrating the Magnetometer

POSTED IN ATTITUDE DETERMINATION WITH QUATERNION USING EXTENDED KALMAN FILTER

11 comments on "*Quaternion Rotation*"

Samuel Garcia

April 11, 2019 at 4:00 am

Hi, my name is Samuel Garcia, I'm student from Mexico, I have a question, How did you get the calibration data?, I have a MPU9250 but I don't know how to calibrate it, in the magnetometer part I followed your tutorial but How to calibrate the accelerometer and the gyroscope? I know something about the offset but I don't know in what part of the code do I have to put it? because in the function getSerialData you have some data that add and subtract but but they do not look anything like my offsets. I hope you can help me.

Thank you for your time.

rikisenia.L

April 12, 2019 at 12:01 am

Hi Samuel,

The calibration for the gyro and accelerometer is really simple. I just took data for a period of time while the sensor is stationary and I took the average of all the values to use as the offset.

I suppose you are referring to the constant that "value" is multiplied by. The first 3 data comes from the gyro, and 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] (line 26 of the Arduino Code), 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,
1) (value * 0.00875) // convert raw gyro sensor data to units of degrees
2) (value * 0.00875) – 0.464874541896) // correct the sensor bias with an offset
3) ((value * 0.00875) – 0.464874541896) / 180.0 * np.pi // convert units to radians

The calibration of the accelerometer follows the same concept.
Hope this clears thing up.

Cheers!

Samuel Garcia

May 2, 2019 at 7:40 pm

Hi, thank you so much for the answer and sorry for answering so late.

Samuel Garcia

May 2, 2019 at 7:42 pm

Hi

Samuel Garcia

May 3, 2019 at 3:50 am

I have another question, I don't know what happend but When I'm running the program(with Extended Kalman Filter Implementation), when I do the pitch, roll and yaw movements, I get strange data. For example when moving in pitch to 90 degrees on both sides, roll and yaw are not affected (well a little), but when I roll to 90 degrees, the yaw is very affected, in fact, it starts the piece as to rotate, the truth is that it does not happen and I've been testing several things, I attach a video so you can observe. I hope you can help me and I appreciate your attention.

https://youtu.be/IGXa0kxau6c

rikisenia.L

May 9, 2019 at 12:12 am

Hi Samuel,

Apologies for the late reply. I was on a holiday.
Did you try fiddling with values in the process and measurement covariance matrices?
I think the problem may be because the kalman filter have not yet converged while you are carrying out the experiment.
This problem can sometimes be solved by leaving the sensor stationary for a few seconds first after running the program or by changing the values of the Q and R covariance matrices.

SensorNovice

June 18, 2019 at 5:27 am

Hi rikisenia.L,

I am having the same problem as Samuel. I tied your recommendation about keeping the sensor stationary for few seconds after running the program.

I am manipulating the process and measurement covariance matrices, now. When I read your tutorial you wrote that p, Q and R values were got from experimentation and observation. Could you explain a little more on how to get the process and measurement covariance matrices ? It would be really helpful if there was already a process of getting the constant multiplicative factor, so that I am just not randomly changing those numbers and then looking at the result.

Thanks, for your help.

rikisenia.L

June 18, 2019 at 9:01 pm

Hi SensorNovice,

If you are using a different sensor from me, you might have to fiddle with the P,Q,R constants to get better results.
Unfortunately, I do not have a set process of obtaining the values as I obtained it through trial and error. You can first try placing a very small value in each of the variable to see how it affects the results individually, and from there you can then fine tune the values.

Samuel Garcia

October 24, 2019 at 5:02 am

Hi SensorNovice and rikisenia

So far, have you had any favorable results ?, because I have not been able to achieve it, when they say that we manipulate covariance matrices, do they refer to constants? those of 0.01.0.001.0.1 ?, because I have tried and I can not keep the yaw at 0 when, I am only moving pitch.

Thanks for the help

Guido

November 28, 2020 at 8:25 pm

You don't know how much you helped me 😀 , thanks for the great work!

Claudio Ortega

July 3, 2021 at 9:30 am

Hi Samuel,

Congratulations on this so well-done blog, it is just outstanding.
I am an electronic engineer and enthusiast of quadrotors.
I am working on my own implementation of the attitude control for a quadrotor.
I am planning to keep notes of my journey, and the looks of your blog are very appealing,
My simple question is how did you write this particular page, I mean what tool/process, etc.

Thank you for your time.

Claudio

Comments are closed.

Categories

Select Category

Archives