Philip Salmony

IMU Attitude Estimation

1. Introduction

An important part of controlling an Unmanned Air Vehicle (UAV) is having the correct states available for feedback in the governing control system. Unfortunately, sensors employed on UAVs, such as gyroscopes and accelerometers (usually packaged together and dubbed *Inertial Measurement Unit (IMU)*), have several problems associated with them:

- Gyroscopes measure angular rate of change and not angular position directly.
- Accelerometers measure more than just linear acceleration, for instance, gravitational acceleration and Coriolis terms.
- Measurements are noisy and biased.
- Body-frame states need to be transformed to the inertial reference frame (e.g. Euler angles).

We aim to develop a method to accurately determine the UAVs attitude (excluding yaw, since this requires an additional magnetometer) purely from the raw accelerometer and gyroscope readings.

I am using an MPU-6050 IMU (6 degrees of freedom, see *Figure 1* below), which is readily available and inexpensive. It uses an I^2C interface and therefore can be easily combined with an Arduino or Raspberry Pi.

Figure 1: MPU-6050 IMU

Implementations of the algorithms presented, written in MatLAB and Python, are available on GitHub.

See the algorithms in action on Youtube.

2. Preliminaries

Before we get into developing algorithms for attitude estimation, we need to define our notation and cover some sensor basics.

Notation
Throughout, we will be using the following notation:

- Ax, Ay, Az – Raw acceleration measurements along each axis ($\frac{m}{s^2}$).
- Gx, Gy, Gz – Raw gyroscope measurements about each axis ($\frac{deg}{s}$), usually referred to as p, q, and r respectively.
- ϕ, θ – Roll angle (rotation about X axis in degrees) and pitch angle (rotation about Y axis in degrees).
- $\hat{\phi}$ – Indicates an estimate of the roll angle.

2.1 Calibration

Both sensors will have an offset (bias), which means at rest they will not read exactly zero – even though in theory they should. This can be mitigated by keeping the IMU at rest, summing the measurements over a period of time, and finally dividing by the total number of measurements. This final value – the average offset – can then be subtracted from all future

measurements. This final value – the average offset – can then be subtracted from all future readings to ensure that the actual measurement is close to zero when the system is at rest and level.

2.2 Accelerometer

When the IMU is at rest, the accelerometer measures gravitational acceleration only. Therefore, by resolving forces using basic trigonometry, we can calculate an estimate of both pitch and roll using the raw data as follows:

$$\hat{\phi}_{Acc} = arctan(\frac{A_y}{\sqrt{A_x^2+A_z^2}}) \cdot \frac{180}{\pi}$$

$$\hat{\theta}_{Acc} = arctan(\frac{A_x}{\sqrt{A_y^2+A_z^2}}) \cdot \frac{180}{\pi}$$

We need to low-pass filter the raw accelerometer data before using the equations above, since there will be additive, high-frequency noise overlayed on the signal. Furthermore, in motion the accelerometer will measure additional acceleration terms which deteriorate our estimations for pitch and roll.

Figure 2 shows an estimate of the roll angle using only the accelerometer data, which shows the noisy and unreliable nature of this data.

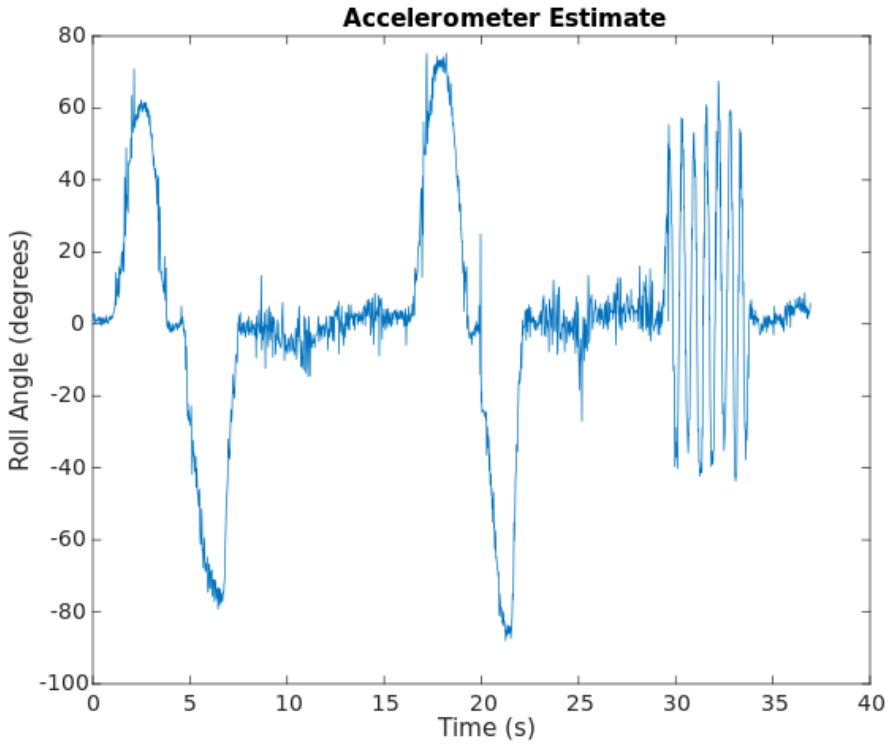

Figure 2: Roll Estimate from Raw Accelerometer Data

If we low-pass filter the raw accelerometer data, for example by using a simple moving average

filter, we get a much cleaner estimate. This is shown in *Figure 3* below. The amplitude of the high-frequency motion however, from $t = 30s$ to $t = 35s$, is attenuated by the low-pass filter – this is not ideal, since this will not give us the correct roll angle at higher frequencies (original estimate goes up to about 60deg, filtered estimate reaches 40deg)! As always, there is a trade-off between how well we can 'clean up' our signal, and how much we impact our response.

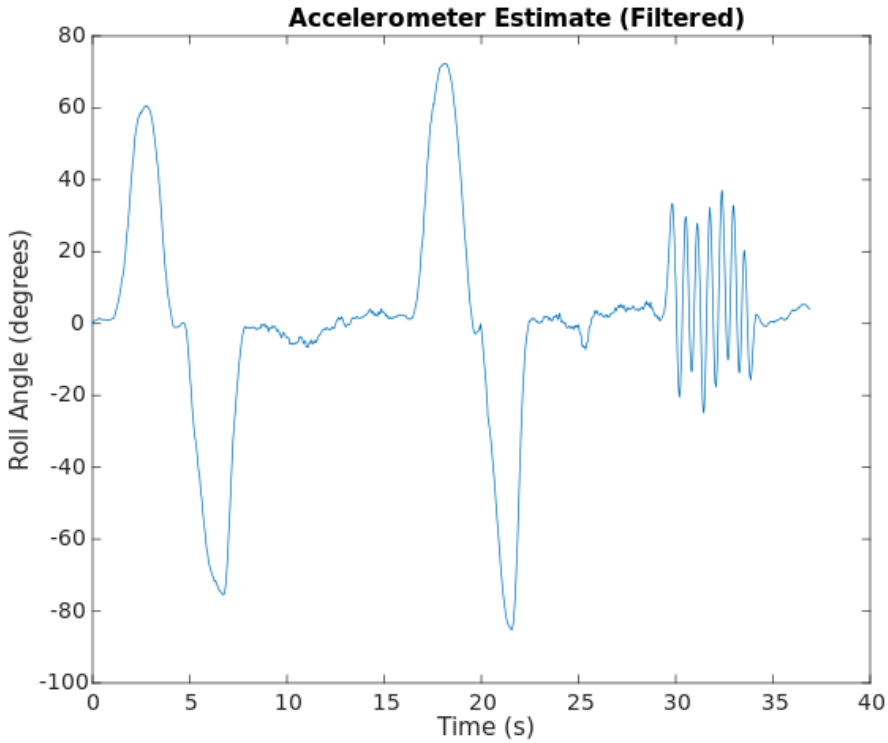

Figure 3: Roll Estimate from Filtered Accelerometer Data

2.3 Gyroscope

The gyroscope measures angular rates of change around each axis with respect to the body frame. Assuming noise-free measurements, we could integrate these raw measurements to get the angular position relative to the body frame. Unfortunately, as usual we will have a fair amount of noise present in the gyroscope readings. If we were to integrate this, we would add up all the small errors due to noise, which then would accumulate over time and become unbounded in magnitude. This is commonly known as gyroscopic drift.

Even heavily low-pass filtering the signal will not help, as firstly this will cause a lag in the measurements and secondly will only delay the onset of drift, since we cannot completely eliminate noise in our signal. (Furthermore, integration on its own already acts as a low-pass filter, which we can best see in the Laplace domain, where integration is simply $\frac{1}{s} = \frac{1}{j\omega}$.)

Figure 4 shows an estimate of the roll angle acquired by integrating the noisy gyroscope data. The drift can be clearly seen, even after such a short period of time.

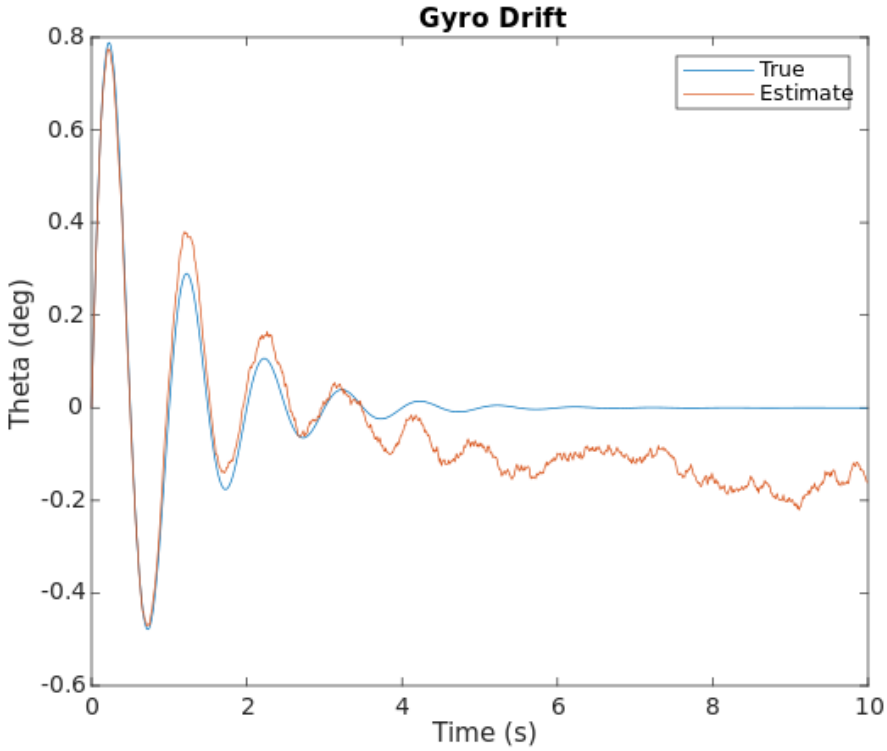

Figure 4: Gyroscopic Drift

Important: As mentioned above, the gyroscopes measure angular rates of change in the body frame. For example, imagine we start from rest and pitch up to 45 degrees, then yaw to either side by rotating 90 degrees about the yaw body frame axis – this will decrease our pitch angle in the inertial frame, simply by yawing in the body frame! Therefore, we **cannot** simply integrate the angular rates of change given by the gyroscope to get our roll, pitch, and yaw angles in the inertial frame. The rates of change of Euler angles depend on more than one body frame angular velocity. To get the rates of change in the inertial frame, we need to transform our states using the following matrix:

$$\begin{pmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\psi} \end{pmatrix} = \begin{pmatrix} 1 & \sin\phi \tan\theta & \cos\phi \tan\theta \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi \sec\theta & \cos\phi \sec\theta \end{pmatrix} \begin{pmatrix} p \\ q \\ r \end{pmatrix}$$

We can then integrate these transformed states – assuming zero noise that is... – to get the inertial attitude. We will use this transformation in all our sensor fusion algorithms.

3. Sensor Fusion

As shown in the previous section, both sensors available to us are far from ideal. Accelerometers are fine for when the system is not in motion and gyroscopes are fine for short periods of time, but over longer periods of time – individually – both sensors will not give

reliable estimates of pitch and roll.

The principle behind sensor fusion, as the name implies, is to combine the data retrieved from two or more imperfect sensors to give us an improved and more reliable estimate of a state.

We will explore some fundamental sensor fusion techniques in this section.

3.1 Complimentary Filter

The complementary filter works by combining the desirable low-frequency characteristics of the accelerometer with the desirable high-frequency characteristics of the gyroscope. Essentially, we are performing the calculations described in Section 2.3 and 2.4, filtering the results, and then combining the two estimates to give a single improved estimate.

The method is as follows:

1. Choose a constant α, such that 0 < α < 1. The larger α, the more the accelerometer measurements are 'trusted'. As α goes to zero, we base our estimate mainly on the gyroscope measurements. A good starting point is $\alpha = 0.1$.
2. Initialise state estimate, e.g. $\hat{\phi}_{t=0} = 0$.
3. For each timestep (sampling time Δt)...
    a. Retrieve raw accelerometer and gyroscope readings from IMU.
    b. Calculate estimate of angle from accelerometer data ($\hat{\phi}_{Acc}$) using equation from Section 2.2.
    c. Combine this estimate with the integral of the **transformed** gyroscope data (*see Section 2.3*): $\hat{\phi}_{t+1} = (1-\alpha) \cdot (\hat{\phi}_t + \dot{\phi}_G \cdot \Delta t) + \alpha \cdot \hat{\phi}_{Acc}$

The final step (3c) is a difference equation which performs the actual filtering. The accelerometer measurements are low-pass filtered, whereas the gyroscope measurements are high-pass filtered. These signals are then combined, depending on the constant α, to form the final state estimate.

The complimentary filter has many advantages. For instance, there is only one tune-able parameter (α), it is easy to implement in a few lines of code and does not require a lot of processing power, and lastly it delivers satisfactory performance. This can be see in *Figure 5* below, showing the complimentary filter running for two different values of α.

For $\alpha = 0.4$, the complimentary filter relies quite heavily on the accelerometer data (which we feed into the complementary filter **without** pre-low-pass filtering it, since that is the complimentary filter's job!), and therefore we receive a fairly noisy output. As we lower α to 0.1,

we rely more on the gyroscope data. As can be seen from the plot and in comparison to only using the accelerometer for estimation (see *Figure 3*), the high-frequency motion is less attenuated, however the noise components are filtered out quite nicely.

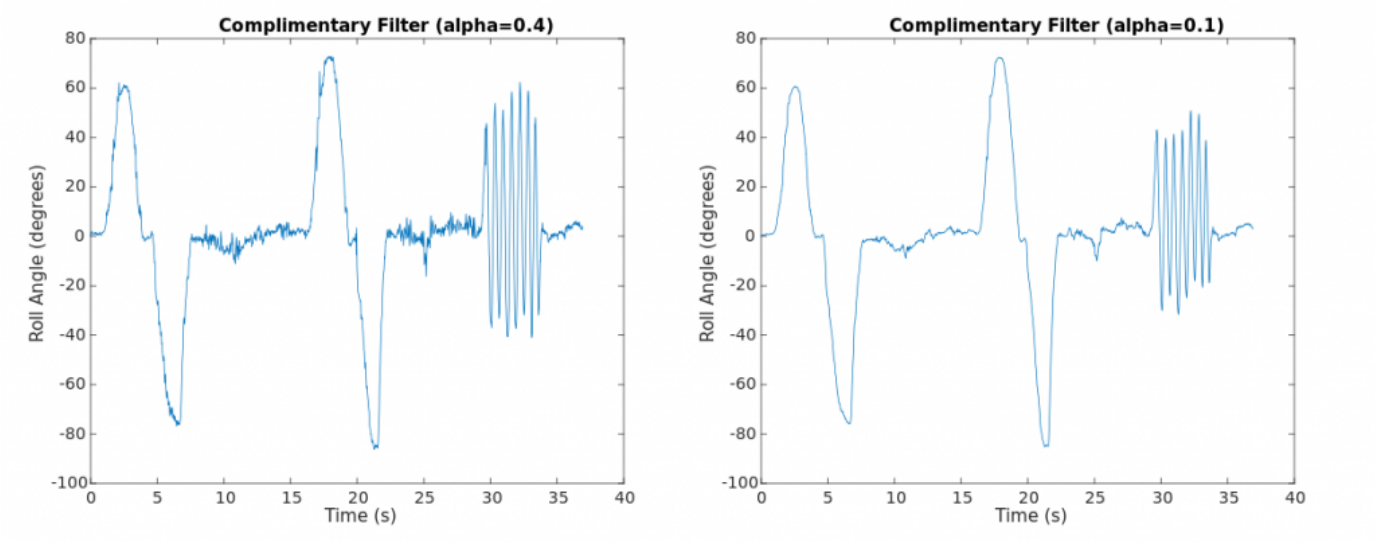

Figure 5: Complimentary Filter for $\alpha = 0.4, 0.1$

3.2 Kalman Filter

You might be wondering how we best choose α in the complimentary filter above. We have to take several things into account, for example, sensor noise and accuracy (accelerometer vs gyroscope) and types of motion encountered. It seems like the best choice of α will vary from situation to situation, and even vary over time when running on the same system! Fortunately, there is a way of choosing filter coefficients like these optimally (mathematically at least...) – let's take a look at the Kalman Filter which does exactly that.

The Kalman Filter is also know as a Linear Quadratic Estimator. It is a type of observer or state estimator which is optimal in the sense that it tries to minimise a quadratic cost function. Furthermore, the Kalman Filter doesn't just take the sensor measurements into account but also the underlying dynamics of the system. I will state the Kalman Filter equations and explain how to implement them. As there is a plethora of information already available both online and in various text books, I will only give a brief explanation of what the filter is doing.

3.2.1 System Dynamics

A causal, linear, and time-invariant system can be described in state-space form using the following format (discrete time):

$$\vec{x}_{t+1} = \mathbf{A} \cdot \vec{x}_t + \mathbf{B} \cdot \vec{u}_t + \vec{w}_t$$

$$\vec{y}_{t+1} = \mathbf{C} \cdot \vec{x}_{t+1} + \vec{v}_{t+1}$$

Where $\vec{x}_t$ is the system's state vector at time t and $\vec{u}_t$ is the input vector at time t. **A**, **B**, and **C** are the system matrices: **A** relates the current states to the next states, **B** relates inputs to the next states, and **C** relates the system states to the measured states.

Lastly, we have additive process ($\vec{w}_t$) and measurement ($\vec{v}_t$) noise – both assumed to be zero-mean Gaussian processes.

Since we are going to be writing an algorithm that targets a large variety of dynamic systems, we will have to formulate our system dynamics as generally as possible. We know which IMU we will be using, but not if it's going to be placed on a UAV, car, robot, etc. – that is down to the end-user of our software. Unfortunately, this will impact our Kalman Filter performance significantly – if we had an accurate model of our actual system dynamics, our filter performance would improve.

One way of constructing the system matrices as general as possible is to recognise the following:

- We wish to eliminate the varying gyro bias at every moment in time as this causes drift.
- We can use the **transformed** angular rates of changed measured by the gyros – subtracting the bias – to get an updated estimate of our angular position.
- We can form estimates of our angular position by using the accelerometers.

We then define our state vector, input vector, and measurement vector:

$$\vec{x}_t = \begin{bmatrix} \hat{\phi}_t \\ b_{\hat{\phi}_t} \\ \hat{\theta}_t \\ b_{\hat{\theta}_t} \end{bmatrix}$$

$$\vec{u}_t = \begin{bmatrix} \dot{\phi}_{G_t} \\ \dot{\theta}_{G_t} \end{bmatrix}$$

$$\vec{z}_t = \begin{bmatrix} \hat{\phi}_{Acc_t} \\ \hat{\theta}_{Acc_t} \end{bmatrix}$$

Where b_{ϕ_t} is the gyro bias at time t associated with our estimate $\hat{\phi}$. This leads us to our general state-space form:

$$\vec{x}_{t+1} = \begin{bmatrix} 1 & -\Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \vec{x}_t + \begin{bmatrix} \Delta t & 0 \\ 0 & 0 \\ 0 & \Delta t \\ 0 & 0 \end{bmatrix} \vec{u}_t + \vec{w}_t$$

$$\vec{y}_{t+1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \vec{x}_{t+1} + \vec{v}_t$$

3.2.2 Kalman Filter Equations

Now that we have defined our system in state-space form, we can take a look at the Kalman Filter equations. These can be put into two groups: *Prediction* and *Update*.

Prediction

$$\vec{x}_{t+1} = \mathbf{A} \cdot \vec{x}_t + \mathbf{B} \cdot \vec{u}_t$$
$$\mathbf{P} = \mathbf{A} \cdot \mathbf{P} \cdot \mathbf{A}^T + \mathbf{Q}$$

Update

$$\widetilde{y}_{t+1} = \vec{z}_{t+1} - \mathbf{C} \cdot \vec{x}_{t+1}$$
$$\mathbf{S} = \mathbf{C} \cdot \mathbf{P} \cdot \mathbf{C}^T + \mathbf{R}$$
$$\mathbf{K} = \mathbf{P} \cdot \mathbf{C}^T \cdot \mathbf{S}^{-1}$$
$$\vec{x}_{t+1} = \vec{x}_{t+1} + \mathbf{K} \cdot \widetilde{y}_{t+1}$$
$$\mathbf{P} = (\mathbf{I} - \mathbf{K} \cdot \mathbf{C}) \cdot \mathbf{P}$$

As an analogy to the complimentary filter mentioned above, $\mathbf{K}$ is known as the Kalman gain, which is optimally chosen depending on what the filter thinks is giving a more reliable estimate of the state – the measurement or the system dynamics model.

$\mathbf{P}$ is the error covariance matrix and is initially set by us and then updated by the Kalman Filter. If we are not sure if our guess of the initial state is correct, we will increase the values in this matrix. However, if we are confident that our initial guess is close to the actual initial state of the system (e.g. we have a quadcopter placed on level ground, $\phi = 0$ and $\theta = 0$), we should decrease the size of the elements in $\mathbf{P}$.

$\mathbf{Q}$ and $\mathbf{R}$ are covariance matrices of the process and measurement noise respectively. $\mathbf{Q}$ is set to tell the filter how unsure we are about our model dynamics, i.e. large values if we believe our

model is inaccurate. $\mathbf{R}$ is set depending mainly on the sensors used in the system, where noise-specific parameters can usually be found in the sensor's datasheet. Again, large values in $\mathbf{R}$ mean greater noise levels.

Generally, $\mathbf{P}$, $\mathbf{Q}$, and $\mathbf{R}$ are diagonal matrices.

3.2.3 Implementation and Performance

Once we have the dynamic model and the Kalman equations, the implementation is fairly straightforward. The only 'difficult' aspect to implementing a Kalman Filter on a microcontroller is that we need to be able to do matrix inversion and numerical libraries are only more readily available for other higher level systems and languages. In my case, I am using a Raspberry Pi 3 and Python, and therefore can use Numpy which handles all matrix operations.
In short, we initialise our state estimate vector $\vec{x}_{t=0}$ and error covariance matrix $\mathbf{P}_{t=0}$, then determine our noise covariance matrices $\mathbf{Q}$ and $\mathbf{R}$, and finally update our state estimate whenever a measurement is available using the equations given in the previous section.
Figure 6 below shows the performance of the Kalman Filter compared to the complimentary filter. It is clear that with our general system model, only a slight performance gain (if any) can be gained by using the Kalman Filter. Furthermore, the implementation of the complimentary filter is far simpler. Had we implemented the Kalman Filter on a system where we knew an accurate dynamic model, the Kalman Filter would – in pretty much all cases – trump the simpler complimentary filter.

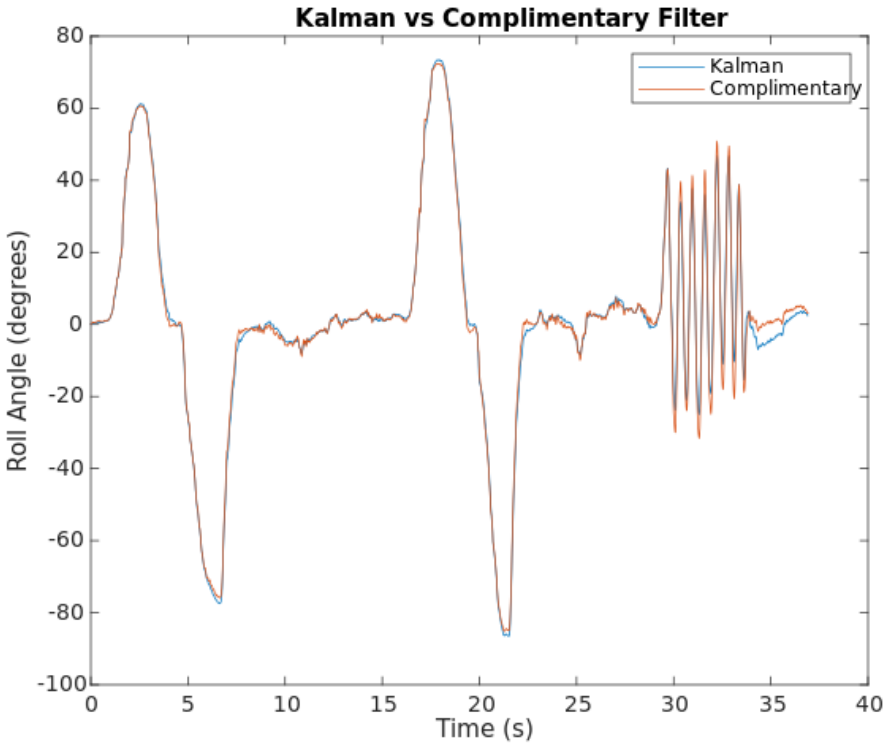

Figure 6: Kalman vs Complimentary Filter

4. Conclusions

As shown in the previous sections, we cannot use simply one of the two sensors, as the accelerometer data is too noisy and the gyroscope data will cause drift. We achieve the greatest sensor fusion performance and easiest implementation by utilising the complementary filter, which merges both measurement sources.

If we have an accurate description of our dynamic system, we should choose the Kalman Filter instead.

An overall comparison of all state estimation schemes is given in *Figure 7* below.

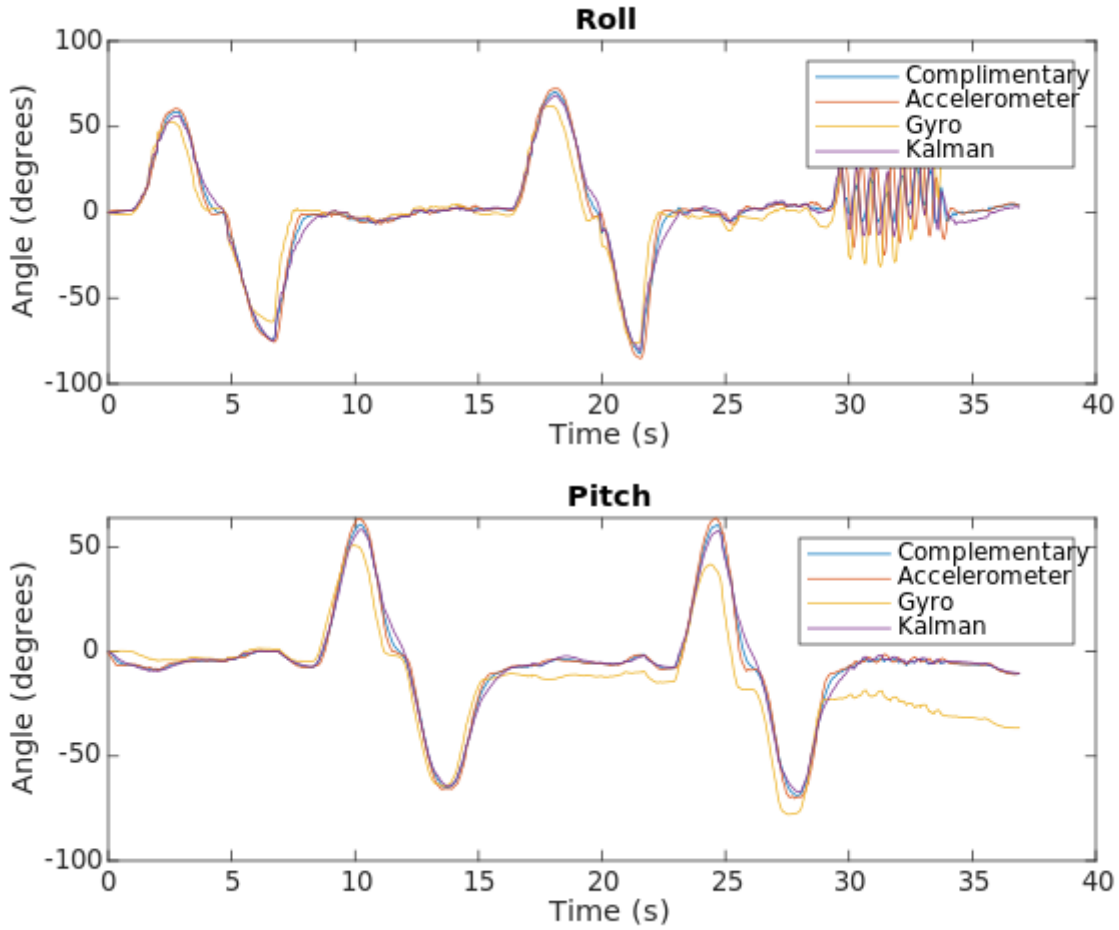

Figure 7: Comparison of All Estimation Schemes

5. Appendix: Data and Code Listings

Raw Gyroscope and Accelerometer Data: imu_data.csv

Code Listings are available on GitHub: https://github.com/pms67/AttitudeEstimation

29 Replies to “IMU Attitude Estimation”

Colin Runnion
FEBRUARY 26, 2020 AT 12:22 PM

Can you fix your figures on this website? They appear blown-up and I believe there is some incomplete data from it. I would very much like to use this well written site!

Reply

phils94
FEBRUARY 26, 2020 AT 12:32 PM

Thanks for pointing that out. I hadn’t seen that changing the site template messed up the figures. Should be fixed now.

Reply

Jay
MARCH 9, 2020 AT 6:27 PM

I was looking at the code and for get_gyro you’re dividing the value by an additional 131.0 and in get_acc you dividing the value by 16384.0
What exactly are these values?

Reply

phils94
MARCH 9, 2020 AT 6:50 PM

Those numbers are the LSB sensitivities for a given full scale range (i.e. how many LSBs

Those numbers are the LSB sensitivities for a given full scale range (i.e. how many LSBs per g (accelerometer) or how many LSBs per deg/s (gyroscope)). I.e. how to convert from raw device readings to relevant units.

If you look at the MPU6050 datasheet/register map document (https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Register-Map1.pdf), you'll see that on page 29 for a +-2g accelerometer scale range, the sensitivity is 16384 LSB/g.

On page 31, for a +-250deg/s gyroscope scale range, the sensitivity is 131 LSB/(deg/s).

Reply

Don Reynolds

MARCH 24, 2020 AT 5:12 AM

This is a great article, thank you.

Reply

phils94

MARCH 29, 2020 AT 10:56 PM

Thank you!

Reply

Simon Jeppesen

MARCH 28, 2020 AT 5:40 PM

This is absolutely fantastic. I'm doing my bachelor on TVC model rockets and I'll definitely be able to use some of what you have described above. Would you say that there is any significant processing advantage in using the complementary filter over the DMP sensor fusion that is already available? It seems that the FIFO buffer overflows quite easily if multiple sensors are on the same I2C bus.
Basically I'm trying to implement a PID loop but I think the DMP might require too much processing power.

Thanks a bunch for taking the time!

Reply

phils94

MARCH 29, 2020 AT 10:55 PM

Thank you!
The complimentary filter effectively can be written in a couple lines of code, which computationally (and space-wise) is a great advantage over the DMP libraries. The DMP should definitely be more robust however. Do you have a way of comparing the two?

However, given that you're going to be using this on a TVC rocket and there'll be very high accelerations, the approaches outlined on this page probably won't yield the greatest of results. If you know the system dynamics, you'll fare much better with an 'Extended Kalman Filter'. See: https://github.com/pms67/EKF-Quaternion-Attitude-Estimation

Let me know if you need some more details.

Reply

Robert

APRIL 12, 2020 AT 8:31 PM

Hi Philip,
I added your KalmanRollPitch.h implementation for testing.
I noticed that the roll and pitch angle always return towards zero – even when the fc remains tilted.
cheers
Robert

Reply

phils94

APRIL 12, 2020 AT 9:26 PM

Hi Robert,
That's quite peculiar. Over what time period does it return to zero? And what hardware are you using to test it?

are you using to test it?
I've tested it both in simulation and in real-time on hardware and haven't noticed that.
Thanks,
Philip

Reply

Robert
APRIL 15, 2020 AT 11:31 AM

Hi Philip,
yes, my fault I used the wrong units 🙁
pointing very 'hard' to the earth.

You are using double functions everywhere.
/* Compute common trig terms */
float sp = sinf(kal->phi);

use functions with an 'f' at the end.
and instead of fabs use fabsf.
The compiler warnings will disappear 🙂

Cheers
Robert

Reply

phils94
APRIL 22, 2020 AT 7:30 PM

Hi Robert,

Ah glad you could fix it! 🙂

Thank you – good catch, I'll change all the double functions to single precision!

Best,
Philip

Reply

HD

APRIL 22, 2020 AT 9:46 AM

Hi Philip,

I have some questions for you 😀

– For all results (Kalman and complimentary filters), you apply beforehand a low pass to the phi_Acc and theta_Acc?
– I try to use your .csv file to make a similar result and realize that Ax, Ay and Az is very small. Did you remove the gravity from these accelerations?

Cheers
HD

Reply

phils94

MAY 17, 2020 AT 10:06 PM

Hi,

Ideally, you would want to apply 'light' low-pass filtering to the measurements before feeding them into the Kalman/Complimentary filters, even though your results should be okay without.
To smooth the incoming data, a moving average filter would be best (i.e. the filter output is the sum of N past measurements).

The CSV file gives the data in units of 'g', so multiply by 9.81m/s^2 to convert to m/s^2.

Philip

Reply

HD

APRIL 22, 2020 AT 10:54 AM

Hi Philip,

I have another question, when you calculate the rates of change in the inertial frame, the

I have another question, when you calculate the rates of change in the inertial frame, the matrix you used does not depend on psi? It seem to be weird for me. Can you explain it to me? Please.

Cheers
HD

Reply

Kai Horstmann

MAY 18, 2020 AT 7:31 PM

Hi,

I am a glider pilot, and I am working on an integrated electronic vario with fully electronic compensation against 'stick thermals' (Lift or sink indication due to moving the stick, and not changes in air currents).
To achieve this I need at least:
– 3-Axis Accelerometer
– 3-Axis Gyroscope
– Absolute pressure sensor: Altitude, and absolute, uncompensated rate of lift or sink, athmospheric densitiy to calculate true air speed from dynamic pressure (see next).
– Relative pressure sensor: For dynamic or impact or ram pressure. Used to calculate Indicated Air Speed (IAS). With absolute pressure I can calculate the True Ais Speed (TAS).
Optional:
– GPS: Absolute position, and ground speed. When cireling in thermals I can estimate the wind drift over time.
– Magnetometer: To improve speed of attitude estimate, and minimize gyro drift, and measure wind drift even in straight flight.

The physical model of a fixed wing aircraft (UAV in your case, or man-carrying glider in my case) is very different from naive assumptions about assessing the attitude by measuring the accelerometer. That may work to a degree for multicopters which spend most of their time in hover flight, or stationary flight straight, except for accelerating or decelerating.
A glider spends on a typical cross-country flight between 30%-70% of the time turing in thermals in very narrow cicles at bank angles of 30 to 50 degrees. And in coordinated flight your acceleration will always point straight down relative to the airplane coordinate system because centrifugal force and gravity are perfectly balanced. Otherwise you are a lousy gilder pilot.
To assess the centrifugal force you need 2 things:
Your true speed relative to the air, and your turn rate (horizontally in world coordinates!)

Your true speed relative to the air, and your turn rate (horizontally in world coordinates!).
Your Gyro will give you a 50-50-mix of the Z-Axis and Y-Axis when turning in a 45° angle circle. So here you need at least a rough initial idea of your attitude.
For your true air speed you need the dynamic pressure, and the absolute pressure to compensate for the decreasing air density with altitude (a real factor for gliders).

With GPS and magnetometer I can in addition estimate the wind too, and compensate for drifts and biases from the IMU sensors. (However my magnetometer has the worst bias internal bias; throw magnetic fields in your vehicle into the mix, and you get the idea.
In your case moving the magnetometer away from your cool board away is IMHO a must, given the strong currents your actuators and stuff will draw at varying rates.

This is a long running pet project of mine, and making slow progress. You can find the code with the physical model, and measurement model on https://github.com/hor63/openEVario if that is helpful for you.
I am moving the inertial sensors away from the instrument panel out to a sensor board. At the moment my MCU there is an ATMega1284p. Connection is planned serial over LVDS (Current prototype runs with RS-422). With gliders you try to save every milliamp because all power comes from batteries, and flights can last up to 10h. Communications protocol is TCP/IP via PPP over Serial with the board which is barely within the capacity of the ATMega. I am planning to upgrade to STM32L452, again a low-power type.
The main program in the link is planned to run on a Cubieboard, or a similar board sporting an Allwinner A20 with dual Cortex A7 ad ~1GHz, and 1GB memory which should give the necessary Boom for the big Kalman filter (32 coefficients, the Matrixes have therefore 1024 members). I plan to drive the Display with that board with the embedded Mali GPU on a 3' round display. Some time.
Feel free to contact me if you have questions or better ideas.

Cheers,
Kai

Reply

yoni
JULY 6, 2020 AT 9:33 PM

Hello,
I saw that you always calculate the roll and pitch. What is the reason that you cannot get the yaw also?

Reply

Kai Horstmann

NOVEMBER 10, 2020 AT 2:20 PM

This IMU does not have a magnetometer/compass. It cannot determine your horizontal heading.

Reply

can

NOVEMBER 9, 2020 AT 8:32 AM

Hi,

Is it possible to get position (displacement) data from mpu6050?

Reply

Kai Horstmann

NOVEMBER 10, 2020 AT 2:21 PM

Can you explain what you mean with “displacement”?

Reply

DIB_gnc

NOVEMBER 10, 2020 AT 2:43 PM

Hello,
really it’s a great job, and I appreciate ur effort and ur work
I’ve a question about matrix, how did u chose them ?
and coeficients in Kalman filter always ?
thank u in advance

Reply

Alain

NOVEMBER 21, 2020 AT 1:57 PM

Hello Kai,
I was just looking at your article because I tried to implement roll angle measure on a motorcycle with a MPU6050. Everything works fine on a table but on the motorcycle, that's another story.
At the moment I just used a complementary filter which gave very bad result.
In fact, I saw the roll angle increasing at the begining of the turn then, coming back to a intermediate value between true angle and 0, maybe depending on the speed and so, to the z acceleration (body frame).
One thing I noticed is that when I give a roll (or pitch) angle to the MPU6050 and then apply movements along z axis, the given angle changes significantly. I was wondering if it comes from bad quality of the MPU6050, or if it's a normal behavior that gyro gets significantly disturbed by linear accelerations..
What do you think.

Reply

Alex

DECEMBER 21, 2020 AT 12:12 PM

Hi Alain, I tried to use the complementary filter, in my case for estimating the movement of a camera (pitch roll). The result was also very bad for my case. The real value is for example 0.3 degrees and the filter gave something like 2 degrees witch is very big. My guess is that this is because I have big accelerations on my system.
The camera's movement is like a cosine, so the system is under linear accelerations consciously. If you try to estimate the angle using the acceleration value, it is assumed that the linear acceleration is 0. I believe that the complementary filter is not useful for my case.

Do you agree? Do you have the same issues? I want to use the Kalman Filter, but still struggling with it

Reply

Simon

MARCH 30, 2021 AT 8:06 PM

Hi

Very nice article great work this is very helpful to me but I have a little question at the beginning of your article, you said the values coming from the gyroscope are in deg/s but if I read your code on GitHub get_gyro() function, you return values in rad/s why? All math formula works with deg/s ans rad/s?

Reply

Erol

APRIL 1, 2021 AT 10:28 AM

Hi, Philip. Thanks and congrats for this nice work. I have a question about your Kalman covariance parameters, I mean diagonal covariance matrices (P,Q,R) factors/coeff.s.

I work with my imu sensor data, when I decide to try different initial state values and change P-value, I noticed no change in the results. And then tried different Q and R values, no change for the results either.

Isn't it odd? or do I miss something? Thanks, Erol.

Reply

Chuong Hoa Loc

APRIL 19, 2021 AT 10:35 AM

Did you make mistake, vector v_tilde is vector u_tilde ?

Reply

Randy

APRIL 20, 2021 AT 2:14 AM

Hi Philip,
I have the same question as @HD. The transform matrix of angular velocity is weird for me too. Is there any document that can explain this transform? I'm wondering how this can be gotten.

Reply

AJ

JULY 29, 2021 AT 2:56 AM

Transform/transfer matrix depends on the sequence of rotation, and coordinate system. Euler rotations could be Rxyz, Rzyx, or many combinations. Similarly, coordinate systems can be NED (North East Down frame) or various other frames I cannot remember the other names. For example, in the NED frame, you have positive Z-axis pointing down. Also, to add more confusion, you generally use a hand rule. I.e right or left-hand rotation. Generally, anti-clockwise rotation is positive. I hope this helps!

Reply

AJ

JULY 29, 2021 AT 2:48 AM

Hey Phil
Thanks for this fantastic work. I have tested the complementary filter vs the Kalman filter with the CSV data you provide, and it works. I was wondering if it is okay to use your code in my research paper? I am working on Raspberry pi and IMU.
Many thanks
AJ

Reply

Leave a Reply

Comment

Name *

Email *

Website

Post Comment

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

philsal.co.uk