UPPSALA UNIVERSITET

Examensarbete 15 hp
Juni 2018

A control system for a 3-axis camera stabilizer

Ali Algoz
Bakhtiyar Asef Hasnain

Abstract

A control system for a 3-axis camera stabilizer

Ali Algoz, Bakhtiyar Asef Hasnain

The purpose of the project is to implement a control system for a 3-axis camera stabilizer. The stabilization is done by controlling three blushless DC motors driving the yaw, pitch and roll movements of the camera stabilizer's frame, respectively. The stabilizer's frame (equipped with three motors) is used in this project, and it is directly taken from a commercial product, Feiyu Tech G4S. The control system concerned in this project consists of a Teensy 3.6 microcontroller unit (MCU) implemented with three PID controllers, the motor drivers to drive the three motors, and an inertial measurement unit (IMU) of 9 degrees of freedom.The MCU is also used to process the IMU angle measurements of the camera position in 3-axis motion, in particular, it converts the IMU raw data to an angle for each of the axis, it then processes the angle data using a Kalman filter to reduce the noise.
In the end of the project a prototype has been built and tested, it uses the control system to run the stabilizing process. It is shown to work quite successfully. In particular, it can run smoothly in the roll and pitch axes and compensate for unwanted movement, however the yaw axis does not function as intended due to a misplacement as well as poor calibration process of the magnetometer sensor in the IMU, which is left for future work.

Handledare: Håkan Sjörling
Ämnesgranskare: Ping Wu
Examinator: Tomas Nyberg
ISRN UTH-INGUTB-EX-E-2018/07-SE
Tryckt av: Uppsala

Populärvetenskaplig sammanfattning

Kamera stabilisatorer har använts under längre perioder av stora filmbolag men har på senare år allt mer vanliga bland vanliga konsumenter. Användningen av stabilisatorer har gått från att användas på större filmkameror till små mobiltelefoner med kameror. Detta projekt fokuserar på en fördjupad undersökning av reglersystemet hos en 3-axlad kamera stabilisator för en mindre typ av action kamera. Syftet med projektet är att skapa ett komplett kontrollsystem för stabilisering av video samt att undersöka signalbehandlingen för att uppnå ett stabilt system.
Stabiliseringen utförs genom att kontrollera tre separata motorer i tre olika led, dessa motorer är en del av konstruktionen som kameran är fäst på. För att kontrollera motorerna används en mikrokontroller av typ Teensy 3.6. Mikrokontrollen används för att underlätta kommunikationen samt programmeringen av olika enheter. I detta projekt används mikrokontrollen för att hantera data som läses av från en rörelsekänslig sensor för att sedan räkna ut kamerans positionering. Rörelsesensorn består av tre olika sensorer som känner av gravtiatonen, accelerationen samt vinkelhastigheten. Informationen från rörelsesensorn kan sedan bearbetas för att få ut önskad information. Denna typ av sensor används exempelvis inom flygteknik, mobiltelefoner och robotteknik. Informationen från sensorn kan ofta vara väldigt brusig och bör därför filtreras. Två olika filter har undersökts i detta syfte för att uppnå den mest noggranna signalen som representerar positioneringen av sensorn. Dessa filter var ett komplementär filter där två olika signaler sammanfogas till en samt ett Kalman filter.
Projektet resulterade i en prototyp modell som kan stabilisera video till en viss del. Två utav tre axlar kan kompensera för oönskade rörelser och stabilisera videobild som planerat. Den tredje axeln fungerade under kortare körningar men blev opålitlig under längre tid. Detta orsakades av både missplacering av rörelsesensorn samt en icke optimal kalibreringsprocess av magnetometern som känner av gravitationen. Det filter som valdes att användas var Kalman filter på grund av dess förmåga att filtrera signalen på ett bra sätt. Resultatet visade även att ett simpelt komplementär filter skulle kunna appliceras för detta ändamål och få liknande resultat med mindre beräkningskraft.

Table of contents

Vocabulary

- IMU – Inertial measurement unit
- PID controller – Proportional-integral-derivative controller
- MPU – Micro processing unit
- MCU – Micro controller unit
- SPWM – Sinusoidal pulse width modulation
- SVPWM – Space vector pulse width modulation
- Feiyu-tech G4S – Commercially available product
- Gimbal – Active camera stabilizer
- BLDC – Brushless DC
- I$_2$C – Inter-integrated communication
- SDA – Serial data line
- SCL – Serial clock line
- C++ – Programming language
- IDE – Integrated development environment

1.Introduction

1.1 Background

Active stabilization has been around for a very long time especially in the film industry where the application has been used to stabilize video footage. Today camera stabilization has been made easily accessible for indie filmmakers and regular people so that you can simply attach your smartphone (that has a camera) on a three-axis gimbal and get smooth video footage relatively cheap.

Recently people have also been implementing the idea behind active stabilization in high end suspension systems in automotive vehicles like cars and motorcycles. The suspension is electronically stabilized in the same manner as camera stabilization.

1.2 Purpose and goals

The purpose of this project is to investigate how a 3-axis gimbal stabilizer works and then create a self-made control system to control a gimbal frame from a commercial product, Feiyu tech G4S. The control system includes a Teensy 3.6 microcontroller with the Arduino add-on, a motor driver circuit, and a motion sensor with 9 degrees of freedom. The measurements from the sensor is filtered with a complementary filter or Kalman filter.
The goals of this project are to implement the control system that provides a functioning gimbal stabilizer.

1.3 Tasks and scope

To achieve the above-mentioned goals, the project includes the following tasks

- Learn more advanced control theory dealing with 3-axis movements and programming a microcontroller
- Build a motor driver to drive three motors for 3-axis movements
- Implement a PID controller to control the motors for 3-axis movements, pitch, roll and yaw to stabilize the camera.
- Implement a filter to process a noisy signal.

Because of the ten-week timeframe, an already existing mechanical construction and handle for the gimbal was used, and the focus of the project is on the above-mentioned tasks.

1.4 Outline

This thesis is divided into 6 main chapters. Chapter 2 explains all the control theory needed to understand this project. Chapter 3 explains the methodology and how the different components and software were used to create the finished assembly. Chapter 4 presents the results that were acquired in the end. Chapter 5 discusses the results and why the project ended with the results it did. The last chapter, 6 is where the conclusion of this project and the future work that would be needed to get a better result.

2. Control Theory

2.1 Camera stabilization

An active camera stabilizer typically uses three motors to stabilize a camera in the correct orientation and keeps it balanced using a complex electronic control system. The control system often consists of a motion sensor (IMU) and a microcontroller or an electronic control unit (ECU). The motion sensor registers the angular velocity, acceleration and the magnetic force to determine the angle of the camera and communicates with a microcontroller which controls each of the motors on the three axes on the gimbal independently. The goal of the stabilization process is to eliminate unwanted camera movement by driving the separate motors in the opposite direction of the unwanted movement. The three axes on the gimbal are represented as yaw, pitch and roll which are perpendicular to each other.

2.2 Automatic Control

Camera stabilization can be realized using a feedback control system. A feedback control system in general as shown in Figure 2.1 consists of a regulator (controller), an actuator and a process as well as a sensor on the feedback. For the camera stabilization, the camera is the process to be stabilized by a regulator, which is a proportional-integral-derivative (PID) controller, through an actuator (3 motors driving roll, pitch and yaw movements, respectively) and an IMU (inertial measurement unit) sensor for the output measurement. The PID-controller has the input of the error i.e. how much the current position deviates from the desired position and corrects it to make it zero. The correction is based on three different constants, P – proportional gain, I – Integral gain and D – derivative gain. The three different gain values of the PID will determine how responsive the system will be [1].

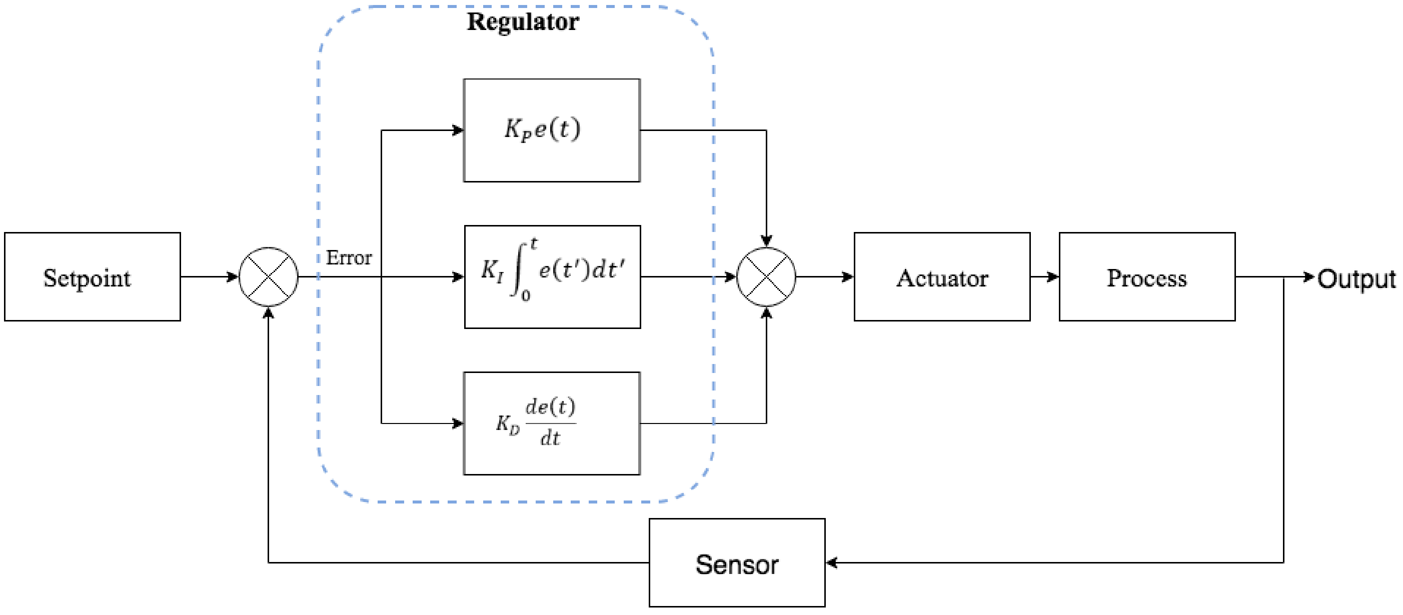

Figure 2.1 Block diagram showcasing the PID regulator

The proportional part calculates the error between the desired output and the actual measured value. The proportional parameter is referred to as K_P which determines the response to errors.

The integrating part sums up the previous error to add on the output to further stabilize around the desired setpoint. The main purpose of the integrating part is to eliminate the steady state-state error and the parameter is referred to as K_I

The derivative part compares the previous error to the current one to determine the rate of change and dampens the system reducing oscillations and overshooting. The main purpose of the deriving parameter is to improve stability and speed and is referred to as K_D.

The output of a PID-controller is described in Eq. (2.1).

$$u(t) = K_P e(t) + K_I \int_0^t e(t')dt' + K_D \frac{de(t)}{dt} \tag{2.1}$$

2.3 Filtering

Filtering is a process where unwanted parts of a signal is excluded or dampened to some degree. There is a large number of different filtering methods available for both digital and analog systems. In this project two types of digital filtering methods are used to get a more accurate reading from the IMU sensor [2].

2.3.1 Complementary Filter

A complementary filter is a filtering method frequently seen in applications where two different types of values are needed to create a smooth single variable that consists of the two variables fed into the system. The basic concept in this project is that the accelerometer data is processed through a high pass filter and the gyroscope data through a lowpass filter and then adding these two components together. The simplified formula written as

$$\theta_k = \alpha(\theta_{k-1} + \omega_k \Delta t) + (1 - \alpha)a_k \tag{2.2}$$

where θ is the estimated angle, ω is the angular velocity from the gyroscope, a is the angle calculated from the accelerometer data and $\alpha = \frac{T}{\Delta t} / \left(1 + \frac{T}{\Delta t}\right)$, T determines the cutoff frequencies and Δt is the time between each new value. This method of filtering is often referred to as data fusion [3].

2.3.2 Kalman Filter

Kalman filter is filtering method that is widely used in navigation, control of vehicles, aircrafts, robotics etc. Its basic function is to find the best estimate of a signal from the signals observation which is noisy. The state of the system is expressed by

$$x_k = F_k x_{k-1} + B_k \dot{\theta}_k + w_k \tag{2.3}$$

where x_k is a state vector that contains the term of interest at a certain time k. The vector θ_k contains any control input in the case of this project it will be the gyroscope measurements. F_k is the state transition matrix and B_k is the control input matrix. The last vector w_k contains the process noise. The measurement (or observation) z_k of the true state x_k is given by

$$z_k = H_k x_k + v_k \tag{2.4}$$

where z_k is the vector of measurements, H_k is a transformation matrix which maps true state space into observed space, v_k is measurement noise. The filter involves two stages. The first stage being prediction step is given by

$$\hat{x}_{k|k-1} = F\hat{x}_{k-1|k-1} + B\dot{\theta}_k \tag{2.5}$$

This part of the filter will estimate the current state of the system based on the previous gyro measurements. An estimation of the a priori error covariance matrix $P_{k|k-1}$ is then made to confirm the trustworthiness of the current estimated state, in the following manner.

$$P_{k|k-1} = FP_{k-1|k-1}F^T + Q_k \tag{2.6}$$

The next step in the Kalman algorithm is the to make the measurements and update the variables. This is done in the following manner.

$$\hat{y}_k = z_k - H\hat{x}_{k|k-1} \tag{2.7}$$

which is called the innovation. This part calculates the difference between the measured value z_k and the previous value $\hat{x}_{k|k-1}$.
After this the calculations for the innovation covariance is made:

$$S_k = HP_{k|k-1}H^T + R \tag{2.8}$$

Here the algorithm predicts how trustworthy the measurements are based on the previous error covariance matrix $P_{k|k-1}$ and the measurement covariance matrix R. If there is a significant amount of noise in the measurement it leads to a high S_k which means that the incoming values are not trustworthy. The next step is to calculate the Kalman gain which is used to indicate how trustworthy the innovation is.

$$K_k = P_{k|k-1}H^T S_k^{-1} \tag{2.9}$$

After this the posteriori estimate of the current state is updated:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k\hat{y}_k \tag{2.10}$$

This equation corrects the estimate of the a priori state $\hat{x}_{k|k-1}$ that in this project was calculated with the gyroscope measurements with the measurements from the acceleration measurements. The final step of the algorithm is to update the posteriori error covariance matrix:

$$P_{k|k} = (I - K_kH)P_{k|k-1} \tag{2.11}$$

In this final step the filter self-corrects the error covariance matrix based on corrected estimate.
In summary this filter estimates the current value based on its previous measurements and estimations and then correcting itself accordingly based on how trustworthy the different measurements are. The noise variance variables are usually found through different methods but in this project these variables are chosen to give a more reliable reading. These values can be changed to give different weights to the sensor data making the filtered result either follow the data more accurately or smooth out the signal. Setting more weight to the accelerometer leads to a more accurate but noisy measurement, giving the gyroscope more weight yields a smoother but slower signal [4].

3. Implementation

3.1 Overview and planning

In the beginning of the project a lot of time went into studying how an active stabilizer functions and its separate components. All the information was gathered online through academic articles, forums and videos through the YouTube platform.

By studying how complex the system of an active camera stabilizer it was decided to eliminate the process of making a mechanical construction of the gimbal and use an existing one due to lack of time. An existing commercial gimbal was disassembled and the mechanical construction along with the motors was utilized in the project. The choice of components that was used in the project was made based on previous similar projects and on the recommendation of the supervisor of the project. The programing of the microcontroller Teensy 3.6 was done in Arduino IDE software and a few libraries like Kalman, MPU9250 and EEPROM was implemented to facilitate the programing.

Each system of the project was tested separately using a breadboard and very basic connections to ensure that the systems worked properly. When testing was done all the necessary components like the motor driver circuits, Teensy microcontroller and the wiring was soldered onto one single experiment card and the separate programs was merged into one.

The block diagram in figure 3.1 describes the complete feedback system simplified to give an overview of the stabilization process. The green block represents the microcontroller, which describes the process inside the software.

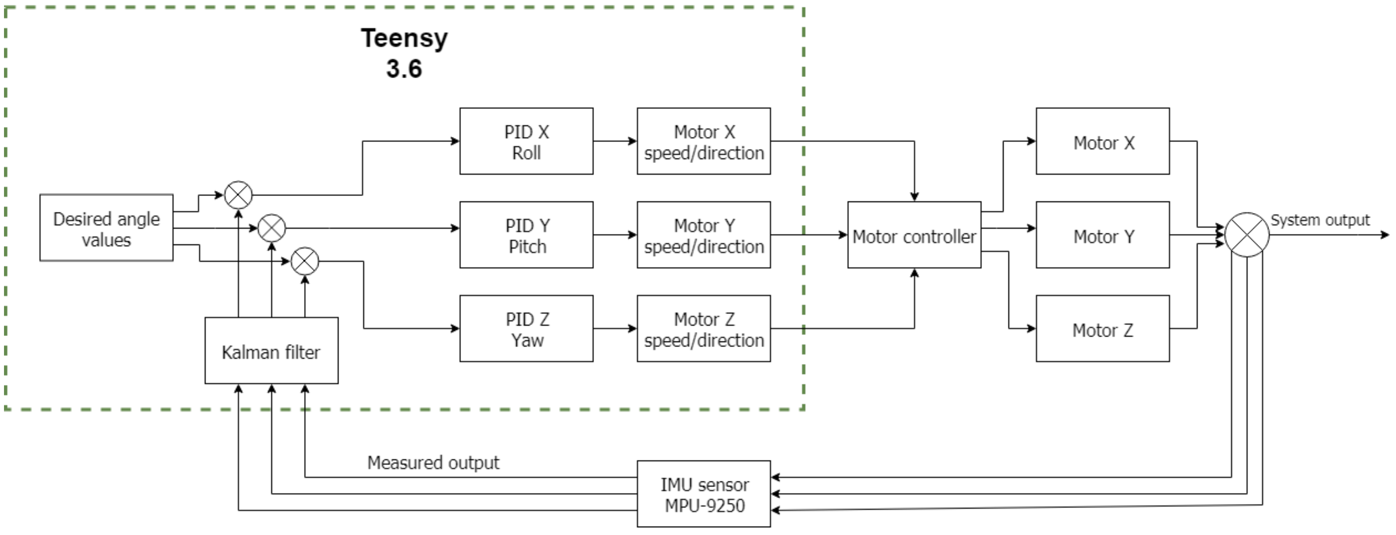

Figure 3.1 Block diagram of the complete system

3.2 Hardware and components

3.2.1 Gimbal frame

The gimbal frame consists of a case to store the microcontroller unit as well as the motor driver circuit and a mechanical construction to hold up each motor and the camera. The construction of the handle and the camera frame is re-used from the commercially available gimbal, Feiyu Tech G4S. The wiring from each motor driver runs through the handle into each separate motor. The placement of the sensor was mounted under the camera parallel with the lens to get the best result of the camera angle. A picture of the camera frame can be seen in figure 3.2.

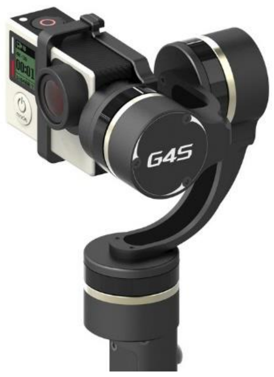

Figure 3.2 Original FeiyuTech G4S gimbal frame

3.2.2 MCU – Teensy 3.6

Teensy 3.6 is a microcontroller board based on USB-connection that features a 32-bit 180 MHz ARM Cortex-M4 processor with a floating-point unit. The more commonly known Arduino Uno was the initial microcontroller to be used however it was advised that the Teensy would offer better computational power. The microcontroller allows the user to program and manipulate input and output data through the Arduino IDE programming software [5].

Figure 3.3 Teensy 3.6 microcontroller board

3.2.3 IMU Sensor MPU-9250

An inertial measurement unit also known as an IMU is a device that is used to acquire different positional data such as angular velocity, acceleration and magnetic field strength throughout a 3D space. These devices are useful in a variety of task but are most often seen used to maneuver aircrafts and UAVs. To acquire these three positional values the IMU needs three different types of sensors called gyroscope, accelerometer and magnetometer. Using the data these three sensors provide it is possible to determine the angle of the IMU in 3D space which is useful when trying keep whatever the IMU is attached to leveled to a plane [6].

Figure 3.4 MPU 9250

Accelerometer: The accelerometer in the IMU measures proper acceleration. This is the acceleration of a body in its own rest frame. Which means that the sensor measures the acceleration of its own internal components to describe the acceleration of the sensor. The accelerometer uses internal components which resembles a chamber where a resting seismic mass is located. The mass then moves around in this chamber giving electrical responses proportional to the acceleration in each axis. This is shown in figure 3.4. This type of sensor would ideally give a sensor value of 9.8 m/s^2 in one axis and 0 m/s^2 in the other two axes when placed on a flat surface on earth.

Accelerometers of this sort can measure both static and dynamic accelerations and are frequently used in applications where orientation information relative to earth's gravity is needed. This is commonly seen in smartphones where the orientation data is interpreted and for instance used to determine which way to display an image.

The IMU can output the accelerometer data in different ways depending on the type of IMU, however a digital output was used for the task in this project. This has the benefit of having less noisy outputs and being more feature rich than its analog counterparts.

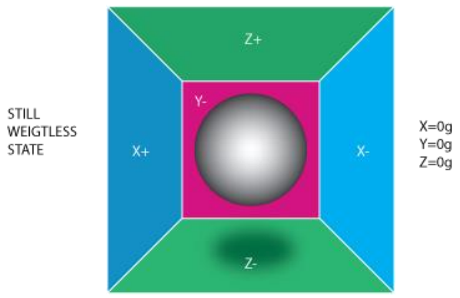

Figure 3.5 Inside look of an accelerometer circuit

Gyroscope: The gyroscope is used to measure the angular velocity about each of the three axes. It is to be noted that one spinning axis does not affect the other, meaning if the sensor were to be placed on a flat surface and spun it would only display a rate of change about the z-axis. The microelectromechanical version of a gyroscope uses a resonating mass that is shifted as the angular velocity changes. The small movements are then proportionally converted to electrical signals that are translated by the IMU to readable values [7].

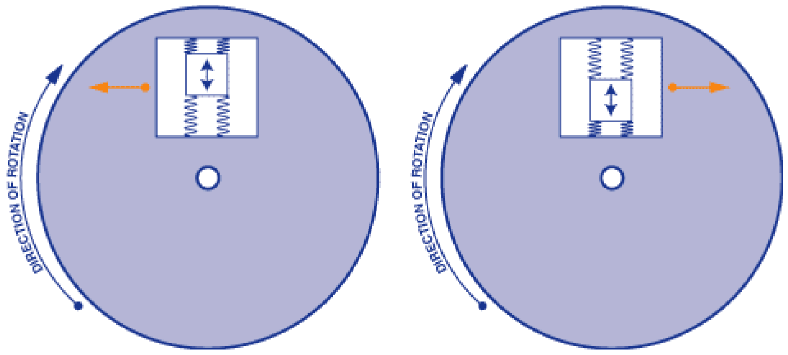

Figure 3.6 Demonstrates a simple gyroscope construction

Magnetometer: There are many ways of sensing the magnetic field but the most common used in solid state devices is a hall effect sensor. These types of sensors produce a voltage proportional to the magnetic field applied that are later translated to readable values from the IMU.

3.2.4 Motors and driver

3.2.4.1 Brushless DC motor

A three-phase brushless DC motor referred to as BLDC motor are also known as synchronous DC motors and are powered by AC electric current which is produced by a switching a DC power supply or an inverter. The advantages of a BLDC motor are the efficiency and the constant torque output the motor delivers but also the level of control it offers. The durability and the low weight of BLDC motors are also advantages over other types of motors.

The rotor of a BLDC motor is a permanent magnet and the stator consists of several coils. By applying DC power to the one of the coils at the time, the coils will sequentially energize and create a rotating electric field that rotates the rotor. The rotational electric field is created by energizing different coils in a special order depending on the position of the rotor. A typical BLDC system requires a six-step commutation sequence for one electrical revolution. The energizing sequence of the coils is described in the figure below.

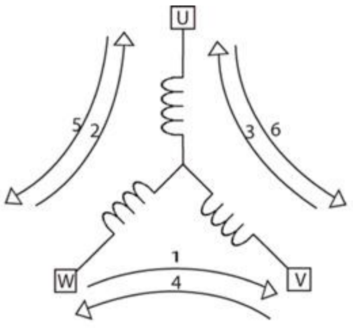

Figure 3.7 Energizing sequence for a three-phase BLDC motor

To control the energizing sequence a feedback system is often implemented to determine the position of the rotor. Introducing three hall-effect sensors to the system is a popular way to establish the position of the rotor. The hall-effect sensors can sense the change in the magnetic field and anticipate when the different polarities of the rotor pass by the sensor. The signal generated by the hall sensors lets the control unit know which phase to drive (the phases u, v and w seen in figure 2.1) to create the rotation in a BLDC motor [8].

3.2.4.2 Pulse width modulation

Pulse width modulation referred to as PWM is a frequency dependent square wave. The PWM square wave is limited to a ON- and OFF-state, often described as high or low. The duty-cycle of a PWM signal describes the relationship between how long the signal is high or low and can be set from 0 to 100 % where 100 is the maximum power output. The PWM square wave is often generated by using a microcontroller which can manipulate the duty-cycle through software or by integrated circuits [9].

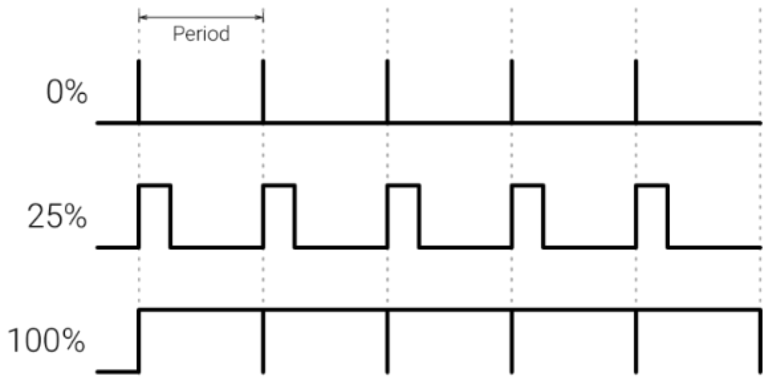

Figure 3.8 PWM signal with 0, 25 and 100 % duty cycle

Sinusoidal pulse width modulation (SPWM) is a technique used to generate AC electric current by switching the duty-cycle from high to low in a manner that follows a sinusoidal wave. The mean value of the total output of the generated PWM can be seen in the figure below

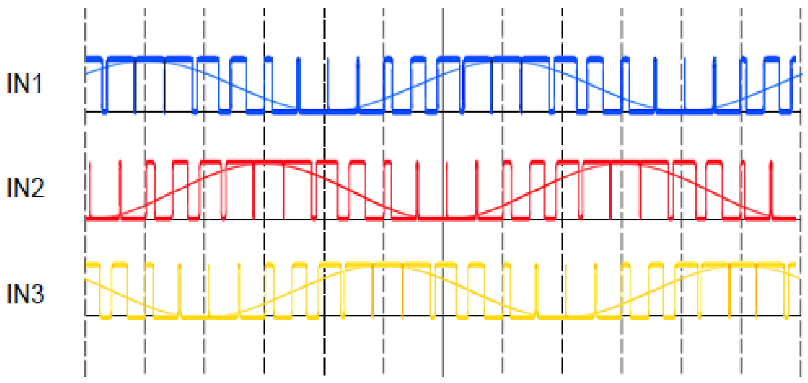

Figure 3.9 Three sinusoidal waves generated with 120° phase shift using PWM

3.2.4.3 Motor driver L298N

The motor driver L298N [10] is a power electronic circuit that consist of a dual h-bridge which can be seen in figure 3.6.

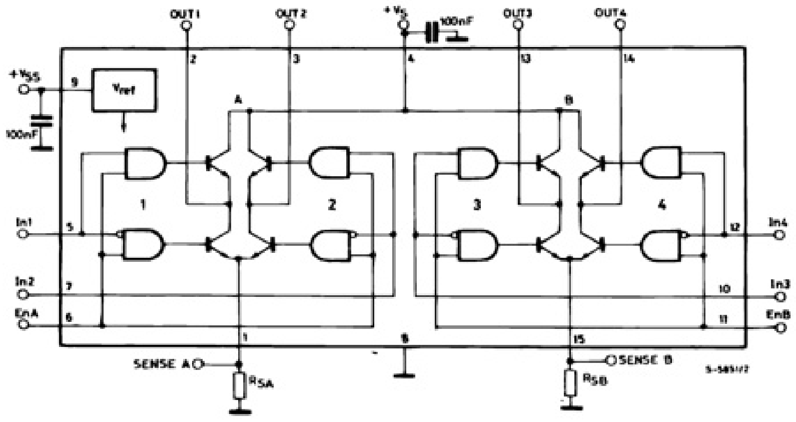

Figure 3.10 Schematic of the motor driver L298N

A h-bridge is an electronic circuit that is often used as a motor driver. The design of the h-bridge is usually built on different types of transistors, common in motor drivers is to use mosfets. The h-bridge allows a motor to run in the opposite directions by feeding the circuit voltage from a power source and using a PWM to control the gates on the transistors. The figure below describes the operation to run the motor in different directions by opening and closing the separate transistors.

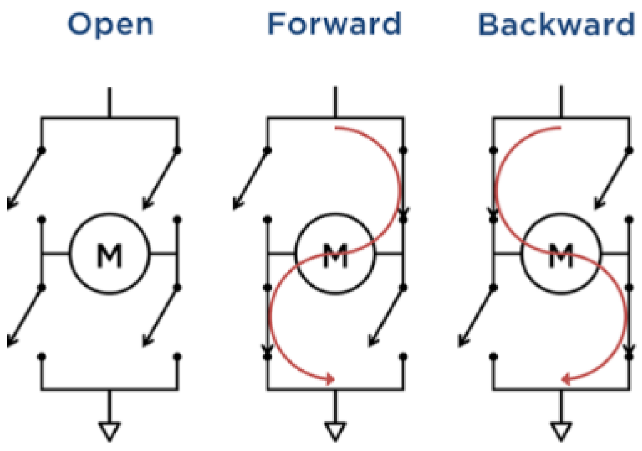

Figure 3.11 Controlling the direction a motor using a H-bridge

3.3 Software and libraries

The Teensy 3.6 is programmed through the Arduino IDE which requires a Teensyduino plug-in found on the PJRC website [11]. This IDE allows for easy uploading to the microcontroller once a program has been written inside the software. The programming language used to program the Teensy is therefor the same as on the Arduino which is C++. The Arduino IDE also makes it possible to use preexisting libraries created by other people to be quickly implemented in the user's programs. There are libraries for a variety of different use cases, they often contain complex mathematical formulas, register handling code, filters among other things. Using these preexisting libraries can make the users program a lot more compact and easier to understand.

3.4 Implementation

3.4.1 Prototype

A prototype has been constructed as shown in figure 3.12. The complete control unit, including the teensy and the three motor driver circuits soldered onto an experimental card can be seen in figure 3.13. The schematic for the circuitry including one motor driver for each BLDC motor, the microcontroller and the IMU sensor can be seen in figure 3.14.

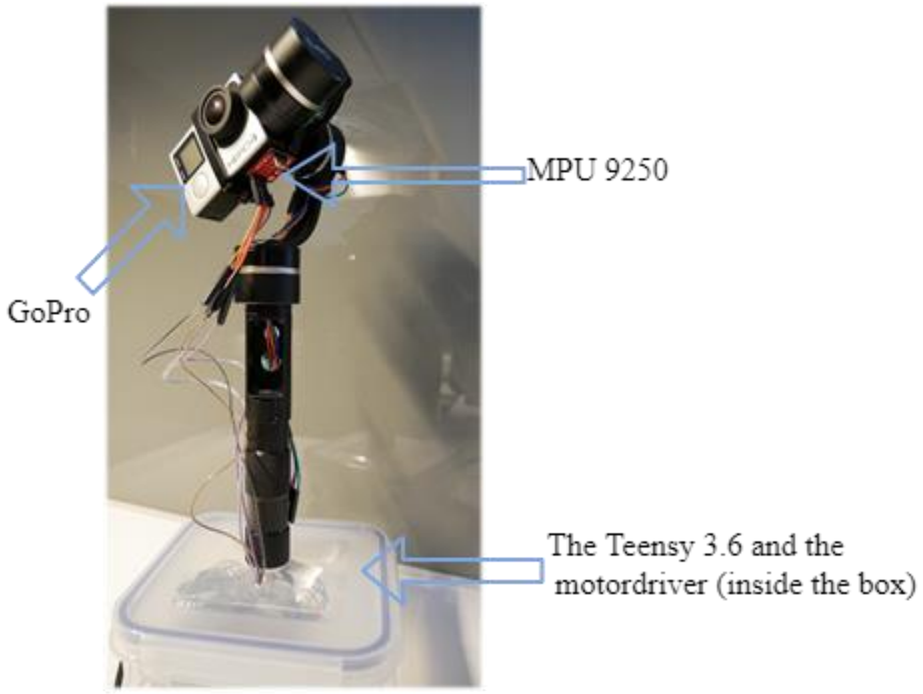

Figure 3.12 Complete assembly of the camera stabilizer

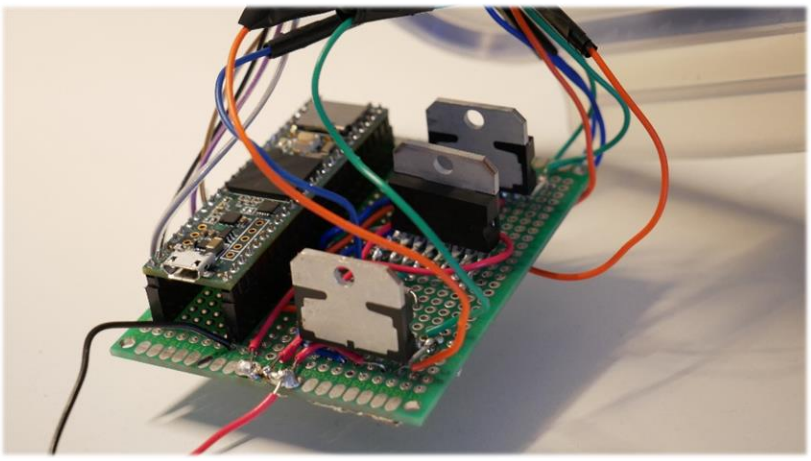

Figure 3.13 Control unit including the MCU - Teensy and the motor drivers

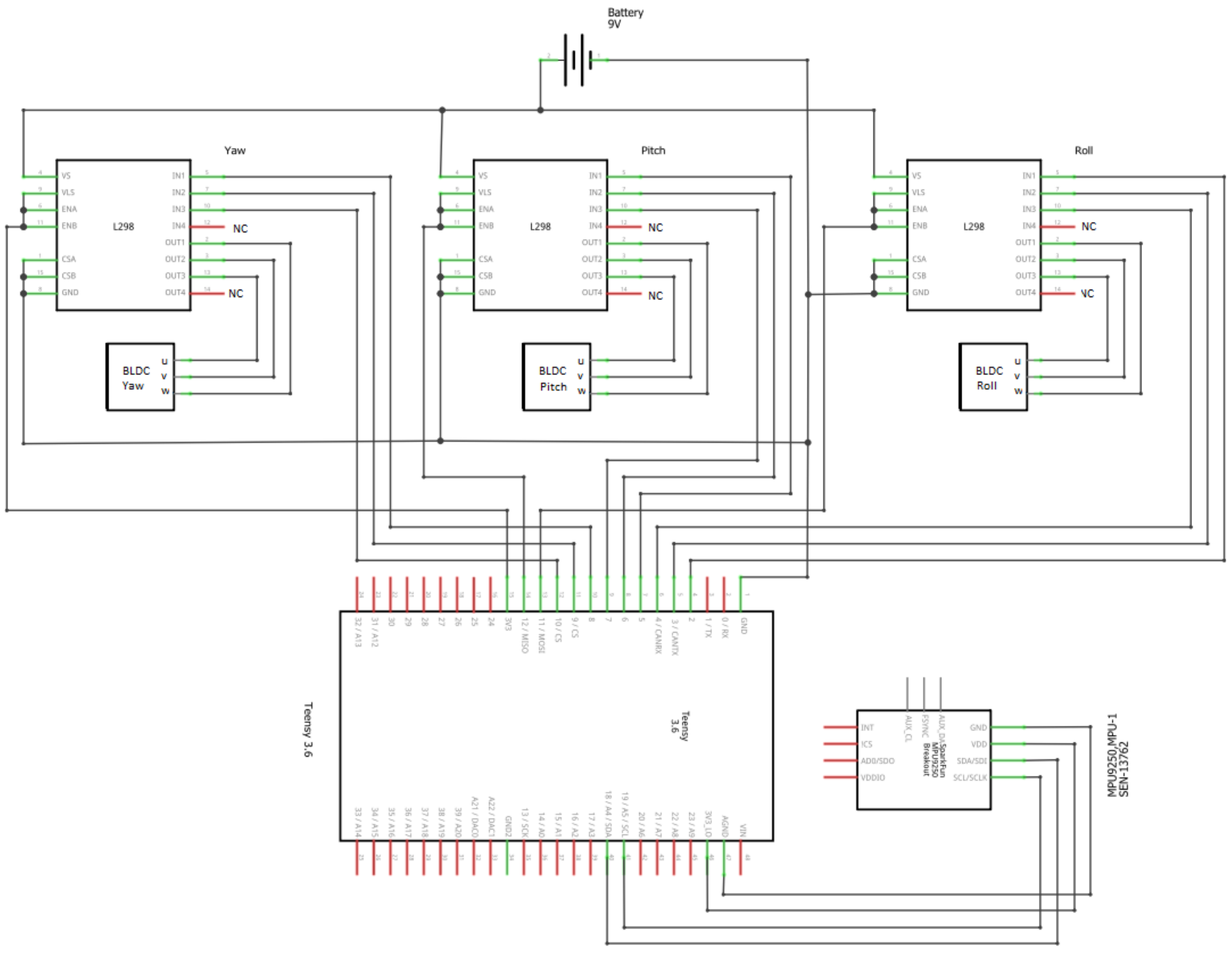

Figure 3.14 Schematic overview over the complete gimbal system

3.4.2 IMU Sensor Measurement Reading

3.4.2.1 Serial communication

The communication between a microcontroller and a sensor can be solved in many ways. Inter-integrated Circuit henceforward referred to as I^2C is one way to communicate between different units. I^2C consists of two signals, Serial Data Line (SDA) and Serial Clock Line (SCL) where the SCL is the clock signal and the SDA the data signal. A typical I^2C system consists of one master unit and several slave units, i.e. a microcontroller and a IMU sensor and only requires two wires and a common ground connection. The data transferred between the different units is done by a 7-bit address space where the information is sent bit by bit because of the single line connection. The different slave units have unique addresses which makes it possible for the different units to share the same SDA and SCL signals and lets the master unit know where to send the acquired data [12].

3.4.2.2 Signal processing and programming

To be able to control and monitor signal data the microcontroller teensy 3.6 was used. This microcontroller is used to make the calculations needed in the program, read sensor data and output the SPWM to the motors. To retrieve positional data of the camera a sensor is needed. In this project a sensor named MPU9250 was utilized. The plan was originally to use a MPU 6050 but after further research on previous projects a decision was made to use the latest MPU9250. The 9250 has the advantage of an extra sensor imbedded in the chip, this being the magnetometer which is used to get a more accurate reading.

The sensor is powered through the teensy and the positional data is read digitally through a serial communication protocol called I^2C. The sensor has many functions, the ones used in this project are the basic sensitivity settings and the refresh rate of the sensor. This project involves small movements, therefore the highest sensitivity option was chosen to yield best possible reading from the sensor.
An Arduino library for the sensor was used to convert the information from bits to raw sensor data. The raw sensor data is then converted according to the included datasheet to match the sensitivity settings and get the right unit of measurement for each internal sensor. This data can then be used to calculate the pitch roll and yaw angles. The angles from the acceleration data are calculated in the following manner:

$$roll = \arctan\left(\frac{a_x}{\sqrt{a_y^2 + a_z^2}}\right) \tag{3.1}$$

$$pitch = arctan\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right) \tag{3.2}$$

The yaw angle is calculated in a few different ways before settling for a sensor fusion method with magnetometer and gyro data. One can get the angle by integrating the angular velocity:

$$\theta = \theta_0 + \int \omega(t)dt \tag{3.3}$$

This method yields a rather inaccurate result over time since the small integration errors adds up quickly making the angle drift. The acceleration angle obtained contains a lot of noise and disturbances from miniscule movements. This needed to be filtered out to get a more reliable reading. To get a more stable angle the gyroscope and acceleration angles were fused together through both complementary and Kalman filters. The different weight values for both Kalman and complementary filter were obtained through trial and error.

3.4.3 Motor Control

The motors used in the project are derived from the existing Feiyu tech G4S gimbal and are typical three phase BLDC motors. To drive the motors a motor control system was designed using the Teensy 3.6 microcontroller and three motor driver circuits, L298N consisting of dual H-bridges. Each H-bridge contributed to a triple half bridge to run the motors. A three-phase sinusoidal signal with 120° phase shift was generated by a SPWM lookup table to drive one of the motors in both directions using one of the motor driver circuits. Several different look-up tables with different resolutions was used to optimize the motor control. A look-up table with 8-bit resolution containing 360 values with a maximum amplitude of 255 seemed to generate the best result. The frequency of the PWM signal generating the sinusoidal waves was originally set for >20 kHz to avoid noise. A frequency of 488 Hz ended up being the optimal frequency to run the motor with the program that was created. The idea behind the program that was designed was to run the motor in a similar fashion as a synchronous motor. The generated sine wave rotates the magnetic field in the motor but unlike a synchronous motor the speed of the rotational magnetic field can be controlled. This method does not utilize any feedback to sense the position of the rotor which means the system ended up being an open-loop system. A single program was created to run one motor and then the same program was applied to the second and third motor.

3.4.4 Regulation of the system

After calculating the necessary angles, positional data of the camera is retrieved by mounting the sensor on the frame. To get the system to regulate itself meaning changing motor direction and speed depending on the position of the camera a PID-regulator was implemented in the program. This closes the loop between the motor control and the sensor retrieving values. The PID then controls the motor speed by varying the time it takes to step through the dedicated lookup table to create a SPWM.

4. Results

4.1 Prototype

A prototype has been built as shown in figure 3.12. It has been tested and evaluated. The results have shown that it is able to run all three axes to an extent. However, the third axis being the yaw axis is not functioning the way it was intended during longer runs since the magnetometer was not used in the final construction.

4.2 IMU sensor

The acquired raw data signal from the IMU sensor was processed and filtered to obtain a reliable position of the camera angle. The accelerometer data was very noisy and the gyroscope data tended to drift over time. To solve this problem a sensor-fusion algorithm was introduced by first applying a complementary filter to the accelerometer- and gyroscope data. Later a Kalman filter was introduced to further stabilize the obtained signals. In figure 4.1 and 4.2 the noise of the accelerometer data and the drift from the gyroscope data can be observed.

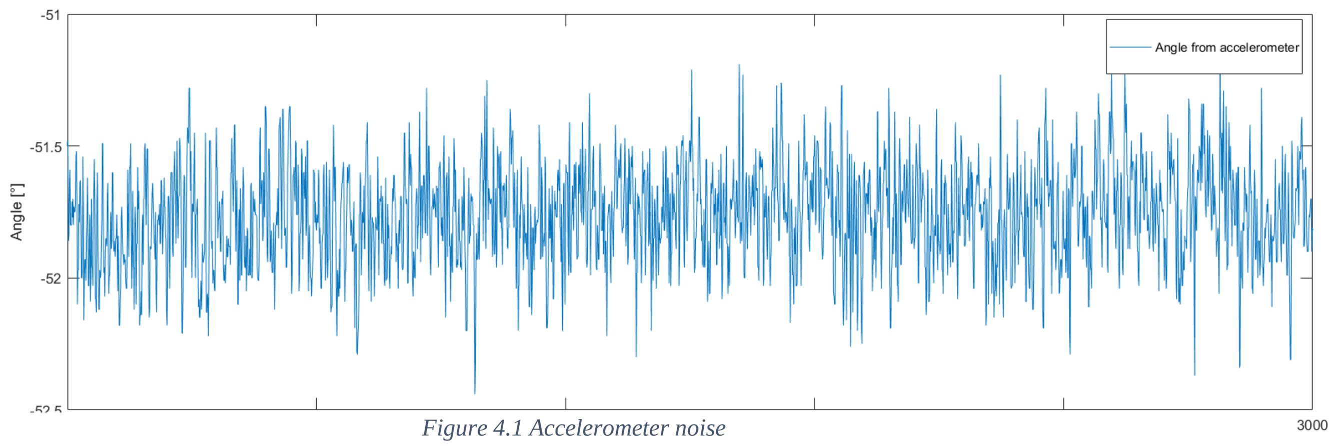

Figure 4.1 Accelerometer noise

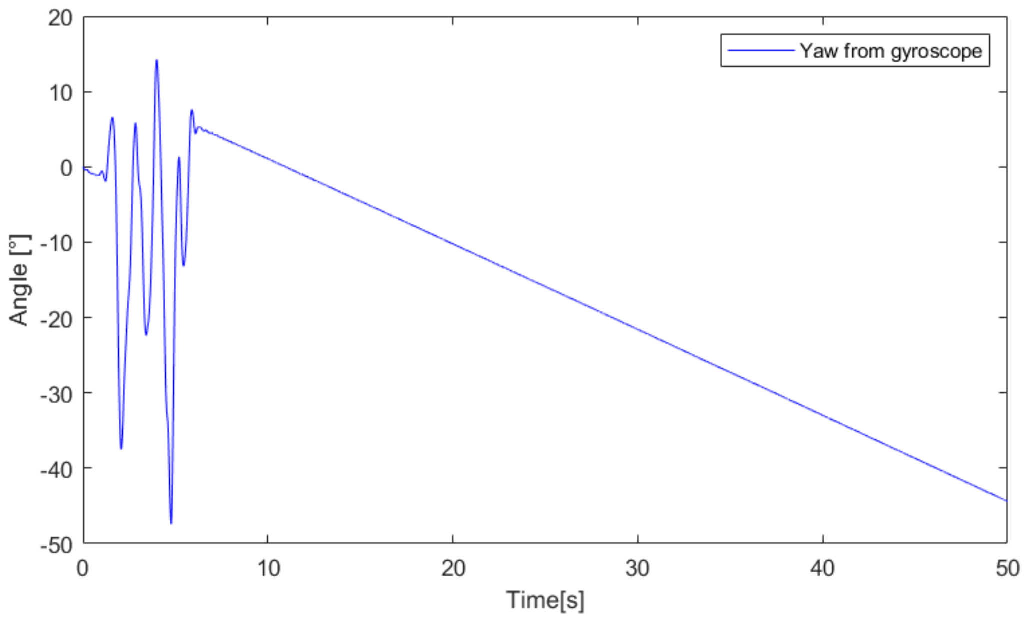

Figure 4.2 Drift overtime from the gyroscope data

The drift from the gyroscope data is caused by an integration error and occurs even though the bias is zero. The drift is a result of accumulating white noise of the integration reading.

By implementing the sensor fusion algorithm, the data from the gyroscope and accelerometer are merged into one signal by giving the signals different weights, according to Eq. (2.2). For example, when the complementary filter was used the weight of the accelerometer data was set to 0.04 (4%) and the gyroscope was set to 0.96 (96 %) to obtain a smooth and filtered signal as can be seen in figure 4.7. The Kalman filter also fuses the two signals into one with varying weights for the different filter parameters Q-Bias, Q-Angle and R-measure. Q-bias was set to 0.003, Q-Angle to 0.001 and R-Measure to 0.03 and a further stabilized signal was acquired compared to the complementary filter. This can be seen in the figure below. Q-Bias process the noise variance for the gyro bias meanwhile the Q-angle process the noise variance for the accelerometer and R-Measure the general noise variance.

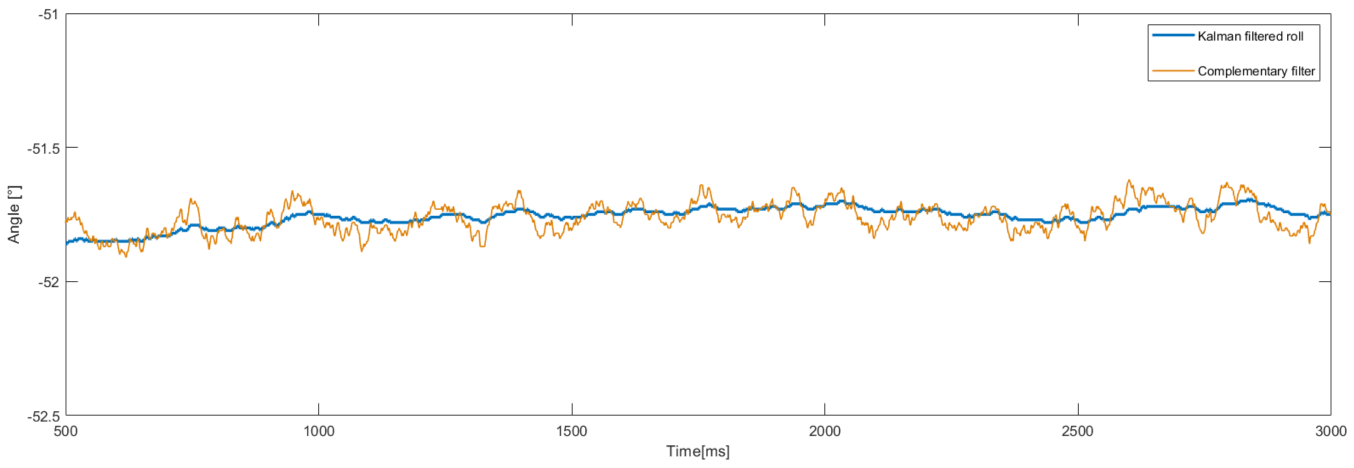

Figure 4.3 The filtered signal from the roll angle using complementary- and Kalman filter

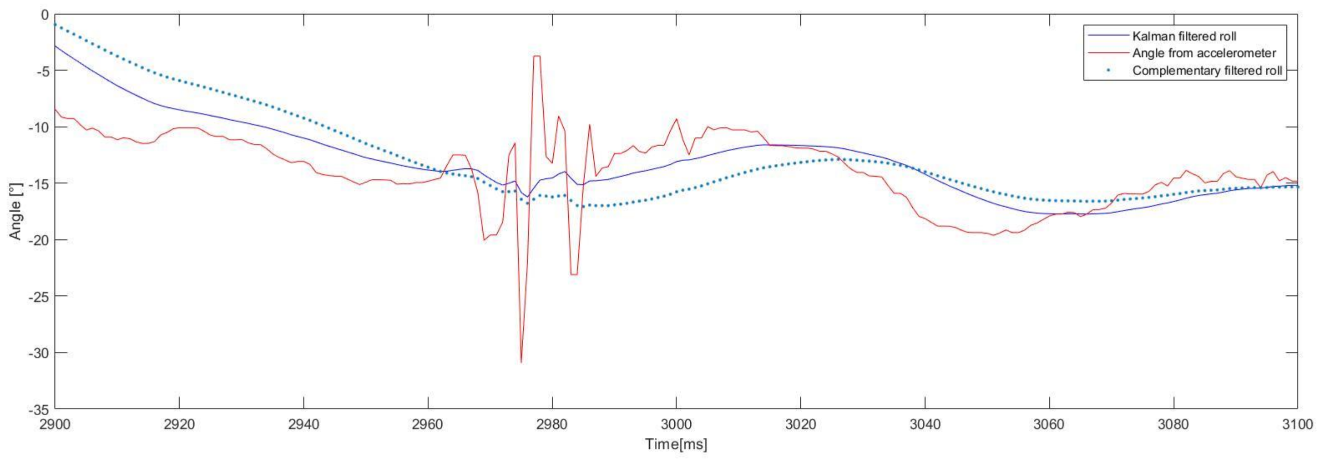

Figure 4.4 Kalman filter and complementary filter removing unwanted spike

In figure 4.4 a clear almost 30-degree unpredictable spike from the accelerometer is shown. It is also visible that both the Kalman and complementary filter does a good job of excluding this inconsistency.

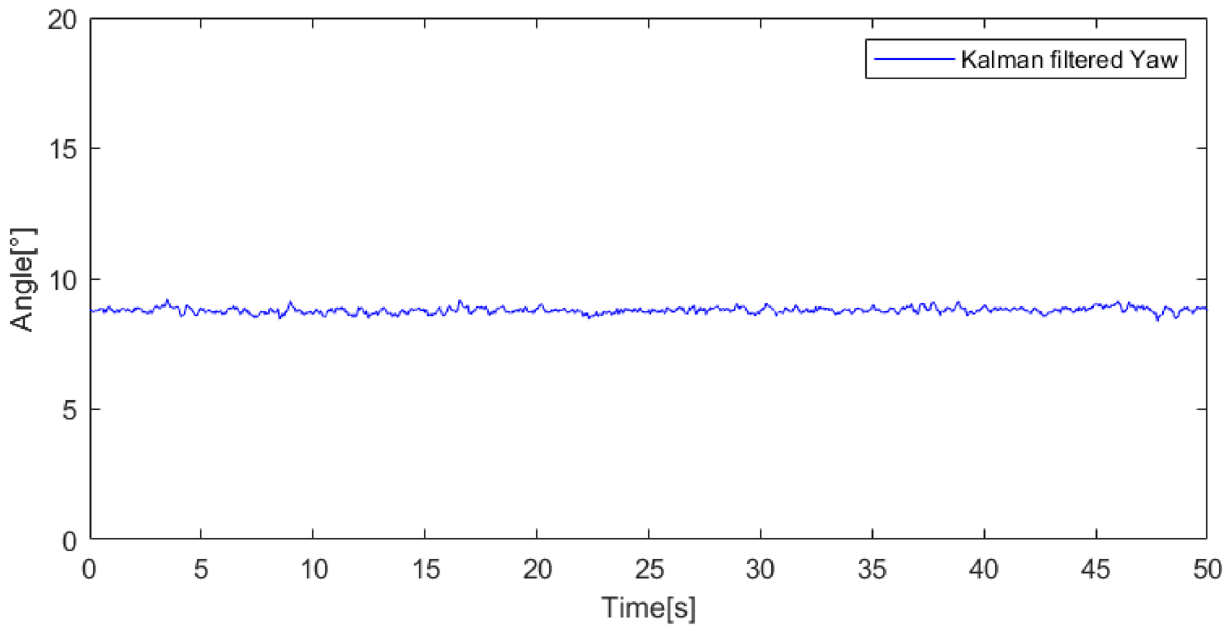

Figure 4.5 Filtered yaw angle without drift with magnetometer data

Figure 4.5 shows a longer run where the yaw angle is obtained from the Kalman filter which is fusing data from the magnetometer together with the gyroscope values. This yields a very stable yaw output with close to no drift compared to the yaw angle obtained from only the gyroscope as seen in figure 4.5.

4.3 Complete system

The gimbal can stabilize two out of the three axes efficiently using the input from the IMU sensor to sense the angle position of the camera and stabilize it by running each separate motor to the desired angle. The stabilization process works well on the roll and pitch axis but lacks in precision for the yaw axis.

The IMU sensor data is processed and filtered using a Kalman filter to obtain the most reliable angle position of the camera possible. The motor control then uses a basic PID-controller to regulate the speed and direction of the motors to keep them at the desired angle. The speed of the motors is controlled by how fast the values from the SPWM look-up table is processed which generates the sinusoidal wave to drive the motors. The greater the error the faster the look-up table values are run through.

The PID gains, K_P, K_D and K_I were carefully chosen by firstly applying Ziegler Nichols method [13] and then further improved by trial and error. To obtain a stable system a dedicated PID-controller were implemented for each of the three axes with different gain values. The PID-gains used for the different axes can be seen in table 1.

	K_P	K_D	K_I
Roll	45	5.6	0,002
Pitch	35	5,2	0,02
Yaw	10	10	0

Table 4.1: PID parameters for the different axes

The performance of the gimbal is relatively good but has room for improvement. The step response curves below show an estimation of the performance for each axis on the gimbal.

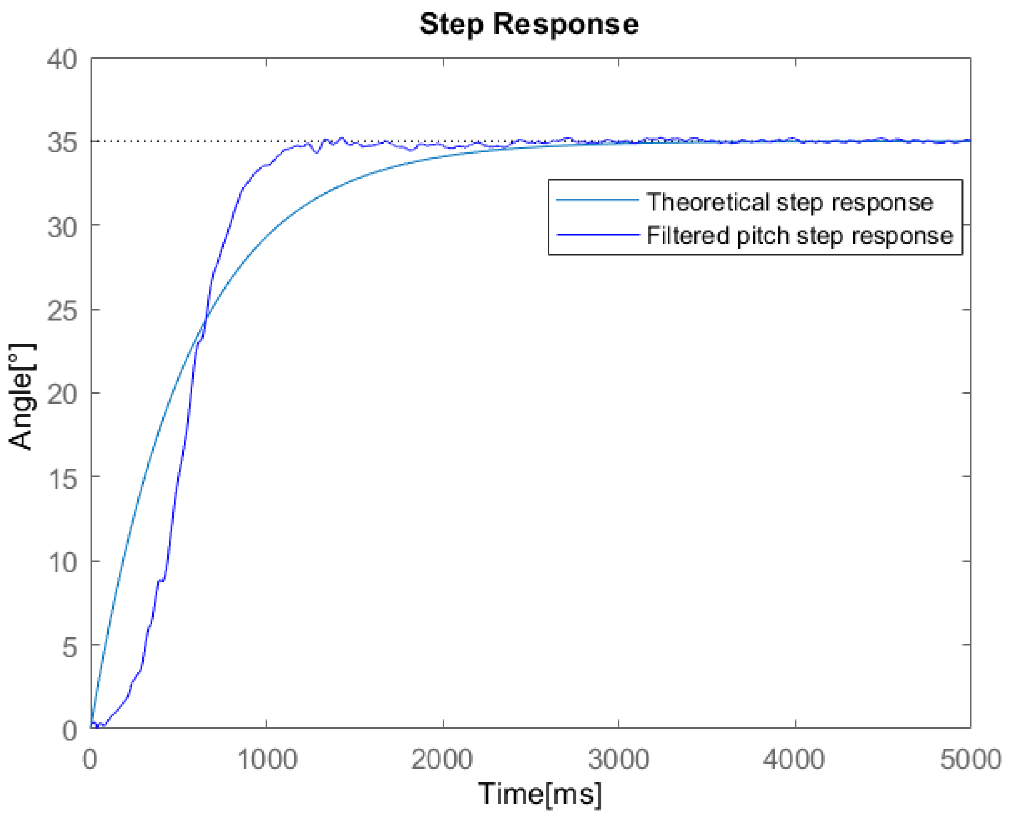

Figure 4.6 Step response curve of the pitch direction, from 0 to 35°

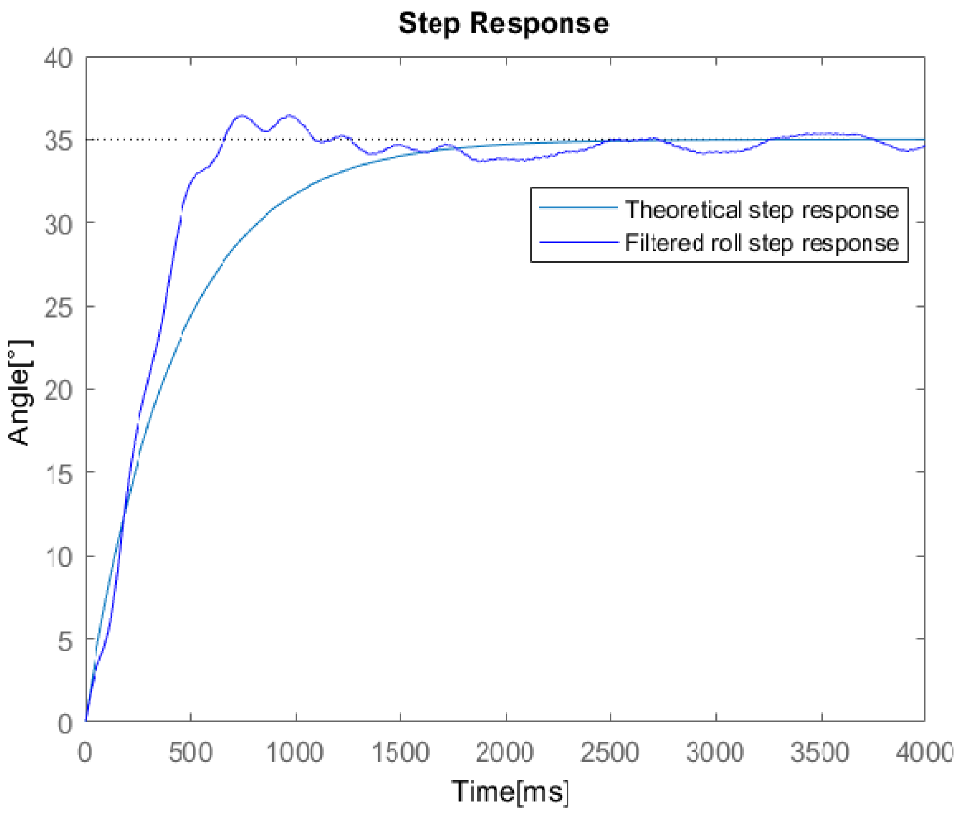

Figure 4.7 Step response curve of the roll direction, from 0 to 35°

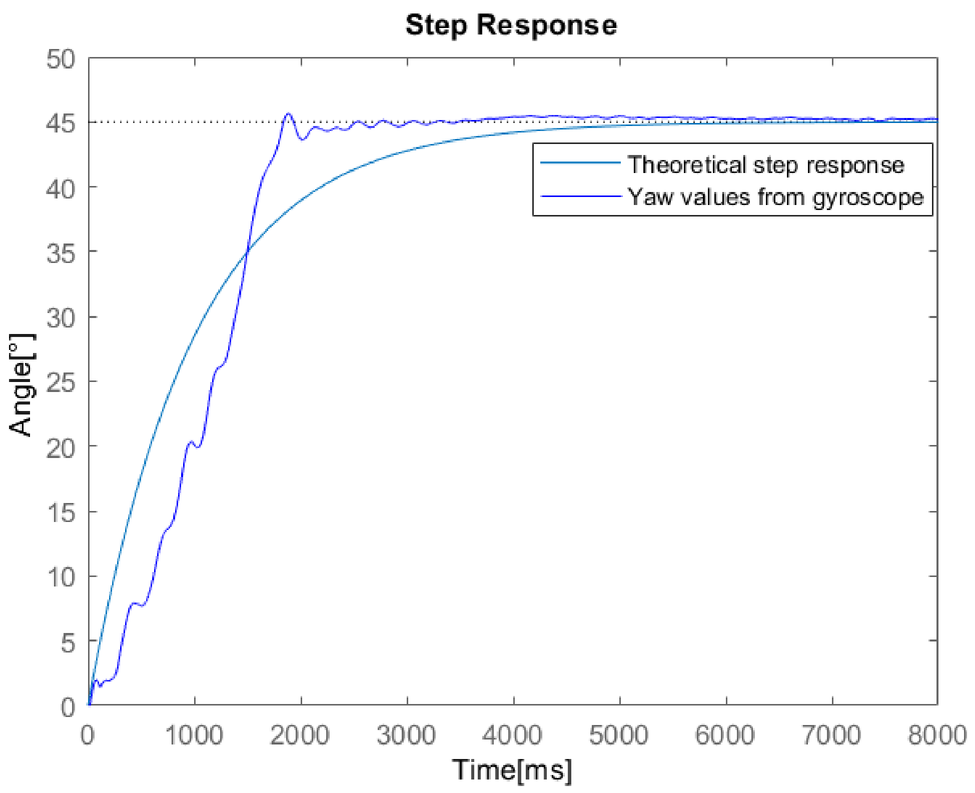

Figure 4.8 Step response curve of the yaw direction, moving from 0 to 45°

The step response curves clearly follow a first order transfer function which has the form of a first order differential equation as follows:

$$\tau\dot{y} + y = k_{dc}u \text{ or } \dot{y} + ay = bu \tag{4.1}$$

and the transfer function given by

$$G(s) = \frac{k_{dc}}{\tau s + 1} = \frac{b}{s + a} \tag{4.2}$$

Where k$_{dc}$ the DC gain [14] for a first order system can be calculated as follow $k_{dc} = {}^{b}/_{a}$ and τ represents the time constant of a first order system which is equal to 63 % of the time it takes to reach its steady state and can be calculated as follows $\tau = {}^{1}/_{a}$.

The step response of the above first order system is given by

$$y(t) = K - Ke^{-at} = K(1 - e^{-at}) \tag{4.3}$$

5. Discussion

Throughout the project there were many setbacks and ideas that had to be scrapped because of the limited timeframe of ten weeks. The problems that were met designing the control system for the gimbal took longer than suspected and took time to solve.

5.1 Motors and motor control

The motors originally used in the Feiyu-tech G4S were reused in this project. This was later discovered to be problematic since neither datasheets nor motor information were available online. This resulted in a lot of testing and trial and error to obtain suitable running conditions. A consequence of this was that an inaudible frequency where the motor would run smoothly was not found. Instead it was run at 488 Hz to compromise the noise for a smoother running gimbal.

A common issue when running a BLDC motor very slowly, which was required in this application is torque cogging. To get a smooth-running motor without any torque cogging several different SPWM and SVPWM look-up tables were tested. Different resolutions and amplitudes were also tested until the motor could run very slowly in a smooth manner.

The motor control system could be greatly improved by implementing hall-effect sensor. This would mean going from an open loop system to a closed one. The benefits of using hall-effect sensors would be knowing the position of the rotor which would make the energizing sequence of the coils much more accurate and make the motor run even better.

5.2 IMU values

Obtaining roll and pitch values from the IMU was done without issues. The yaw values however proved to be a lot more challenging. It was possible to get a simple yaw reading from just integrating the gyroscope values. The problem with this was the terrible drift that occurred during a longer run making it a very unreliable way of obtaining the yaw angle. Next method was to implement the magnetometer values to plant the gyro values and stop it from drifting. This was done through the Kalman filtering process which yielded reliable values after calibrating the magnetometer. This worked decently when the sensor was not mounted on the construction however after mounting the sensor the yaw angles obtained from the fused magnetometer and gyroscope were nonfunctioning. This was caused by the magnets inside the motor which is located very close to the sensor. These magnets interfere with the magnetometer readings making it function very poorly. This could most likely be solved by repositioning the sensor to a position where the magnets would not interfere as much. In this case it would be difficult since all the wiring and mounting places were limited to an existing construction. With more time a redesign could be made to make everything work as intended.

5.2 Filtering

The filters used in this project were effective at reducing unpredictable spikes and inconstancies from the accelerometer. After test running and plotting the complementary filter against the Kalman filter it was discovered that the output was not significantly different. If a micro controller with a lot less computational power were to be used it would be possible to use a complementary filter instead of the more computationally heavy Kalman filter. The result would be very close if not the same in the case of this project which is seen in figure 4.4.

5.4 PID- values

The PID values were obtained through trial and error. In the end they were set to values which yielded acceptable result without much instability. It could however be improved to create better step responses which in turn creates better stability for the camera.

6. Conclusions and future work

The finished version of the gimbal works as intended on two out of the three axes. The third axis which is the yaw axis in this case was not implemented in the way we wanted. The reason behind this is that the magnetometer which was initially planned to be used to calculate the yaw was working on a testing breadboard. However, when the sensor was mounted on the gimbal we discovered a lot of random values and very unreliable readings yielding this method of calculating yaw non-functioning on our prototype. This was likely caused by the magnets located directly underneath the sensor. The permanent magnets in the motors were neglected during the building process which unfortunately lead to the result of a yaw axis not working as intended. Hence the use of gyroscope values only, unlike the other two axes which used sensor fusion to get proper readings. This of course meant that the gimbal was not able to run yaw properly during longer runs because of the drift which the gyroscope introduces.
In the end our task and goals were accomplished within the ten-week timeframe. For future work a lot could be improved. If we were to do the project again we would first try to measure all the values that the original product outputs. This would simplify the choice of a frequency to run the motors efficiently and for performance comparison. Another improvement would be to implement hall effect sensor to run the motor at its optimal capability. Finally, we would either shield off the magnetic disturbance from the motor below the sensor or move the sensor to a location where the magnetometer would run undisturbed.

References

[1] "PID Controller," School of Engineering, University of Jordan, 11 December 2017. [Online]. Available: http://engineering.ju.edu.jo/Laboratories/07-PID%20Controller.pdf. [Accessed 16 May 2018].

[2] "Filter (signal processing)," Wikipedia, 14 May 2018 [Online]. Available: https://en.wikipedia.org/wiki/Filter_(signal_processing). [Accessed 20 May 2018].

[3] Olliw, "IMU Data Fusing: Complementary, Kalman, and Mahony Filter," 16 Jan 2015. [Online]. Available: http://www.olliw.eu/2013/imu-data-fusing/#chapter21. [Accessed 21 May 2018].

[4] K. S. Lauszus, "A practical approach to Kalman filter and how to implement it," TKJ Electronics, 10 Sep 2012. [Online]. Available: , http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it/. [Accessed 20 May 2018].

[5] "An Introduction to Brushless DC Motor Control," Digikey, 27 Mars 2013. [Online]. Available: https://www.digikey.com/en/articles/techzone/2013/mar/an-introduction-to-brushless-dc-motor-control. [Accessed 20 May 2018].

[6] Terbytes, "Implementation of an sPWM signal," Github, 14 Nov 2015. [Online]. Available: https://github.com/Terbytes/Arduino-Atmel-sPWM. [Accessed 20 May 2018].

[7] P. Stoffregen and R. Coon, "Teensy USB Development Board," PJRC, [Online]. Available: https://www.pjrc.com/store/teensy36.html. [Accessed 28 May 2018].

[8] "Inertial measurement unit," Wikipedia, 3 June 2018. [Online]. Available: https://en.wikipedia.org/wiki/Inertial_measurement_unit. [Accessed 28 May 2018].

[9] Sparkfun, "Accelerometer, Gyro and IMU Buying Guide," SparkFun, [Online]. Available: https://www.sparkfun.com/pages/accel_gyro_guide. [Accessed 26 May 2018].

[10] STMicroelectronics, " L298 Dual Full Bridge Driver " [Online]. Available: http://www.st.com/en/motor-drivers/l298.html. [Accessed 28 May 2018].

[11] P. Stoffregen and R. Coon, "Teensyduino," PJRC, [Online]. Available: https://www.pjrc.com/teensy/teensyduino.html. [Accessed 05 April 2018].

[12] SFUPTOWNMAKER, "I2C," SparkFun, [Online]. Available: https://learn.sparkfun.com/tutorials/i2c#i2c-at-the-hardware-level. [Accessed 23 May 2018].

[13] "Ziegler–Nichols method," Wikipedia, 22 April 2018. [Online]. Available: https://en.wikipedia.org/wiki/Ziegler%E2%80%93Nichols_method. [Accessed 23 May 2018].

[14] B. Messner, D. Tilbury, R. Hill, and J.D. Taylor, "Extras: Generating a Step Response in MATLAB," Michigan Engineering, Uuniversity of Michigan, [Online]. Available: http://ctms.engin.umich.edu/CTMS/index.php?aux=Extras_step [Accessed 24 May 2018].

Appendix

Main code to run the gimbal:

```
#include "MPU9250.h"
#include "Kalman.h"
#include "EEPROM.h"

//Kalman filter objects created
Kalman kalmanX;
Kalman kalmanY;
Kalman kalmanZ;

// MPU9250 object created.
MPU9250 IMU(Wire, 0x68);

int status;
double timer;
double pitch, roll, yaw;
float ax, ay, az, accX, accY, accZ;
float gxb, gyb, gzb, gx, gy, gz, mxb, myb, mzb, mxs, mys, mzs;
double magx, magy, magz;
double r2d = (180 / M_PI);
double mx, my,  mz;

double freq = 500;
int sineArraySize;

int table[] = {128, 130, 132, 134, 136, 138, 139, 141,
               143, 145, 147, 149, 151, 153, 155, 157,
               159, 161, 163, 165, 167, 169, 171, 173,
               174, 176, 178, 180, 182, 184, 185, 187,
               189, 191, 192, 194, 196, 198, 199, 201,
               202, 204, 206, 207, 209, 210, 212, 213,
               215, 216, 218, 219, 220, 222, 223, 224,
               226, 227, 228, 229, 231, 232, 233, 234,
               235, 236, 237, 238, 239, 240, 241, 242,
               243, 244, 245, 245, 246, 247, 247, 248,
               249, 249, 250, 250, 251, 251, 252, 252,
               253, 253, 253, 254, 254, 254, 254, 255,
               255, 255, 255, 255, 255, 255, 255, 255,
               255, 255, 254, 254, 254, 254, 253, 253,
               253, 252, 252, 251, 251, 250, 250, 249,
               249, 248, 247, 247, 246, 245, 245, 244,
               243, 242, 241, 240, 239, 238, 237, 236,
               235, 234, 233, 232, 231, 229, 228, 227,
               226, 224, 223, 222, 220, 219, 218, 216,
               215, 213, 212, 210, 209, 207, 206, 204,
               202, 201, 199, 198, 196, 194, 192, 191,
               189, 187, 185, 184, 182, 180, 178, 176,
               174, 173, 171, 169, 167, 165, 163, 161,
               159, 157, 155, 153, 151, 149, 147, 145,
               143, 141, 139, 138, 136, 134, 132, 130,
               128, 125, 123, 121, 119, 117, 116, 114,
               112, 110, 108, 106, 104, 102, 100, 98,
               96, 94, 92, 90, 88, 86, 84, 82,
               81, 79, 77, 75, 73, 71, 70, 68,
               66, 64, 63, 61, 59, 57, 56, 54,
               53, 51, 49, 48, 46, 45, 43, 42,
               40, 39, 37, 36, 35, 33, 32, 31,
               29, 28, 27, 26, 24, 23, 22, 21,
               20, 19, 18, 17, 16, 15, 14, 13,
               12, 11, 10, 10, 9, 8, 8, 7,
                6, 6, 5, 5, 4, 4, 3, 3,
                2, 2, 2, 1, 1, 1, 1, 0,
                0, 0, 0, 0, 0, 0, 0, 0,
                0, 0, 1, 1, 1, 1, 2, 2,
                2, 3, 3, 4, 4, 5, 5, 6,
                6, 7, 8, 8, 9, 10, 10, 11,
                12, 13, 14, 15, 16, 17, 18, 19,
                20, 21, 22, 23, 24, 26, 27, 28,
                29, 31, 32, 33, 35, 36, 37, 39,
                40, 42, 43, 45, 46, 48, 49, 51,
                53, 54, 56, 57, 59, 61, 63, 64,
                66, 68, 70, 71, 73, 75, 77, 79,
                81, 82, 84, 86, 88, 90, 92, 94,
                96, 98, 100, 102, 104, 106, 108, 110,
                112, 114, 116, 117, 119, 121, 123, 125
               };

double gyroXangle, gyroYangle; // Angle calculate using the gyro only
double compAngleX, compAngleY; // Calculated angle using a complementary filter
double kalAngleX, kalAngleY; // Calculated angle using a Kalman filter
double compPitch, gyroYaw, compRoll, magYaw, gxang;

//PID Variabler.
float PIDX, PIDY, PIDZ, errorX, errorY, errorZ, prev_errorX, prev_errorY, prev_errorZ;
float pidX_p, pidY_p, pidZ_p = 0;
float pidX_i, pidY_i, pidZ_i = 0;
float pidX_d, pidY_d, pidZ_d = 0;


double kxp = 45, kyp = 35, kzp = 10; //3.55
double kxi = 0.002, kyi = 0.002, kzi = 0.00; //0.003
double kxd = 2, kyd = 1.5, kzd = 10; //2.05,d=5.32 7.5

float desired_angleX = 0, desired_angleY = 0, desired_angleZ = 0;


//Motorstyrning
int motorXDelayActual, motorYDelayActual, motorZDelayActual = 60000;
int mXPin1 = 2, mXPin2 = 3, mXPin3 = 4;
int mYPin1 = 5, mYPin2 = 6, mYPin3 = 7;
int mZPin1 = 8, mZPin2 = 9, mZPin3 = 10;
int stepX, stepY, stepZ = 1;
int EN1 = 11;
int EN2 = 12;

int currentStepXA;
int currentStepXB;
int currentStepXC;
int currentStepYA;
int currentStepYB;
int currentStepYC;
int currentStepZA;
int currentStepZB;
int currentStepZC;

long lastMotorXDelayTime, lastMotorYDelayTime, lastMotorZDelayTime = 0;
double mCounter = micros();

uint8_t EepromBuffer[48];
float axb, axs, ayb, ays, azb, azs;
float hxb, hxs, hyb, hys, hzb, hzs;

void setup() {

  analogWriteResolution(8);
```

```
// serial to display data
Serial.begin(115200);
while (!Serial) {}

// start communication with IMU
status = IMU.begin();
delay(100);


accX = IMU.getAccelX_mss(), accY = IMU.getAccelY_mss(), accZ = IMU.getAccelZ_mss();
gx = IMU.getGyroX_rads(), gy = IMU.getGyroY_rads(), gz = IMU.getGyroZ_rads();
mx = IMU.getMagX_uT(), my = IMU.getMagY_uT(), mz = IMU.getMagZ_uT();

double roll = (180 / M_PI) * atan(accX / sqrt(sq(accY) + sq(accZ)));
double pitch = (180 / M_PI) * atan(accY / sqrt(sq(accX) + sq(accZ)));
double yaw = atan2(my, mx) * r2d;

// Set starting angle

kalmanX.setAngle(roll);
kalmanY.setAngle(pitch);
kalmanZ.setAngle(yaw);
gyroXangle = roll;
gyroYangle = pitch;
compAngleX = roll;
compAngleY = pitch;

if (status < 0) {
  Serial.println("IMU initialization unsuccessful");
  Serial.println("Check IMU wiring or try cycling power");
  Serial.print("Status: ");
  Serial.println(status);
  while (1) {}
}
for (size_t i = 0; i < sizeof(EepromBuffer); i++) {
  EepromBuffer[i] = EEPROM.read(i);
}
memcpy(&axb, EepromBuffer + 0, 4);
memcpy(&axs, EepromBuffer + 4, 4);
memcpy(&ayb, EepromBuffer + 8, 4);
memcpy(&ays, EepromBuffer + 12, 4);
memcpy(&azb, EepromBuffer + 16, 4);
memcpy(&azs, EepromBuffer + 20, 4);
memcpy(&hxb, EepromBuffer + 24, 4);
memcpy(&hxs, EepromBuffer + 28, 4);
memcpy(&hyb, EepromBuffer + 32, 4);
memcpy(&hys, EepromBuffer + 36, 4);
memcpy(&hzb, EepromBuffer + 40, 4);
memcpy(&hzs, EepromBuffer + 44, 4);

IMU.setAccelCalX(axb, axs);
IMU.setAccelCalY(ayb, ays);
IMU.setAccelCalZ(azb, azs);

IMU.setMagCalX(hxb, hxs);
IMU.setMagCalY(hyb, hys);
IMU.setMagCalZ(hzb, hzs);

//Set the accelorometer range
IMU.setAccelRange(MPU9250::ACCEL_RANGE_2G);
// et the gyroscope range.
IMU.setGyroRange(MPU9250::GYRO_RANGE_250DPS);
// setting DLPF bandwidth to 184 Hz
IMU.setDlpfBandwidth(MPU9250::DLPF_BANDWIDTH_184HZ);
// setting SRD to 0 for a 100 Hz update rate
```

```
  IMU.setSrd(0);

  pinMode(mXPin1, OUTPUT);
  pinMode(mXPin2, OUTPUT);
  pinMode(mXPin3, OUTPUT);
  pinMode(EN1, OUTPUT);
  pinMode(EN2, OUTPUT);
  digitalWrite(EN1, HIGH);
  digitalWrite(EN2, HIGH);

  timer = micros();

  analogWriteFrequency(mXPin1, freq);
  analogWriteFrequency(mXPin2, freq);
  analogWriteFrequency(mXPin3, freq);
  analogWriteFrequency(mYPin1, freq);
  analogWriteFrequency(mYPin2, freq);
  analogWriteFrequency(mYPin3, freq);
  analogWriteFrequency(mZPin1, freq);
  analogWriteFrequency(mZPin2, freq);
  analogWriteFrequency(mZPin3, freq);

  sineArraySize = sizeof(table) / sizeof(int);
  int phaseShift = sineArraySize / 3;

  currentStepXA = 0;
  currentStepXB = currentStepXA + phaseShift;
  currentStepXC = currentStepXB + phaseShift;

  currentStepYA = 0;
  currentStepYB = currentStepYA + phaseShift;
  currentStepYC = currentStepYB + phaseShift;

  currentStepZA = 0;
  currentStepZB = currentStepZA + phaseShift;
  currentStepZC = currentStepZB + phaseShift;

  sineArraySize--;
}

void loop() {
  IMU.readSensor(); // read the sensor
  double dT = (double)(micros() - timer) / 1000000;
  timer = micros();

  double accX = IMU.getAccelX_mss();
  double accY = IMU.getAccelY_mss();
  double accZ = IMU.getAccelZ_mss();
  double gyroX = IMU.getGyroX_rads();
  double gyroY = IMU.getGyroY_rads();
  double gyroZ = IMU.getGyroZ_rads();

  mx = IMU.getMagX_uT();
  my = IMU.getMagY_uT();
  mz = IMU.getMagZ_uT();

  //Angle from accelorometer
  double roll = (180 / M_PI) * atan(accX / sqrt(sq(accY) + sq(accZ)));
  double pitch = (180 / M_PI) * atan(accY / sqrt(sq(accX) + sq(accZ)));
  double yaw = atan2(my, mx) * r2d;

  // Angle from gyro
  double gyroXrate = gyroXrate + (gyroX * RAD_TO_DEG) * dT;
  double gyroYrate = gyroYrate + (gyroY * RAD_TO_DEG) * dT;
  double gyroZrate = gyroZrate + (gyroZ * RAD_TO_DEG) * dT;
  //Angle from kalman
  double kalRoll = kalmanX.getAngle(roll, gyroXrate, dT);
  double kalPitch = kalmanY.getAngle(pitch, gyroYrate, dT);
  double kalYaw = kalmanZ.getAngle(yaw, gyroZrate, dT);

  //Angle from comp.
  compRoll = (double)0.96 * (compRoll + gyroY * dT) + 0.04 * roll;
  compPitch = (double)0.96 * (compPitch + gyroX * dT) + 0.04 * pitch;

  gyroYaw = (double)(gyroYaw + gyroZ * dT);

  runPIDX(kalRoll, desired_angleX, dT);
  runPIDY(kalPitch, desired_angleY, dT);
  runPIDZ(gyroYaw, desired_angleZ, dT);

  //Run Motors
  if ((micros() - lastMotorXDelayTime) >  motorXDelayActual) {
    runMotorX();
    lastMotorXDelayTime = micros();
  }
  if ((micros() - lastMotorYDelayTime) >  motorYDelayActual) {
    runMotorY();
    lastMotorYDelayTime = micros();
  }
  if ((micros() - lastMotorZDelayTime) >  motorZDelayActual) {
    runMotorZ();
    lastMotorZDelayTime = micros();
  }

}

void calAcc(void) {
  Serial.println("Starting Accelerometer Calibration"), IMU.calibrateAccel();
  Serial.println("Switch"), delay(5000), IMU.calibrateAccel();
  Serial.println("Switch"), delay(5000), IMU.calibrateAccel();
  Serial.println("Switch"), delay(5000), IMU.calibrateAccel();
  Serial.println("Switch"), delay(5000), IMU.calibrateAccel();
  Serial.println("Switch"), delay(5000), IMU.calibrateAccel();
  Serial.println("Done");
}
//Run motors
void runMotorX(void) {

  currentStepXA = currentStepXA + stepX;
  if (currentStepXA > sineArraySize) currentStepXA = 0;
  if (currentStepXA < 0) currentStepXA = sineArraySize;
  currentStepXB = currentStepXB + stepX;
  if (currentStepXB > sineArraySize) currentStepXB = 0;
  if (currentStepXB < 0) currentStepXB = sineArraySize;
  currentStepXC = currentStepXC + stepX;
  if (currentStepXC > sineArraySize) currentStepXC = 0;
  if (currentStepXC < 0) currentStepXC = sineArraySize;

  analogWrite(mXPin1, table[currentStepXA]);
  analogWrite(mXPin2, table[currentStepXB]);
  analogWrite(mXPin3, table[currentStepXC]);

}

void runMotorY(void) {

  currentStepYA = currentStepYA + stepY;
  if (currentStepYA > sineArraySize) currentStepYA = 0;
  if (currentStepYA < 0) currentStepYA = sineArraySize;
  currentStepYB = currentStepYB + stepY;
  if (currentStepYB > sineArraySize) currentStepYB = 0;
  if (currentStepYB < 0) currentStepYB = sineArraySize;
  currentStepYC = currentStepYC + stepY;
  if (currentStepYC > sineArraySize) currentStepYC = 0;
  if (currentStepYC < 0) currentStepYC = sineArraySize;


  analogWrite(mYPin1, table[currentStepYA]);
  analogWrite(mYPin2, table[currentStepYB]);
  analogWrite(mYPin3, table[currentStepYC]);


}
void runMotorZ(void) {

  currentStepZA = currentStepZA + stepZ;
  if (currentStepZA > sineArraySize) currentStepZA = 0;
  if (currentStepZA < 0) currentStepZA = sineArraySize;
  currentStepZB = currentStepZB + stepZ;
  if (currentStepZB > sineArraySize) currentStepZB = 0;
  if (currentStepZB < 0) currentStepZB = sineArraySize;
  currentStepZC = currentStepZC + stepZ;
  if (currentStepZC > sineArraySize) currentStepZC = 0;
  if (currentStepZC < 0) currentStepZC = sineArraySize;


  analogWrite(mZPin1, table[currentStepZA]);
  analogWrite(mZPin2, table[currentStepZB]);
  analogWrite(mZPin3, table[currentStepZC]);

}
//Run PIDs
void runPIDX(double kalRoll, double desired_angleX, double dT) {

  errorX = kalRoll - desired_angleX;  pidX_p = kxp * errorX;

  if (errorX < 5 && errorX > -5) {
    pidX_i = pidX_i + (kxi * errorX);
  } else {
    pidX_i = 0;
  }

  pidX_d = kxd * ((errorX - prev_errorX) / dT);

  PIDX = pidX_p + pidX_i + pidX_d;
  prev_errorX = errorX;
  if (errorX > 0) {
    stepX = -1;
  } else {
    stepX = 1;
  }

  if (PIDX < 5 && PIDX > -5) {
    motorXDelayActual = 100000;
  } else {
    motorXDelayActual = abs(motorXDelayActual - abs(PIDX));
  }
  if (motorXDelayActual < 1000) {
    motorXDelayActual = 1000;
  }
}
void runPIDY(double kalPitch, double desired_angleY, double dT) {

  errorY = kalPitch - desired_angleY;
  pidY_p = kyp * errorY;
  if (-5 < errorY  && errorY < 5) {
    pidY_i = pidY_i + (kyi * errorY);
  } else {
    pidY_i = 0;
  }
  pidY_d = kyd * ((errorY - prev_errorY) / dT);
  PIDY = pidY_p + pidY_i + pidY_d;
  prev_errorY = errorY;

  if (errorY > 0) {
    stepY = -1;
  } else {
    stepY = 1;
  }

  if (PIDY < 5 && PIDY > -5) {
    motorYDelayActual = 100000;
  } else {
    motorYDelayActual = abs(motorYDelayActual - abs(PIDY));
  }
  if (motorYDelayActual < 1000) {
    motorYDelayActual = 1000;
  }
}
void runPIDZ(double kalYaw, double desired_angleZ, double dT) {

  errorZ = kalYaw - desired_angleZ;
  pidZ_p = kzp * errorZ;
  if (-5 < errorZ < 5) {
    pidZ_i = pidZ_i + (kzi * errorZ);
  } else {
    pidZ_i = 0;
  }
  pidZ_d = kzd * ((errorZ - prev_errorZ) / dT);
  PIDZ = pidZ_p + pidZ_i + pidZ_d;
  prev_errorZ = errorZ;

  if (errorZ > 0) {
    stepZ = -1;
  } else {
    stepZ = 1;
  }

  if (PIDZ < 5 && PIDZ > -5) {
    motorZDelayActual = 100000;
  } else {
    motorZDelayActual = abs(motorZDelayActual - abs(PIDZ));
  }
  if (motorZDelayActual < 1000) {
    motorZDelayActual = 1000;
  }
}
```

Library code: MPU9250

```
/*
MPU9250.h
Brian R Taylor
brian.taylor@bolderflight.com

Copyright (c) 2017 Bolder Flight Systems

Permission is hereby granted, free of charge, to any person obtaining a copy of this
software
and associated documentation files (the "Software"), to deal in the Software without
restriction,
including without limitation the rights to use, copy, modify, merge, publish,
distribute,
sublicense, and/or sell copies of the Software, and to permit persons to whom the
Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies
or
substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
INCLUDING
BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE
AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM,
DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
*/

#ifndef MPU9250_h
#define MPU9250_h

#include "Arduino.h"
#include "Wire.h"    // I2C library
#include "SPI.h"     // SPI library

class MPU9250{
  public:
    enum GyroRange
    {
      GYRO_RANGE_250DPS,
      GYRO_RANGE_500DPS,
      GYRO_RANGE_1000DPS,
      GYRO_RANGE_2000DPS
    };
    enum AccelRange
    {
      ACCEL_RANGE_2G,
      ACCEL_RANGE_4G,
      ACCEL_RANGE_8G,
      ACCEL_RANGE_16G
    };
    enum DlpfBandwidth
    {
      DLPF_BANDWIDTH_184HZ,
      DLPF_BANDWIDTH_92HZ,
      DLPF_BANDWIDTH_41HZ,
      DLPF_BANDWIDTH_20HZ,
      DLPF_BANDWIDTH_10HZ,
      DLPF_BANDWIDTH_5HZ
    };
    enum LpAccelOdr
    {
      LP_ACCEL_ODR_0_24HZ = 0,
      LP_ACCEL_ODR_0_49HZ = 1,
      LP_ACCEL_ODR_0_98HZ = 2,
      LP_ACCEL_ODR_1_95HZ = 3,
      LP_ACCEL_ODR_3_91HZ = 4,
      LP_ACCEL_ODR_7_81HZ = 5,
      LP_ACCEL_ODR_15_63HZ = 6,
      LP_ACCEL_ODR_31_25HZ = 7,
      LP_ACCEL_ODR_62_50HZ = 8,
      LP_ACCEL_ODR_125HZ = 9,
      LP_ACCEL_ODR_250HZ = 10,
      LP_ACCEL_ODR_500HZ = 11
    };
    MPU9250(TwoWire &bus,uint8_t address);
    MPU9250(SPIClass &bus,uint8_t csPin);
    int begin();
    int setAccelRange(AccelRange range);
    int setGyroRange(GyroRange range);
    int setDlpfBandwidth(DlpfBandwidth bandwidth);
    int setSrd(uint8_t srd);
    int enableDataReadyInterrupt();
    int disableDataReadyInterrupt();
    int enableWakeOnMotion(float womThresh_mg,LpAccelOdr odr);
    int readSensor();
    float getAccelX_mss();
    float getAccelY_mss();
    float getAccelZ_mss();
    float getGyroX_rads();
    float getGyroY_rads();
    float getGyroZ_rads();
    float getMagX_uT();
    float getMagY_uT();
    float getMagZ_uT();
    float getTemperature_C();

    int calibrateGyro();
    float getGyroBiasX_rads();
    float getGyroBiasY_rads();
    float getGyroBiasZ_rads();
    void setGyroBiasX_rads(float bias);
    void setGyroBiasY_rads(float bias);
    void setGyroBiasZ_rads(float bias);
    int calibrateAccel();
    float getAccelBiasX_mss();
    float getAccelScaleFactorX();
    float getAccelBiasY_mss();
    float getAccelScaleFactorY();
    float getAccelBiasZ_mss();
    float getAccelScaleFactorZ();
    void setAccelCalX(float bias,float scaleFactor);
    void setAccelCalY(float bias,float scaleFactor);
    void setAccelCalZ(float bias,float scaleFactor);
    int calibrateMag();
    float getMagBiasX_uT();
    float getMagScaleFactorX();
    float getMagBiasY_uT();
    float getMagScaleFactorY();
    float getMagBiasZ_uT();
    float getMagScaleFactorZ();
    void setMagCalX(float bias,float scaleFactor);
    void setMagCalY(float bias,float scaleFactor);
    void setMagCalZ(float bias,float scaleFactor);
  protected:
    // i2c
```

```
uint8_t _address;
TwoWire *_i2c;
const uint32_t _i2cRate = 400000; // 400 kHz
size_t _numBytes; // number of bytes received from I2C
// spi
SPIClass *_spi;
uint8_t _csPin;
bool _useSPI;
bool _useSPIHS;
const uint8_t SPI_READ = 0x80;
const uint32_t SPI_LS_CLOCK = 1000000;  // 1 MHz
const uint32_t SPI_HS_CLOCK = 15000000; // 15 MHz
// track success of interacting with sensor
int _status;
// buffer for reading from sensor
uint8_t _buffer[21];
// data counts
int16_t _axcounts,_aycounts,_azcounts;
int16_t _gxcounts,_gycounts,_gzcounts;
int16_t _hxcounts,_hycounts,_hzcounts;
int16_t _tcounts;
// data buffer
float _ax, _ay, _az;
float _gx, _gy, _gz;
float _hx, _hy, _hz;
float _t;
// wake on motion
uint8_t _womThreshold;
// scale factors
float _accelScale;
float _gyroScale;
float _magScaleX, _magScaleY, _magScaleZ;
const float _tempScale = 333.87f;
const float _tempOffset = 21.0f;
// configuration
AccelRange _accelRange;
GyroRange _gyroRange;
DlpfBandwidth _bandwidth;
uint8_t _srd;
// gyro bias estimation
size_t _numSamples = 100;
double _gxbD, _gybD, _gzbD;
float _gxb, _gyb, _gzb;
// accel bias and scale factor estimation
double _axbD, _aybD, _azbD;
float _axmax, _aymax, _azmax;
float _axmin, _aymin, _azmin;
float _axb, _ayb, _azb;
float _axs = 1.0f;
float _ays = 1.0f;
float _azs = 1.0f;
// magnetometer bias and scale factor estimation
uint16_t _maxCounts = 1000;
float _deltaThresh = 0.3f;
uint8_t _coeff = 8;
uint16_t _counter;
float _framedelta, _delta;
float _hxfilt, _hyfilt, _hzfilt;
float _hxmax, _hymax, _hzmax;
float _hxmin, _hymin, _hzmin;
float _hxb, _hyb, _hzb;
float _hxs = 1.0f;
float _hys = 1.0f;
float _hzs = 1.0f;
float _avgs;
```

```
// transformation matrix
/* transform the accel and gyro axes to match the magnetometer axes */
const int16_t tX[3] = {0,  1,  0};
const int16_t tY[3] = {1,  0,  0};
const int16_t tZ[3] = {0,  0, -1};
// constants
const float G = 9.807f;
const float _d2r = 1.0f;
// MPU9250 registers
const uint8_t ACCEL_OUT = 0x3B;
const uint8_t GYRO_OUT = 0x43;
const uint8_t TEMP_OUT = 0x41;
const uint8_t EXT_SENS_DATA_00 = 0x49;
const uint8_t ACCEL_CONFIG = 0x1C;
const uint8_t ACCEL_FS_SEL_2G = 0x00;
const uint8_t ACCEL_FS_SEL_4G = 0x08;
const uint8_t ACCEL_FS_SEL_8G = 0x10;
const uint8_t ACCEL_FS_SEL_16G = 0x18;
const uint8_t GYRO_CONFIG = 0x1B;
const uint8_t GYRO_FS_SEL_250DPS = 0x00;
const uint8_t GYRO_FS_SEL_500DPS = 0x08;
const uint8_t GYRO_FS_SEL_1000DPS = 0x10;
const uint8_t GYRO_FS_SEL_2000DPS = 0x18;
const uint8_t ACCEL_CONFIG2 = 0x1D;
const uint8_t ACCEL_DLPF_184 = 0x01;
const uint8_t ACCEL_DLPF_92 = 0x02;
const uint8_t ACCEL_DLPF_41 = 0x03;
const uint8_t ACCEL_DLPF_20 = 0x04;
const uint8_t ACCEL_DLPF_10 = 0x05;
const uint8_t ACCEL_DLPF_5 = 0x06;
const uint8_t CONFIG = 0x1A;
const uint8_t GYRO_DLPF_184 = 0x01;
const uint8_t GYRO_DLPF_92 = 0x02;
const uint8_t GYRO_DLPF_41 = 0x03;
const uint8_t GYRO_DLPF_20 = 0x04;
const uint8_t GYRO_DLPF_10 = 0x05;
const uint8_t GYRO_DLPF_5 = 0x06;
const uint8_t SMPDIV = 0x19;
const uint8_t INT_PIN_CFG = 0x37;
const uint8_t INT_ENABLE = 0x38;
const uint8_t INT_DISABLE = 0x00;
const uint8_t INT_PULSE_50US = 0x00;
const uint8_t INT_WOM_EN = 0x40;
const uint8_t INT_RAW_RDY_EN = 0x01;
const uint8_t PWR_MGMNT_1 = 0x6B;
const uint8_t PWR_CYCLE = 0x20;
const uint8_t PWR_RESET = 0x80;
const uint8_t CLOCK_SEL_PLL = 0x01;
const uint8_t PWR_MGMNT_2 = 0x6C;
const uint8_t SEN_ENABLE = 0x00;
const uint8_t DIS_GYRO = 0x07;
const uint8_t USER_CTRL = 0x6A;
const uint8_t I2C_MST_EN = 0x20;
const uint8_t I2C_MST_CLK = 0x0D;
const uint8_t I2C_MST_CTRL = 0x24;
const uint8_t I2C_SLV0_ADDR = 0x25;
const uint8_t I2C_SLV0_REG = 0x26;
const uint8_t I2C_SLV0_DO = 0x63;
const uint8_t I2C_SLV0_CTRL = 0x27;
const uint8_t I2C_SLV0_EN = 0x80;
const uint8_t I2C_READ_FLAG = 0x80;
const uint8_t MOT_DETECT_CTRL = 0x69;
const uint8_t ACCEL_INTEL_EN = 0x80;
const uint8_t ACCEL_INTEL_MODE = 0x40;
const uint8_t LP_ACCEL_ODR = 0x1E;
```

```
    const uint8_t WOM_THR = 0x1F;
    const uint8_t WHO_AM_I = 0x75;
    const uint8_t FIFO_EN = 0x23;
    const uint8_t FIFO_TEMP = 0x80;
    const uint8_t FIFO_GYRO = 0x70;
    const uint8_t FIFO_ACCEL = 0x08;
    const uint8_t FIFO_MAG = 0x01;
    const uint8_t FIFO_COUNT = 0x72;
    const uint8_t FIFO_READ = 0x74;
    // AK8963 registers
    const uint8_t AK8963_I2C_ADDR = 0x0C;
    const uint8_t AK8963_HXL = 0x03;
    const uint8_t AK8963_CNTL1 = 0x0A;
    const uint8_t AK8963_PWR_DOWN = 0x00;
    const uint8_t AK8963_CNT_MEAS1 = 0x12;
    const uint8_t AK8963_CNT_MEAS2 = 0x16;
    const uint8_t AK8963_FUSE_ROM = 0x0F;
    const uint8_t AK8963_CNTL2 = 0x0B;
    const uint8_t AK8963_RESET = 0x01;
    const uint8_t AK8963_ASA = 0x10;
    const uint8_t AK8963_WHO_AM_I = 0x00;
    // private functions
    int writeRegister(uint8_t subAddress, uint8_t data);
    int readRegisters(uint8_t subAddress, uint8_t count, uint8_t* dest);
    int writeAK8963Register(uint8_t subAddress, uint8_t data);
    int readAK8963Registers(uint8_t subAddress, uint8_t count, uint8_t* dest);
    int whoAmI();
    int whoAmIAK8963();
};

class MPU9250FIFO: public MPU9250 {
  public:
    using MPU9250::MPU9250;
    int enableFifo(bool accel,bool gyro,bool mag,bool temp);
    int readFifo();
    void getFifoAccelX_mss(size_t *size,float* data);
    void getFifoAccelY_mss(size_t *size,float* data);
    void getFifoAccelZ_mss(size_t *size,float* data);
    void getFifoGyroX_rads(size_t *size,float* data);
    void getFifoGyroY_rads(size_t *size,float* data);
    void getFifoGyroZ_rads(size_t *size,float* data);
    void getFifoMagX_uT(size_t *size,float* data);
    void getFifoMagY_uT(size_t *size,float* data);
    void getFifoMagZ_uT(size_t *size,float* data);
    void getFifoTemperature_C(size_t *size,float* data);
  protected:
    // fifo
    bool _enFifoAccel,_enFifoGyro,_enFifoMag,_enFifoTemp;
    size_t _fifoSize,_fifoFrameSize;
    float _axFifo[85], _ayFifo[85], _azFifo[85];
    size_t _aSize;
    float _gxFifo[85], _gyFifo[85], _gzFifo[85];
    size_t _gSize;
    float _hxFifo[73], _hyFifo[73], _hzFifo[73];
    size_t _hSize;
    float _tFifo[256];
    size_t _tSize;
};

#endif
```

Library code: Kalman filter

```
/* Copyright (C) 2012 Kristian Lauszus, TKJ Electronics. All rights reserved.

 This software may be distributed and modified under the terms of the GNU
 General Public License version 2 (GPL2) as published by the Free Software
 Foundation and appearing in the file GPL2.TXT included in the packaging of
 this file. Please note that GPL2 Section 2[b] requires that all works based
 on this software must also be made publicly available under the terms of
 the GPL2 ("Copyleft").

 Contact information
 -------------------

 Kristian Lauszus, TKJ Electronics
 Web      :  http://www.tkjelectronics.com
 e-mail   :  kristianl@tkjelectronics.com
 */

#include "Kalman.h"

Kalman::Kalman() {
    /* We will set the variables like so, these can also be tuned by the user
    Q_angle = 0.001f;
    Q_bias = 0.003f;
    R_measure = 0.03f;*/

    Q_angle = 0.001f;
    Q_bias = 0.003f;
    R_measure = 0.03f;

    angle = 0.0f; // Reset the angle
    bias = 0.0f; // Reset bias

    P[0][0] = 0.0f; // Since we assume that the bias is 0 and we know the starting
angle (use setAngle), the error covariance matrix is set like so - see:
http://en.wikipedia.org/wiki/Kalman_filter#Example_application.2C_technical
    P[0][1] = 0.0f;
    P[1][0] = 0.0f;
    P[1][1] = 0.0f;
};

// The angle should be in degrees and the rate should be in degrees per second and the
delta time in seconds
float Kalman::getAngle(float newAngle, float newRate, float dt) {
    // KasBot V2  -  Kalman filter module - http://www.x-firm.com/?page_id=145
    // Modified by Kristian Lauszus
    // See my blog post for more information: http://blog.tkjelectronics.dk/2012/09/a-
practical-approach-to-kalman-filter-and-how-to-implement-it

    // Discrete Kalman filter time update equations - Time Update ("Predict")
    // Update xhat - Project the state ahead
    /* Step 1 */
    rate = newRate - bias;
    angle += dt * rate;

    // Update estimation error covariance - Project the error covariance ahead
    /* Step 2 */
    P[0][0] += dt * (dt*P[1][1] - P[0][1] - P[1][0] + Q_angle);
    P[0][1] -= dt * P[1][1];
    P[1][0] -= dt * P[1][1];
    P[1][1] += Q_bias * dt;

    // Discrete Kalman filter measurement update equations - Measurement Update
("Correct")
    // Calculate Kalman gain - Compute the Kalman gain
    /* Step 4 */
    float S = P[0][0] + R_measure; // Estimate error
    /* Step 5 */
    float K[2]; // Kalman gain - This is a 2x1 vector
    K[0] = P[0][0] / S;
    K[1] = P[1][0] / S;

    // Calculate angle and bias - Update estimate with measurement zk (newAngle)
    /* Step 3 */
    float y = newAngle - angle; // Angle difference
    /* Step 6 */
    angle += K[0] * y;
    bias += K[1] * y;

    // Calculate estimation error covariance - Update the error covariance
    /* Step 7 */
    float P00_temp = P[0][0];
    float P01_temp = P[0][1];

    P[0][0] -= K[0] * P00_temp;
    P[0][1] -= K[0] * P01_temp;
    P[1][0] -= K[1] * P00_temp;
    P[1][1] -= K[1] * P01_temp;

    return angle;
};

void Kalman::setAngle(float angle) { this->angle = angle; }; // Used to set angle, this
should be set as the starting angle
float Kalman::getRate() { return this->rate; }; // Return the unbiased rate

/* These are used to tune the Kalman filter */
void Kalman::setQangle(float Q_angle) { this->Q_angle = Q_angle; };
void Kalman::setQbias(float Q_bias) { this->Q_bias = Q_bias; };
void Kalman::setRmeasure(float R_measure) { this->R_measure = R_measure; };

float Kalman::getQangle() { return this->Q_angle; };
float Kalman::getQbias() { return this->Q_bias; };
float Kalman::getRmeasure() { return this->R_measure; };
```