# Checkpoint 1: Attitude Estimation, Digital Filtering, Sensor Fusion

## Part 1: Attitude Estimation

Section 1.1: Conceptual Pitch Estimation

The pitch angle of a quadcopter can be estimated by using on-board sensors. The two sensors used to make this estimate are the accelerometer and the rate gyro sensors. The accelerometer can estimate pitch angle by calculating the angle relative to the g-vector using the xyz components of the accelerometer. Since the g-vector always points down, the angle between the forward vector and the g-vector will determine the forward and backward tilt (pitch).

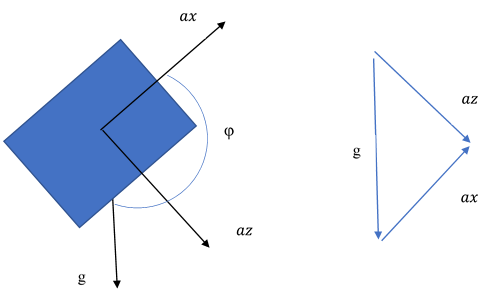


Figure 1. Pitch Angle Estimation

Figure 1 shows the pitch angle in relation to the acceleration vectors. Because ax and az are orthogonal, the pitch angle can be calculated with the following equation:



Depending on the signs of ax and az, this equation cannot account for what quadrant the angle is in. In order to account for this, the function atan2 is used instead.



The rate gyro sensor estimates the pitch by measuring angular velocity about the xyz axes. The equation for angular velocity is as follows:



In order to get the pitch angle from the angular velocity of the rate gyro, the x component of the angular velocity must be integrated with respect to time. Using the trapezoid rule will be an accurate enough integration technique for this application.



Section 1.2: Practical Pitch Estimation

Accelerometer and Rate Gyro Pitch Angle Estimate C++ code:

pitchAcc = (atan2(a\_mpu[0],-a\_mpu[2])\*(180/PI));

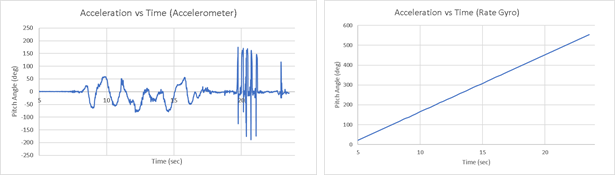
pitchGyr = pitch0 + 0.01\*(((1-g\_mpu[1])+g\_old)/2)\*(180/PI);

//0.01 is dt

g\_old = 1\*g\_mpu[1];

pitch0 = pitchGyr;

The following two figures show pitch angle estimates versus time for both the accelerometer and rate gyro sensors using the C++ code above.

Figure 2. Pitch Estimate (Accelerometer) Figure 3. Pitch Estimate (Rate Gyro)

In the above figures, the accelerometer pitch estimate looks noisy compared to the actual motion of the quadcopter, and at around 20 seconds the accelerometer pitch estimate shows much larger angles than the actual motion. This noise is due the sensitivity of the accelerometer to vibrations and rapid movements. The rate gyro pitch estimate drifts because integration increases uncertainty over time.

## Part 2: Digital Filtering

Section 2.1: Conceptual Digital Filtering

Sensors are great for collecting data from the real world, but a common problem is they can pick up on an additional information that is not relevant to the application. In this case, it can be helpful to employ digital filtering to eliminate the aspects of a signal that are not useful. The idea behind filtering is that once a signal is transformed into the frequency domain, it is possible to differentiate between the useful information and the noise. Filters are mathematical operations done to the data that attenuate signals depending on their frequency in the data. Different filter types can be selected based on the kind of information being filtered and the characteristics of the signal.

Finite Impulse Response (FIR) filters involve simple manipulation of an input signal to generate the output. FIR often needs a higher order of computations which can be a burden to process in real time. There are two main benefits of FIR filters. First, the filter does not require any feedback, meaning that the previous filter data does not need to be fed back into the system to produce the desired signal. The other benefit to FIR filtering is that this technique will always yield a stable solution.

A typical example of a FIR filter is the Moving Average. In this method, previous data points are averaged to generate the response. As a new data point is sent to the filtering function, the oldest data point is dropped from the calculation. This means that only the most recent set of points take part in generating the output.

Infinite Impulse Response (IIR) filters use the output from previous computations to determine how to handle the new input. This method of feedback ends up producing a filter that generates a response even after the signal has stopped. This also has the effect of producing a phase lag that gets worse with the higher the order of filter being used. Fortunately, these filters do not need to be overly complex to generate results as good or better than a comparable FIR.

Named filters belong to the IIR category, including Butterworth, Chebyshev, Bessel, and Elliptic. In a Butterworth filter, coefficients are selected based on a desired cutoff frequency and the sampling frequency for the dataset. A new output point is generated with a combination of coefficients, previous input values, and previous output values.

There are 4 main types of filters. A high pass filter removes low frequencies and allows high frequency information through. A low pass filter removes high frequencies, allowing low frequency signals to pass. Band pass filters remove both high and low frequencies, so that only signals between specified frequencies get through. A notch filter works opposite from the band pass, where a range of frequencies are attenuated.

Section 2.2: Implementing a digital filter in C++

#include <cmath>

#define PI 3.14159265

using namespace std;

float butter2 (float f\_c, float f\_s, float dat\_in, bool highpass)

{

/\*

2nd order butterworth filter for high or low pass data filtering

this function is designed to be called within a

computational loop and the idea is that it stores each input

value to build and adjust the filter each run

f\_c = cutoff frequency

f\_s = sampling frequency

dat\_in = value to be filtered

highpass = if true, use high pass coefficients

\*/

// these coefficients are going to be the same

float gamma = tan((PI\*f\_c)/f\_s);

float coef\_d = pow(gamma, 2) + pow(2, 0.5)\*gamma + 1;

float a\_1 = (2 \* (pow(gamma, 2) - 1)) / coef\_d;

float a\_2 = (pow(gamma, 2) - pow(2, 0.5)\*gamma + 1)/coef\_d;

// set highpass or lowpass filtering coefficients

float b\_0, b\_1, b\_2;

if (highpass == true)

{

b\_0 = 1;

b\_1 = -2;

b\_2 = 1;

} else {

b\_0 = (pow(gamma, 2)) / coef\_d;

b\_1 = (2 \* b\_0) / coef\_d;

b\_2 = b\_0;

}

// hopefully these static declarations store the values

// so we don't overwrite the coefficients each time the loop

// hits this part.

static float x\_n1 = 0;

static float x\_n2 = 0;

static float y\_n1 = 0;

static float y\_n2 = 0;

// calculate the new filtered value, then cycle variables

float y\_n = b\_0\*dat\_in + b\_1\*x\_n1 + b\_2\*x\_n2 - a\_1\*y\_n1 - a\_2\*y\_n2;

x\_n2 = x\_n1;

x\_n1 = dat\_in;

y\_n2 = y\_n1;

y\_n1 = y\_n;

return y\_n;

}

The Butterworth filter function takes a sampling frequency, a cutoff frequency, and an indicator that determines whether the filter is high pass or low pass. The sampling frequency is determined by the sampling rate for data fed into the raspberry pi. Ideally, the cutoff frequency will be at a very different rate from the expected frequency of data points. In practice, however, the sensors generate a cleaner, more useful signal when the cutoff frequency is closer to 1Hz. This value can be tuned through trial and error until the data is acceptable. For the experimental data, a sampling frequency of 100Hz was used. A compromise must be made between performance and accuracy, so the above function is for a 2nd order filter. This is not as accurate, and the transition band is wider than higher orders, but the time delay is not as substantial.

## Figure 4. Filtered vs Unfiltered (Gyroscope) Figure 5. Filtered vs Unfiltered (Accelerometer)

## 

## In Figure 4, the gyroscope data is shown before and after filtering. The raw data drifts to several orders of magnitude away from possible real pitch angle values. This drift is low frequency information that skews the results away from zero. After high pass filtering, the gyroscope data only deviates when the pitch is changing very fast.

Figure 5 shows the effect of filtering on the accelerometer. The fuzzy high frequency noise is transformed into a smooth line that closely follows the actual movement of the quad after low pass filtering. This graph also shows the time delay associated with Butterworth filtering.

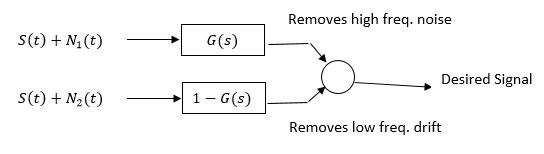
## Part 3: Sensor Fusion

Section 3.1: Complimentary Filter Theory

A complimentary filter combines the output from two or more sensors to get a better estimate of the physical state of the system than any of the sensors is able to detect on its own. The primary requirement for this technique is that each of the sensors used complements each other so as to make up for the shortcomings of each measurement method. These filters are easy to implement and do not need a dynamic model to tune, but they are not always the most accurate.

These filters operate on the principle that a filter output consists of a signal term, with sensor noise and filtering terms from each sensor in the system. In the case of a quadcopter, and accelerometer and a gyroscope are used to generate a roll estimate. The accelerometer generates high frequency noise that makes the raw data plot look jagged, while the gyroscope produces a low frequency noise that shows up as drift. Each sensor is good at detecting a specific type of motion. A complementary filter combines the useful information from both sensors while eliminating the noise by collecting the filtered sensor output from the portion of the frequency domain where the sensor is the most effective.

To use a high pass and a low pass filter together, both filter parameters must use the same cutoff frequency. This is to prevent holes in the frequency domain. If the cutoff frequency of the low pass filter is below the high pass cutoff, undesirable noise gets transferred to the output. In the opposite situation where the low pass cutoff is above the high pass cutoff frequency, the filters prevent some of the useful information from being recorded.



Section 3.2: Implementing a Complimentary Filter

In practice, applying this technique to a gyroscope and accelerometer pitch estimate is done by collecting the sum of the filter outputs from both sensors. To accomplish this in C++, a structure has been created to handle Butterworth filtering parameters for multiple devices.

#include “Navio/digital\_filter.h”

//variable declarations

//takes arguments(order,type,cutoff\_freq,sample\_freq)

digital\_filter g\_filt(2,’h’,1,100);

digital\_filter a\_filt(2,’l’,1,100);

//complementary filter loop commands

c\_gyro\_filt = g\_filt.filter\_new\_input(pitchG);

c\_acc\_filt = a\_filt.filter\_new\_input(pitchA);

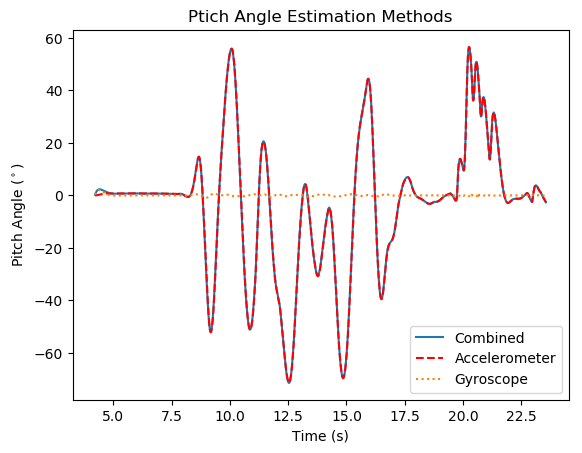
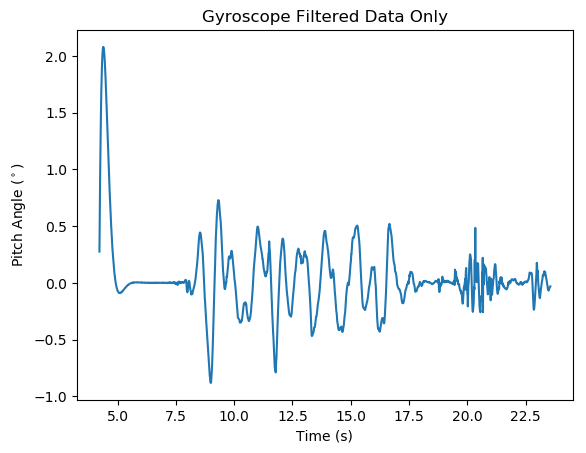
comp\_filter = c\_gyro\_filt + c\_acc\_filt;

Figure 6. Pitch Angle (Accelerometer, Gyro, &Combined Filter) Figure 7. Pitch Angle (Soley Filtered Gyro)

Figure 6 shows each filtered sensor output and the resulting complimentary filter results. The accelerometer does the bulk of the estimating. The gyroscope data looks almost completely flat, but in the code, the values should be adding together. By looking at the gyroscope filter output alone, Figure 7 shows that the greatest gyro data contribution (after initialization) was less than 1 degree of pitch angle to the overall estimate.