Stepan Katovich

Visualizing Object’s Orientation In 3D Space

Metropolia University of Applied Sciences

Bachelor of Engineering

Electronics

Bachelor’s Thesis

17 January 2023

Tiivistelmä

Tekijä:	Stepan Katovich
Otsikko:	Kappaleen orientaation visualisointi 3D-avaruudessa
Sivumäärä:	29 sivua + 16 liitettä
Aika:	31.1.2023

Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Sähkö- ja automaatiotekniikka
Ammatillinen pääaine:	Elektroniikka
Ohjaajat:	Yliopettaja Heikki Valmu

Tämän opinnäytetyön tarkoituksena oli luoda järjestelmä, joka seuraa ja visualisoi kappaleen asentoa kolmiulotteisessa avaruudessa inertiamittausyksikön avulla. Työn taustalla on henkilökohtainen kiinnostus aihetta kohtaan ja vastaavanlaisen systeemin laajat käyttömahdollisuudet muun muassa navigointijärjestelmissä, ilmailussa, robotiikassa, autoteollisuudessa ja viihde-elektroniikassa.

Projektin pääpaino oli visualisointiohjelmiston, mikro-ohjaimen laiteohjelmiston ja sensorifuusioalgoritmin toteuttamisessa. Projektin aikana haasteiksi muodostuivat muun muassa meluisan datan käsittely sekä kolmiulotteisen asennon arvioinnin monimutkaisuudet, kuten gimbaalilukko. Projekti tarjosi syvällistä ymmärrystä kappaleen asennon arviointiin liittyvistä konsepteista, kuten antureista sekä anturifuusion menetelmistä. Lisäksi projekti antoi käytännön kokemusta ohjelmoinnista ja datan keruusta sekä käsittelystä.

Projektin tuloksena syntyi toimiva järjestelmä, joka onnistuneesti seuraa ja esittää kappaleen asentoa 3D-avaruudessa. Luotu järjestelmä on hyvä pohja jatkokehitykselle ja muille aiheeseen liittyville projekteille. Systeemiä voi edelleen parantaa lisäämällä siihen magnetometri sekä soveltamalla tehokkaampia fuusioalgoritmeja.

Avainsanat: imu, asennon estimointi, sensorifuusio

Abstract

Author: Stepan Katovich
Title: Visualizing Object’s Orientation In 3D Space
Number of Pages: 29 pages + 16 appendices
Date: 17 January 2023

Degree: Bachelor of Engineering
Degree Programme: Degree Programme in Electronics
Professional Major: Electronics
Supervisors: Heikki Valmu, Principal Lecturer

The purpose of this thesis was to create a system that tracks and visualizes an orientation of an object in three-dimensional space using an inertial measurement unit. The project was motivated by a personal interest in inertial measurement units and and their extensive use in fields such as navigation systems, aviation, robotics, automotive and consumer electronics.

The main focus of the project was on the implementation of the visualization software, microcontroller firmware and a sensor fusion algorithm. The challenges encountered during project included dealing with noisy data and the complexities of three-dimensional attitude estimation, such as gimbal lock. The project delved into sensor-related concepts such as sensor fusion, data collection and data analysis, while also offering a hands-on learning experience in programming.

As the result of the project, a functioning system was developed that successfully tracks and displays the orientation of an object in three-dimensional space. While there is still room for improvement, the created system is a solid foundation for further development and other projects related to the topic.

Keywords: sensor fusion, imu, attitude estimation

Contents

List of Abbreviations

1 Introduction....1

2 Theoretical Background....2

2.1 Attitude Estimation....2
2.2 Inertial Measurement Unit and MEMS Sensors....4
2.2.1 Accelerometer....5
2.2.2 Gyroscope....5
2.2.3 Magnetometer....7
2.3 Sensor Fusion....8
2.4 Basic Complementary Filter....9

3 Materials and Setup....10

3.1 Arduino UNO....10
3.2 Adafruit 9-DOF Breakout Board....11
3.2.1 LSM303DLHC....11
3.2.2 L3GD20H....12
3.3 Connecting Arduino and Adafruit 9-DOF....13

4 Arduino Firmware....14

4.1 Wire Library....14
4.2 Controlling Sensor Behavior and Parameters....15
4.3 Reading Sensor Output....16
4.4 Zero-Rate Level Correction....19
4.5 Quaternion-Based Complementary Filter....19

5 Visualization Software....21

6 Measurements and Discussion....22

7 Conclusions....25

References....26

Appendices

Appendix 1: Arduino Code
Appendix 2: Quaternion-Based Complementary Filter Code
Appendix 3: Visualizer Code

List of Abbreviations

DOF: Degrees of freedom

DPS: Degrees per second

HPF: High-pass filter

I2C: Inter-Intergrated Circuit

IC: Intergrated circuit

IMU: Inertial measurement unit

LPF: Low-pass filter

LSB: Least significant bit

MEMS: Microelectromechanical systems

MSB: Most significant bit

SCL: Serial clock

SDA: Serial data

1 Introduction

Estimating the attitude of a body, also known as orientation, is a crucial task in many applications like navigation, guidance, control of drones, automated vehicles and many other. The demand for such systems is increasing as the use of autonomous agents increases in our everyday life. However, determining the attitude of a body is a challenging task. It requires handling noisy data and dealing with sensor measurement errors. Various methods are available to overcome these challenges such as Kalman filtering, Madgwick filtering, and complementary filtering. These methods are based on combining data from multiple sensors and mathematical models to estimate the attitude.

The goal of this project was to accurately track the attitude of an object in three-dimensional space using an inertial measurement unit and to visualize the orientation. The project was divided into three parts: data collection, creating a visualizer software and implementing attitude estimation algorithm. Additionally, this thesis gives a brief overview of the working principles behind sensors used in the project, as well as an explanation of the concepts and techniques applied in the implementation.

2 Theoretical Background

2.1 Attitude Estimation

Attitude estimation is the process of estimating the orientation of an object relative to a known reference frame. Reference frame is some coordinate system that describes object's position and orientation in three-dimensional space. Different types of frames such as sensor frame (Figure 1), body frame (Figure 2) and Earth frame (Figure 3), all can be used as a reference frame. [1.]

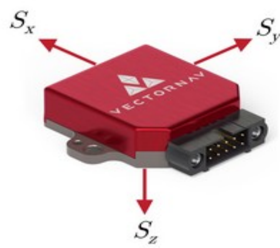

Figure 1: Sensor frame [1].

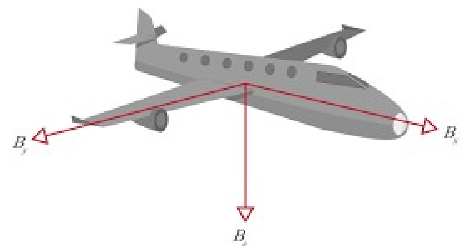

Figure 2: Body frame [1].

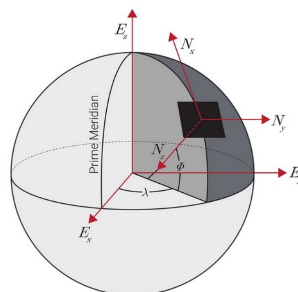

Figure 3: Earth frame (Ex, Ey, Ez) and North-East-Down frame (Nx, Ny, Nz) [1].

Two popular methods to represent orientation of a body are using Euler angles or using rotation quaternions. Euler angles (see Figure 4) is a set of three angles pitch (φ), roll (θ), and yaw (ψ). These angles represent the rotations around the axes x, y and z. [2.]

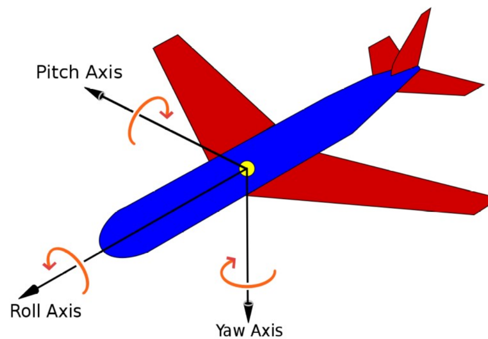

Figure 4: Euler angles [27].

The rotations are sequentially applied to the chosen body frame in a specific order, setting the body frame into new orientation after each rotation. Order is given by the set, for example (3-2-1) is equivalent to yaw-pitch-roll and (1-2-3) is equivalent to roll-pitch-yaw orders. As demonstrated in Figure 5, the end result is dependant on the order. [1; 2.]

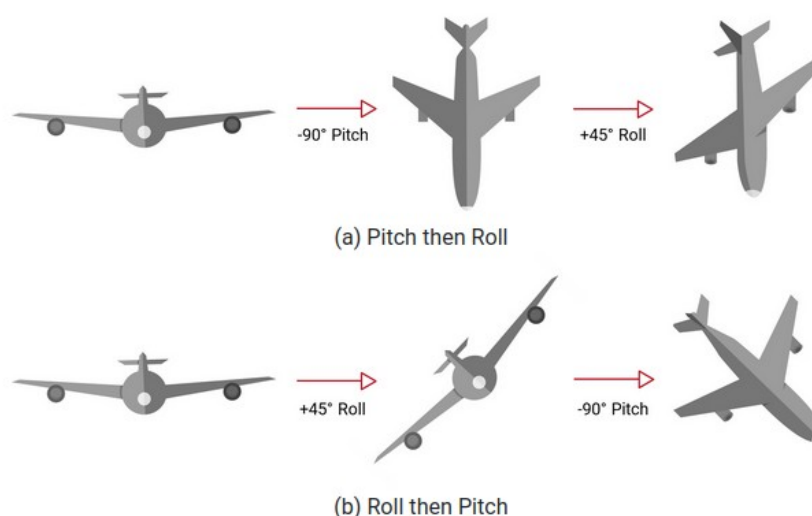

Figure 5: Euler angle order example [2].

When using Euler angles, the possibility of gimbal lock arises. Gimbal lock occurs when two axes become aligned, resulting in the body getting locked into a single orientation. This causes a loss of one degree of freedom (DOF) and leads to incorrect representation of certain orientations. To overcome this issue, one approach is to use rotation quaternions as an alternative to Euler angles. Quaternions are a mathematical construct that can be used to represent three-dimensional orientation and rotations. Quaternions have a scalar component and a vector component. The scalar component specifies the angle of rotation around a specific axis defined by the vector component (Figure 6). [2; 3.]

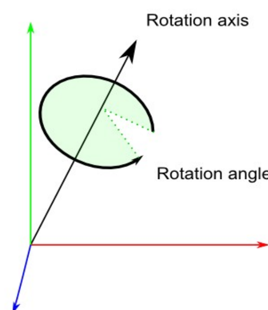

Figure 6: Quaternion representation of rotation [28].

2.2 Inertial Measurement Unit and MEMS Sensors

Inertial measurement unit is a device that combines multiple sensors that can be used to determine orientation. A 9-DOF IMU (see Figure 13) is capable of measuring the linear acceleration, angular velocities and magnetic field, as it contains a 3-axis accelerometer, 3-axis gyroscope and 3-axis magnetometer. [4.]

Microelectromechanical systems (MEMS) refer to the technology of tiny mechanical devices and systems on a microchip. Many IMUs include MEMS sensors to measure acceleration, angular velocities and magnetic fields. There are two basic types of MEMS sensing principles: capacitive and ohmic. Majority

of MEMS accelerometers, gyroscopes and magnetometers use capacitive sensing principle. Capacitive sensing principle is based on the displacement of a proof mass, usually a metal plate suspended by small beams, in an electric field. When a force is applied to the proof mass, it changes the capacitance between the proof mass and the electrodes. The change in capacitance is detected by the electronics on the microchip, which is then converted into voltage and processed by the electronics to measure the acceleration, angular rate or strength of magnetic field. [5; 6.]

2.2.1 Accelerometer

An accelerometer measures linear acceleration along one or several of the x, y and z axis. Single accelerometer's internal structure consists of a proof mass and fixed electrodes (Figure 7). The proof mass is connected to the housing of the device by suspension beams or springs, allowing the mass to move in response to acceleration. This movement can be detected as change in capacitance. The acceleration is measured in units of g, which is the acceleration due to gravity. [5, 2-6; 6.]

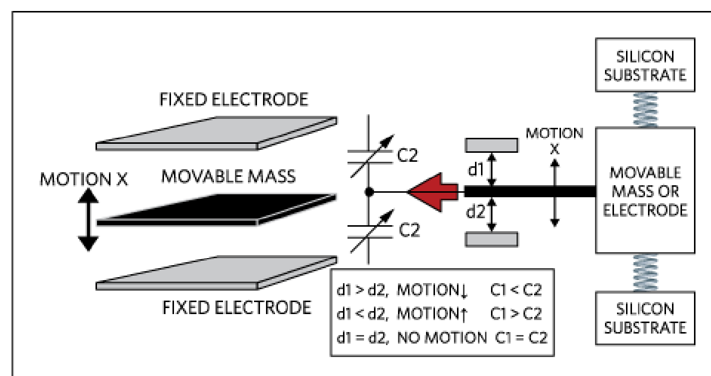

Figure 7: Acceleration associated with a single moving mass [5].

2.2.2 Gyroscope

Gyroscopes are devices that measure angular velocities (how fast a body is rotating around a particular axis). Angular velocity's unit is either degrees or radians per second. To determine angular velocity many MEMS gyroscopes make use of vibrating structrure that is based on Coriolis effect. Coriolis force is a force that acts on moving body on a rotating plane. Figure 8 shows that when body is moving towards the center of the rotation the Coriolis force (represented by green arrow) is exerted to the right. When body is moving outwards of the center, the force is exerted to the left. [6; 7.]

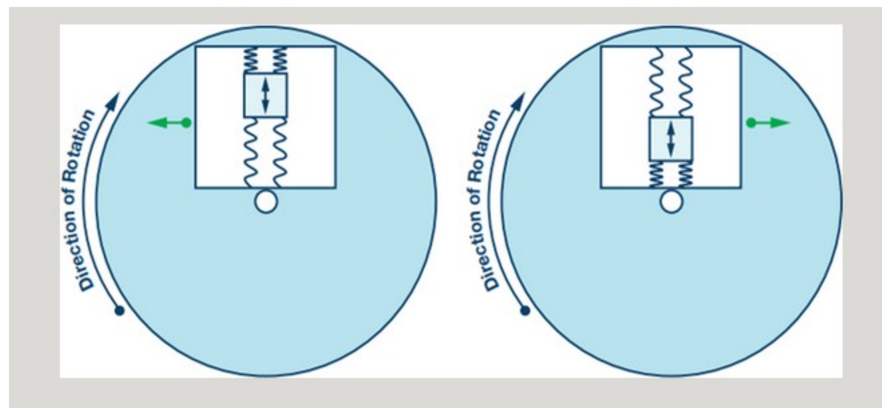

Figure 8: Demonstration of Coriolis effect [7].

IMU that was used in this project is equipped with a MEMS gyroscope L3GD20H. According to documentation on Adaruit webpage [8, 3] L3GD20 uses a 'tuning fork' structure (Figure 9). Some tuning fork structures consist of two identical tuning forks that are mounted parallel to each other, with a small gap between them. Each tuning fork is driven to vibrate at its resonant frequency by an electrical current. This is called the 'drive mode'. When the gyroscope is rotated about its axis, the Coriolis force causes the tines to vibrate out of phase with each other and cause a change in the capacitance. This is called the 'detection mode'. [6; 9.]

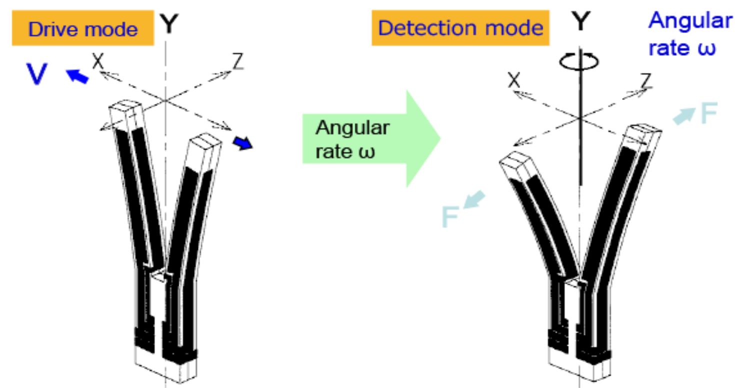

Figure 9: Example of tuning fork structure [9].

2.2.3 Magnetometer

Most MEMS magnetometer designs are based on Hall Effect, magneto-resistance and the fluxgate effects. In this project LSM303DLHC magnetometer was used. According to documentation on Adafruit webpage [11, 7] LSM303DLHC magnetometer is a Lorentz-force-based sensor. Lorentz-type sensors rely on mechanical motion of the MEMS structure under the Lorentz force, that is generated by an excitation current flowing through the coils in the magnetic field. Figure 10 displays the operation principle of the MEMS torsional resonant magnetometer that utilizes the Lorentz-force. A small silicon resonator is suspended by torsional beams. The resonator is exposed to an external magnetic field and the Lorentz force generated by the field causes the resonator to move. The motion is sensed by a capacitive system. The measured strength of magnetic field is given in gauss. [6; 10.]

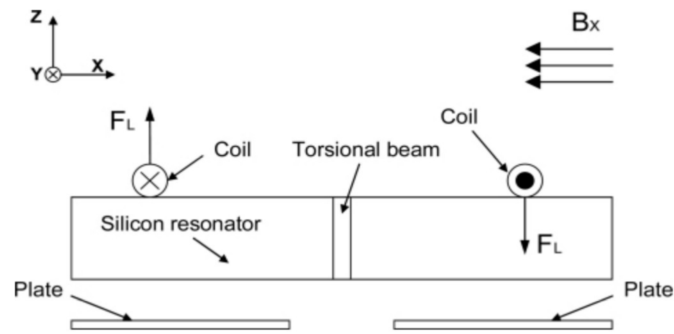

Figure 10: Operation principle of the MEMS torsional resonant magnetometer [10].

2.3 Sensor Fusion

Gyroscopes, accelerometers and magnetometers have limitations that affect their performance. The gyroscope suffers from drift. That means that the value measured by the gyroscope sensor will gradually change over time, even if the device is not moving. The accelerometer and magnetometer sensors have a higher noise error density than the gyro, but they do not suffer from drift [12]. Limitations of IMU sensors listed in Table 1 below:

Table 1: Limitations of accelerometer, gyroscope and magnetometer.

Sensor	**Limitation**
Gyroscope	Drift
Magnetometer	Sensitive to magnetic noise
Accelerometer	Sensitive to vibration and mechanical noise

Sensor fusion techniques combine data from multiple sensors to estimate a more accurate and reliable result. One such technique is complementary filtering. Complementary filtering is a method of integrating data from two sensors, such as a gyroscope and accelerometer, over time to improve the

accuracy of the estimate. It is a good choice for its simplicity and low computational cost. Another popular fusion technique is Kalman filtering. It is more complex in nature than complementary filter, but often offers better accuracy and better noise reduction. [12; 13.]

2.4 Basic Complementary Filter

This technique uses a low-pass filter (LPF) to calculate the slow-varying component of the signal and a high-pass filter (HPF) to calculate the fast-varying component. Figure 11 shows complementary filter used with gyroscope and accelerometer data:

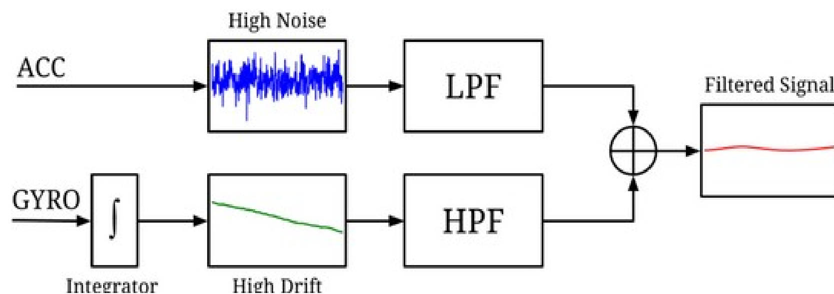

Figure 11: Schematic of complementary filter [14, 8].

The general form of complementary filter is shown in equation (1) with the data from gyroscope and acceleromter represented as gyro and acc, coefficient α as a gain coefficient (typically chosen to be greater than 0.90), dt as the sampling period and angle at time t is the result given as Euler angle.

$$angle_t = \alpha \cdot (angle_{t-1} + gyro_{data} \cdot dt) + (1 - \alpha) \cdot (acc_{data}) \quad (1)$$

By pairing the gyroscope with the accelerometer, we can only achieve the corrected pitch and roll values. In order to calculate the corrected yaw angle a similar filter could be implemented, where the low-pass portion would act on magnetometer readings instead of accelerometer. [12; 14.]

3 Materials and Setup

3.1 Arduino UNO

Arduino is an open-source hardware and software company that offers various development kits used in many electronics projects. Their Arduino UNO board (Figure 12) was selected as a microcontroller for this project. It features a simple USB interface that allows it to be programmed using computer and a number of pins to both power the sensors and retrieve the output data from them. Arduino offers a development software called Arduino IDE that allows to write, compile and upload code to the board, and to also monitor the serial data using its Serial Monitor function.

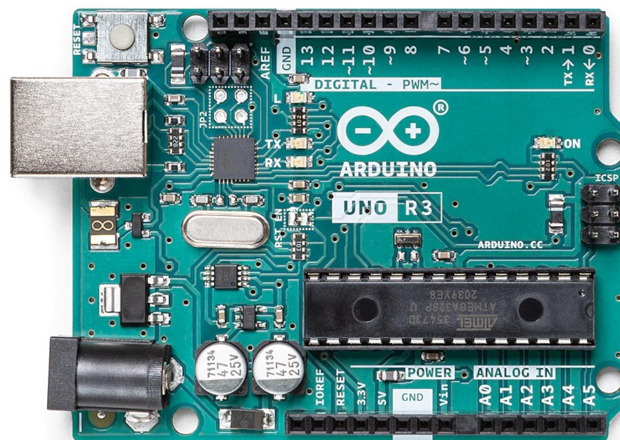

Figure 12: Arduino UNO development board. Image copied from https://docs.arduino.cc/static/9fab734836cc264a392563733be102ed/A000066-datasheet.pdf

3.2 Adafruit 9-DOF Breakout Board

Adafruit is an open-source hardware company that manufactures and sells electronics related products. Adafruit 9-DOF breakout board (Figure 13) is a sensor board that combines two separate intergrated circuits (IC): the L3GD20H 3-axis gyroscope and the LSM303DHLC 3-axis accelerometer and magnetometer. The breakout board allows you to communicate with these sensors using a microcontroller, such as an Arduino via a two-pin Inter-Integrated Circuit (I2C) serial communication protocol. [15.]

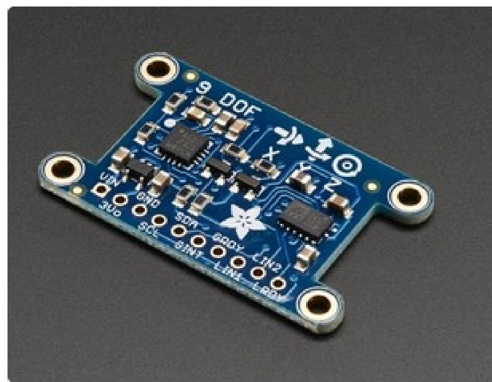

Figure 13: Adafruit 9-DOF IMU breakout board [15].

3.2.1 LSM303DLHC

LSM303DLHC combines 3-axis accelerometer (Figure 14) and 3-axis magnetometer (Figure 15) sensors in one package. A combination of accelerometer and magnetometer allows the sensor to be used as a tilt-compensated compass, that is a compass that can determine heading regardless of board inclination. LSM303DLHC outputs a 16-bit reading per axis and provides measurement range of ±2g / ±4g / ±8g / ±16g for the accelerometer and full scale of ±1.3 / ±1.9 / ±2.5 / ±4.0 / ±4.7 / ±5.6 / ±8.1 gauss for magnetometer. [16.]

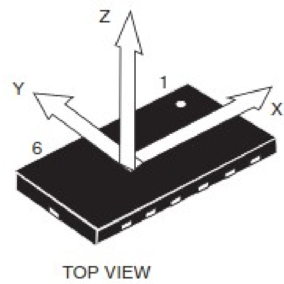

Figure 14: Direction of detectable accelerations [21].

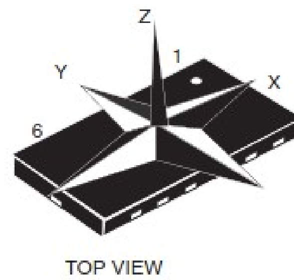

Figure 15: Direction of detectable magnetic fields [21].

3.2.2 L3GD20H

The L3GD20H is a 3-axis gyroscope that measures the angluar rates of rotation about the roll (x), pitch (y) and yaw (z) axes (Figure 16). It outputs a 16-bit reading per axis and has a full scale of ±245/±500/±2000 degrees per second (dps). [17.]

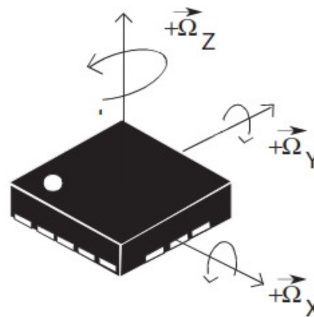

Figure 16: L3GD20H directions of the detectable angular rate [22].

3.3 Connecting Arduino and Adafruit 9-DOF

The Adafruit 9-DOF board was connected to Arduino UNO using four wires as shown by Figure 17:

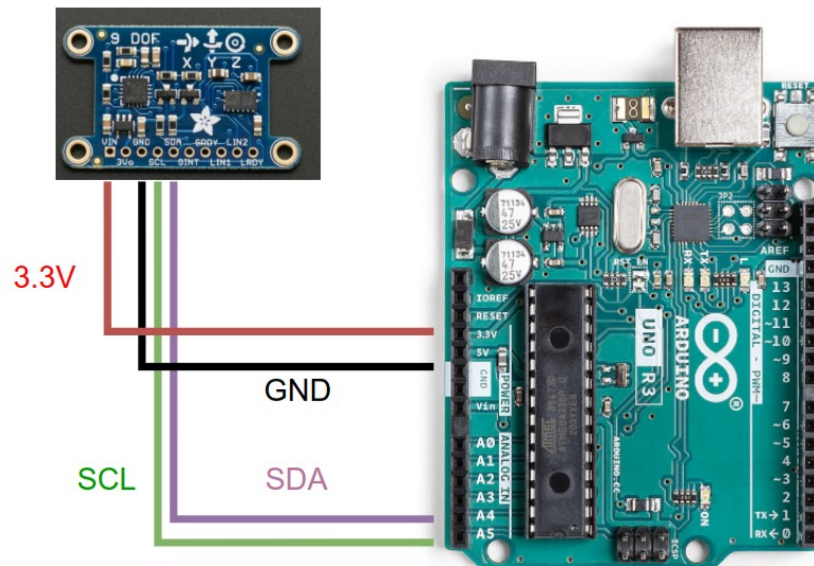

Figure 17: Arduino and 9-DOF Adafruit IMU connection.

The serial data (SDA) and serial clock (SCL) wires are used to establish I2C communication between Arduino and Adafruit 9-DOF board. I2C a synchronous master/slave bus. Communication is synchronized using the SCL line that transmits a clock signal generated by master device (Arduino), while the SDA wire transmits the actual data between the two devices. Once everything is wired, Arduino can be connected to a computer via USB and Arduino IDE could then be used to configure the sensors and to monitor the output using the serial monitor.

4 Arduino Firmware

In the context of this project, the firmware refers to the program that runs on the microcontroller and is responsible for controlling the sensors and fetching and processing data. The following sections of this chapter will explain the basic concepts and techniques used to implement the firmware. Some listings in the sections display modified snippets from the actual code. For the full code see Appendix 1.

4.1 Wire Library

To communicate with I2C devices a library called Wire.h was used [18]. This library provides an easy-to-use interface for sending and receiving data over I2C communication protocol. Listing 1 demonstrates how to use the Wire library to read and write a single byte from/to a register on a I2C device:

```
byte value;                   // a variable that stores the byte to be
                                 read or written

Wire.begin();                 // initialize I2C communication

// write a byte

Wire.beginTransmission(device_i2c_address); // begin i2c transmission
                                               with the device
Wire.write(reg);              // select the register to be written to
Wire.write(value);            // write a value to the register
Wire.endTransmission();       // end transmission


// read a byte

Wire.beginTransmission(device_i2c_address);
Wire.write(reg);              // select the register to be read from
Wire.endTransmission();
Wire.requestFrom(device_i2c_address, 1);   // request 1 byte from the
                                              register

while (Wire.available() < 1);     // wait until 1 byte received

value = Wire.read();              // store the byte in value
```

Listing 1: Reading and writing a single byte using Wire library.

4.2 Controlling Sensor Behavior and Parameters

Sensor behavior is controlled by a set of 8-bit registers. Each register is identified by a 7-bit address. Writing values to these registers will set or change various parameters such as power states, data rate and sensitivity. Register descriptions can be found in datasheets [21; 22].

Each sensor is assigned its own unique I2C address for communication. Adafruit device address list [19] shows that the default I2C addresses for LSM303DLHC and L3GD20H are 0x19 (accelerometer), 0x1E (magnetometer) and 0x6A or 0x6B (gyroscope). To verify these addreses, I2C scanner program [20] was uploaded to Arduino to print out all the devices connected to the I2C bus. The detected devices are listed in Figure 18 below:

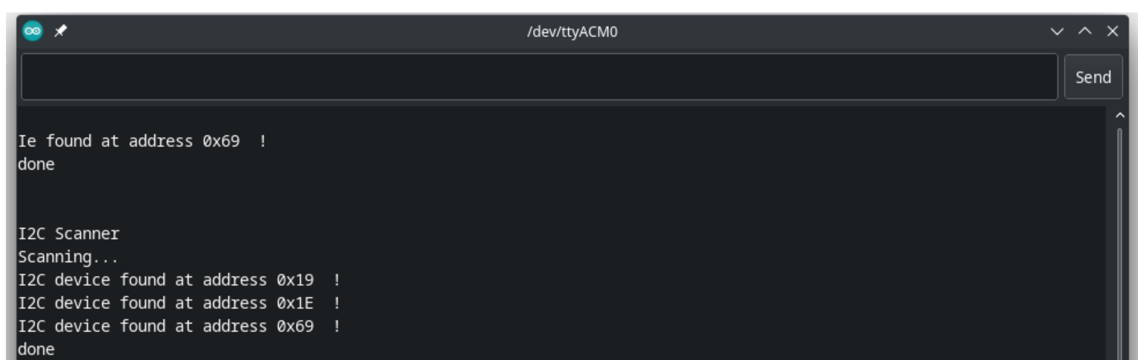

Figure 18: Result of I2C device scan.

From the figure, it appears that the address of the L3GD20H sensor is seen at 0x69, rather than the expected 0x6A or 0x6B. However, in this case, 0x69 has been proven to be the correct address for gyroscope.

According to Pololu [16; 17] sensors LSM303DLHC and L3GD20H are powered off by default and need to be turned on by writing to their control registers. To activate accelerometer, control register CTRL_REG1_A at address 0x20 is set to value 0x57. This enables measurements on x, y and z axes and

sets the data output rate to 100 Hz [21, 25]. To activate magnetometer, value 0x00 is set to register MR_REG_M at address 0x02 and value 0x10 is set to control register CRA_REG_A at address 0x00. This puts the magnetometer into continous-conversion mode and sets the data output rate to 15 Hz [21, 37-38]. Finally gyroscope is enabled by writing value 0x0F to register CTRL1 at address 0x20. This turns on the sensor in normal mode, enables measurement on x, y and z axes and sets data rate to 100 Hz [22, 36-37]. All other registers were left at their default values.

4.3 Reading Sensor Output

As seen in LSM303DLHC datasheet [21, 29] data collected by accelerometer is stored in six 8-bit output registers (Figure 19). The six registers are paired so that each pair stores data for single axis (16 bits per axis) as a two's complementary (a format used to represent negative numbers). The same applies for the magnetometer [21, 39] and the gyroscope [22, 42].

7.1.9 OUT_X_L_A (28h), OUT_X_H_A (29h)

X-axis acceleration data. The value is expressed in two's complement.

7.1.10 OUT_Y_L_A (2Ah), OUT_Y_H_A (2Bh)

Y-axis acceleration data. The value is expressed in two's complement.

7.1.11 OUT_Z_L_A (2Ch), OUT_Z_H_A (2Dh)

Z-axis acceleration data. The value is expressed in two's complement.

Figure 19: Output registers for LSM303DLHC accelerometer. [21, 29.]

To make reading data more efficient multiple bytes can be read by setting the most significant bit (MSB) of the register address to 1 [21, 21; 22, 29]. This enables auto increment feature. For magnetometer auto increment is enabled by default [21, 22]. When using auto-increment, it is important to fill the variables in the order specified by the datasheet [21; 22]. Listing 2

demonstrates how to read data from LSM303DLHC accelerometer output registers:

```
Wire.beginTransmission(0x19);    // select LSM303DLHC accelerometer
Wire.write(0x28 | 0x80);         // select the first output register
                                    0x28 and enable auto-increment
Wire.endTransmission();

Wire.requestFrom(0x19, 6);       // request 6 bytes (1 byte = 8 bits)

while (Wire.available() < 6);    // wait until 6 bytes received

xlo = Wire.read();    // low 8 bits of x-axis data from 0x28
xhi = Wire.read();    // high 8 bits of x-axis data from 0x29
ylo = Wire.read();    // low 8 bits of y-axis data from 0x2A
yhi = Wire.read();    // high 8 bits of y-axis data from 0x2B
zlo = Wire.read();    // low 8 bits of z-axis data from 0x2C
zhi = Wire.read();    // high 8 bits of z-axis data from 0x2D
```

Listing 2: Reading output from LSM303DLHC accelerometer.

The 16-bit output value for each axis is obtained by left shifting the high byte of the data by 8 bits and then using the bitwise OR operator to copy the the bits from low byte, as shown in Listing 3 below:

```
raw_x = (int16_t)(xlo | (xhi << 8));   // 16 bit output x-axis
raw_y = (int16_t)(ylo | (yhi << 8));   // 16 bit output y-axis
raw_z = (int16_t)(zlo | (zhi << 8));   // 16 bit output z-axis
```

Listing 3: Transforming 6 bytes into three 16-bit outputs using bit manipulation.

According to protocol hints given by Pololu [16] both magnetometer and accelerometer have only 12-bit resolution when running in normal mode. For the accelerometer at least 4 least significant bits (LSB) of the output are always 0 and for magenetometer 4 highest bits are always 0. Therefore to obtain the original 12-bit data from the accelerometer, the 16-bit value must be right shifted by 4 bits.

The raw output as such is not yet useful and has to be converted to the unit of measurement. For accelerometer the data was converted to g's, for magnetometer the data was converted to gauss and for gyroscope the data was converted to degrees per second (later to radians per second). The conversion is performed using a conversion factor, which is found in the datasheet for the sensor [21; 22]. Conversion factor represents the smallest change detected by the sensor and is determined by the selected sensitivity scale of the device. The smaller the selected scale the more sensitive the device becomes to smaller forces. For accelerometer and gyroscope the default scales are set to ± 2 g and 245 dps respectively as shown by the register mapping [21, 24; 22, 34]. For magnetometer the default scale is ± 1.3 gauss [21, 24]. Based on these scales the conversion factors are: 1 mg/LSB for accelerometer, 1100 LSB/gauss for magnetometer x/y axes, 980 LSB/gauss for magnetometer z axis and 8.75 mdps/digit for gyroscope [21, 10; 22, 10]. To calculate the final values, the accelerometer and gyroscope raw data was multiplied by the conversion factor, while magnetometer raw data was divided by it, as shown in Listing 4, where the operator is determined by the conversion unit (LSB/unit or unit/LSB).

```
// raw accelerometer data to g's conversion when selected scale is ±2
g
data_x = raw_x * (1 * 10^-3);
data_y = raw_y * (1 * 10^-3);
data_z = raw_z * (1 * 10^-3);

// raw magnetometer data to gauss conversion when selected scale is
±1.3 gauss
data_x = raw_x / 1100;
data_y = raw_y / 1100;
data_z = raw_z / 980;

// raw gyroscope data to dps conversion when selected range is 245 dps
data_x = raw_x * (8.75 * 10^-3);
data_y = raw_y * (8.75 * 10^-3);
data_z = raw_z * (8.75 * 10^-3);
```

Listing 4: Raw data conversion to units LSM303DLHC and L3GD20H.

4.4 Zero-Rate Level Correction

Sensors have internal offset that is measured even when sensor is completely stationary. This offset is refered to as zero-rate level by the datasheets [21, 15; 22, 16.]. An accelerometer in a steady-state should output 0 g on X and Y axes and 1 g on Z axis, whereas gyroscope and magnetometer readings in a steady-state should be 0 on all axes. This is typically achieved by measuring the offset of the sensor while it is stationary and then subtracting that value from all subsequent readings. Accelerometer and gyroscope calibration was done by taking 100 readings while the sensor is stationary and then averaging them to obtain the zero-rate level, which was then substracted from the subsequent readings.

For magnetometers the process is a bit more complex. It involves correcting the hard-iron and soft-iron offsets. Hard-iron offset is caused by external distortions such as magnets and soft-iron offsets are commonly caused by metals. In theory, hard-iron distortions can be corrected using the same averaging method as for gyroscope and accelerometer, but soft-iron offset is usually corrected more advanced tools like matrices. If not calibrated properly, the magnetometer readings cannot not be fused with other sensor data for accurate orientation estimation. [23; 24.]

4.5 Quaternion-Based Complementary Filter

As outlined in Section 2.3, to accurately determine the orientation of an object, the data from sensors such as gyroscopes and accelerometers must be ”fused”. In this work, Philip Schmidt’s complementary filter [25] was used to fuse the sensor data. Its basic operation is illustrated in Figure 20. The filter corrects the drift errors present in the gyroscope data with the help of accelerometer data in the pitch and roll directions. The system can also be extended to correct drift in the yaw direction by including magnetometer data. Schmidt [25] describes his

method as computionally effective and fully 3D capable while also avoiding gimbal locks as it is based on quaternions.

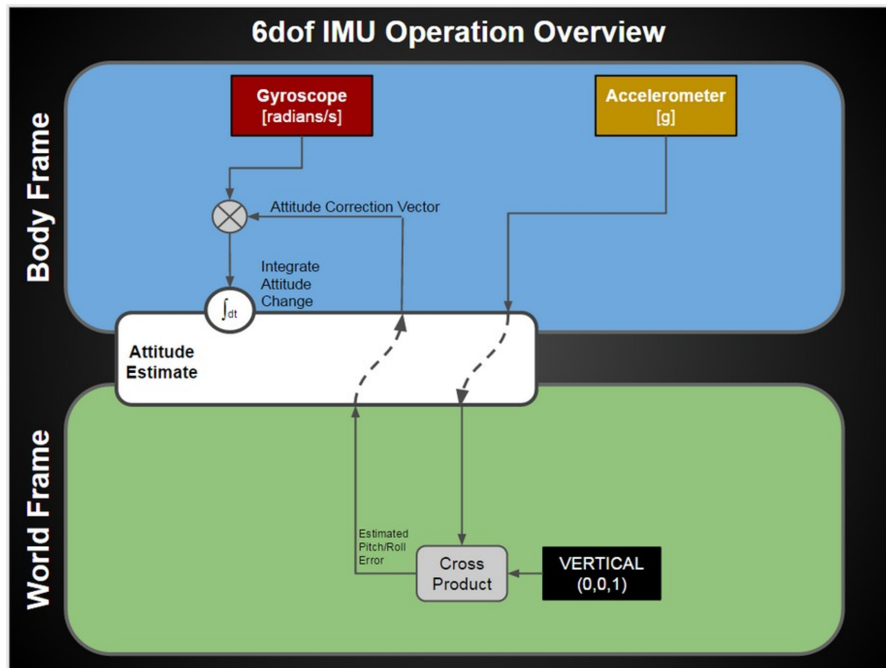

Figure 20: Basic operation of Schmidt's quaternion based complementary filter [25].

The algorithm for compensating gyroscope drift using an accelerometer can be summarized as follows: The filter receives the gyroscope and accelerometer data and stores it in a 3D vector structure as radians per second and g's respectively. The accelerometer vector is then rotated to the Earth frame and cross product between the rotated accelerometer data and vertical Earth vector (0, 0, 1) is taken to estimate the error caused by noise. The corrected vector is rotated back to the body frame, and added to the gyroscope reading. A quaternion is then created based on the corrected gyroscope reading to approximate the change in attitude , and finally combined with the previous attitude estimate by quaternion multiplication as shown in equation (2), where t is sampling period dt / 2, q_{prev} is the previous quaternion estimation and gx, gy and gz are the gyroscope readings on x, y and z axis in radians per second. [25, 26.]

$$\begin{aligned} qw &= qw_{prev} - t \cdot gx \cdot qx_{prev} - t \cdot gy \cdot qy_{prev} - t \cdot gz \cdot qz_{prev} \\ qx &= qx_{prev} + t \cdot gx \cdot qw_{prev} - t \cdot gy \cdot qz_{prev} + t \cdot gz \cdot qy_{prev} \\ qy &= qy_{prev} + t \cdot gx \cdot qz_{prev} + t \cdot gy \cdot qw_{prev} - t \cdot gz \cdot qx_{prev} \\ qz &= qz_{prev} - t \cdot gx \cdot qy_{prev} + t \cdot gy \cdot qx_{prev} + t \cdot gz \cdot qw_{prev} \end{aligned} \quad (2)$$

Lastly, the result quaternion is normalized to minimize errors introduced by any previous operations perfromed on the quaternion. Full code implementation can be seen in Appendix 2.

5 Visualization Software

The visualization software, was created to display the orientation of an IMU device. It was written in C++ and it utilizes the OpenGL library to graphically represent the orientation of the IMU board as a 3D rectangle shape. The use of OpenGL functions in the software allows for low-level access to the graphics hardware, which results in high-performance visualization of the IMU data. The software was designed to read the orientation data from Arduino through the serial port and use that data to rotate and position the 3D rectangle accordingly, giving a clear and accurate representation of the device's current orientation. The orientation is given using quaternions, consisting of w, x, y and z values, where w represents the angle of rotation and x, y and z represent the rotation axis, as displayed in Figure 21.

```
w: 0.940000 x: 0.170000 y: -0.260000 z: -0.150000
w: 0.940000 x: 0.180000 y: -0.230000 z: -0.160000
w: 0.950000 x: 0.190000 y: -0.210000 z: -0.160000
w: 0.950000 x: 0.200000 y: -0.190000 z: -0.170000
w: 0.950000 x: 0.200000 y: -0.160000 z: -0.170000
w: 0.960000 x: 0.200000 y: -0.140000 z: -0.170000
w: 0.960000 x: 0.190000 y: -0.110000 z: -0.160000
w: 0.970000 x: 0.190000 y: -0.080000 z: -0.160000
w: 0.970000 x: 0.190000 y: -0.050000 z: -0.160000
w: 0.970000 x: 0.190000 y: -0.030000 z: -0.160000
```

Figure 21: Attitude estimates in quaternions.

Once the rotation quaternion is parsed from the serial port, it is then converted into a 3D rotation matrix using equation (3) [3]. The rotation matrix is then

multiplied with the original matrix representing the 3D shape to rotate it to its new position. Full code displayed in Appendix 3.

$$R = \begin{bmatrix} 1 - 2\left(y^2 - z^2\right) & 2\left(xy - wz\right) & 2\left(xz + wy\right) & 0 \\ 2\left(xy + wz\right) & 1 - 2\left(x^2 + z^2\right) & 2\left(yz - wx\right) & 0 \\ 2\left(xz - wy\right) & 2\left(yz + wx\right) & 1 - 2\left(x^2 + y^2\right) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

6 Measurements and Discussion

To provide insight into the effectiveness of the filtering, measurements were performed in two scenarios. In the first scenario, the device was set up vertically on top of a table in a resting position and quaternion data was recorded over a short period of time (around 150 seconds). Figure 22 displays the results when quaternion values are generated solely from the gyroscope data, and it is evident that the values begin to drift over time.

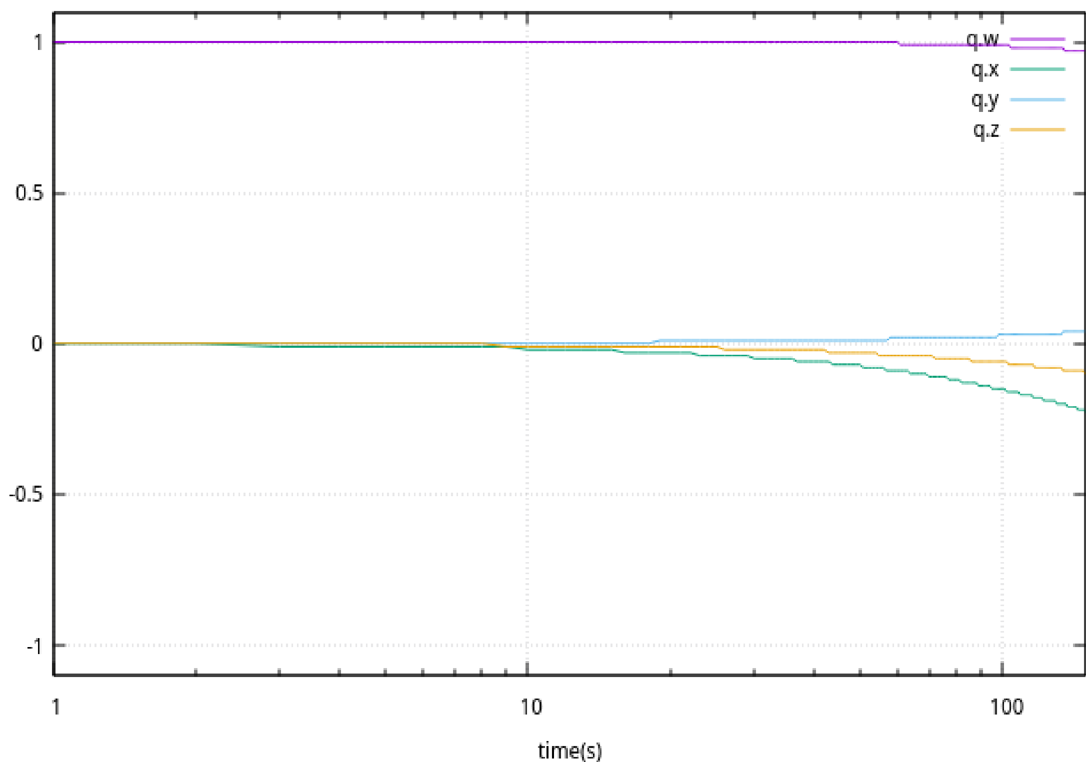

Figure 22: Quaternion attitude estimates based on gyroscope values when the device is at rest.

Figure 23 reveals the same scenario with complementary filtering applied, and the drift in w, x and y values is no longer present, as it is countered with accelerometer readings. However, some drift is still seen in the z-value.

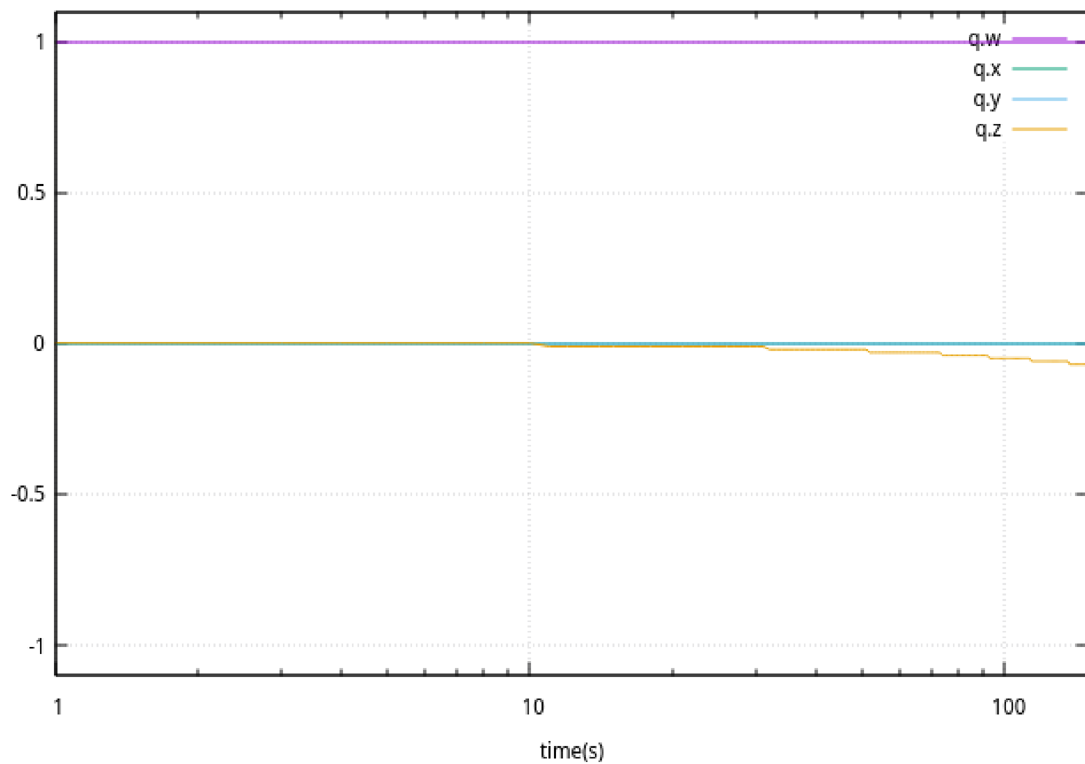

Figure 23: Complementary filtered quaternion attitude estimates when the device is at rest.

In the second scenario, the device is rotated in different directions before being returned to its original resting position. Figure 24 shows that the filtered quaternion is able to recover its w, x and y values when placed back into resting position, while Figure 25 reveals that the quaternion based solely on the gyroscope data is not able to identify the resting position, similar to the first scenario.

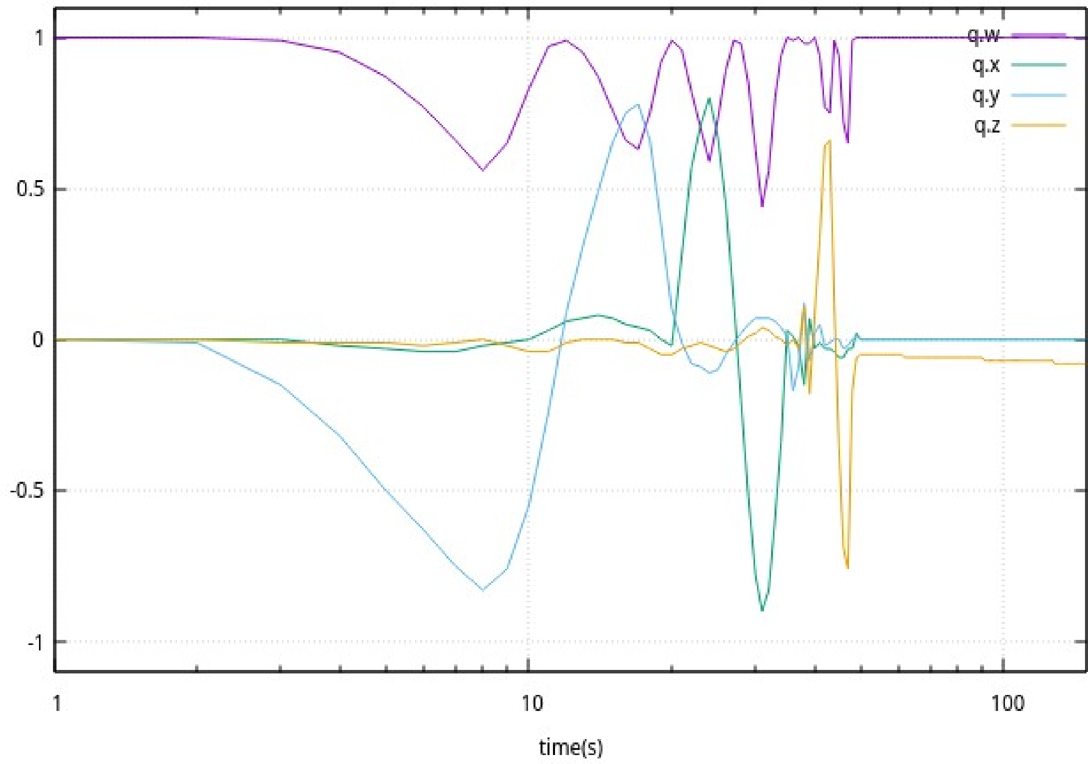

Figure 24: Complementary filtered quaternion attitude estimates when the device is in motion before being placed back to rest position.

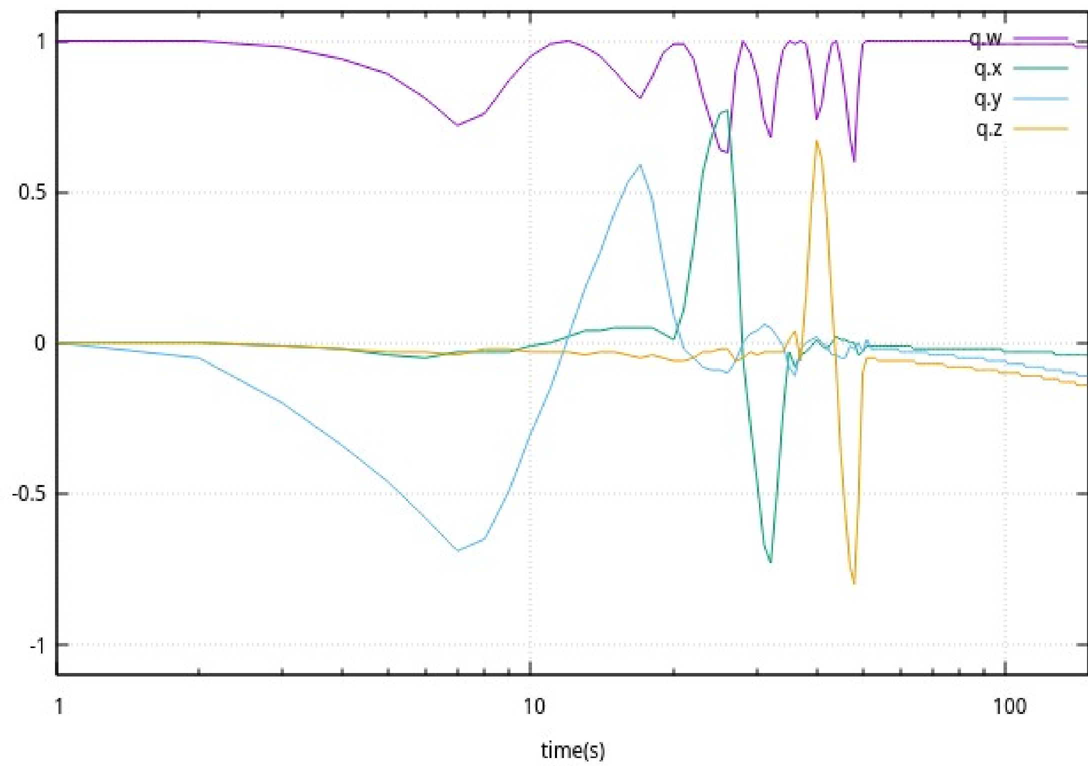

Figure 25: Quaternion attitude estimates based on gyroscope values when the device is in motion before being placed back to rest position.

The measurements show, that although current filtering significantly reduces the drift in the quaternion values, the system may still not be able to fully return the device's orientation to its resting position, due to the lack of magnetometer readings for additional corrections.

Despite its limitation, the system can still be considered effective at determining device’s orientation. During validation of the final version, the visualizer software was consistently and accurately displaying the device’s orientation throughout various movements, with no gimbal lock issues encountered. Adding a magnetometer to the system would further improve the reliability of the attitude estimation, but overall the results were quite good.

7 Conclusions

The goal of this project was to estimate and visualize attitude of an IMU device in three dimensional space. Achieving this goal included various steps such as collecting sensor data using a microcontroller, processing and analysing the data, and then finally visualizing the data.

A significant portion of the software and firmware were developed from scratch, serving as a valuable learning experience. Many of the concepts related to sensors, sensor fusion, matrices, quaternion algebra, etc. were also explored in detail. This project was also very useful in learning new tools for data analysis, 3D graphics and software development.

In conclusion, the project was successful in achieving its goal. The project could be further improved by adding magnetometer to the setup and thus improving the reliability of tracking device’s orientation, but even as such, the created system can be used as a foundation for many other IMU and attitude estimation related projects.

References

1. VectorNav Technologies. Inertial Navigation Primer: Math Fundamentals: Reference Frames [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/math-refframes. [Accessed 18 December 2022]

2. VectorNav Technologies. Inertial Navigation Primer: Math Fundamentals: Attitude Representations [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/math-attituderep. [Accessed 18 December 2022]

3. Cprogramming.com. Using Quaternion to Perform 3D Rotations [online]. (n.d.). Available at: https://www.cprogramming.com/tutorial/3d/quaternions.html. [Accessed 14 December 2022]

4. VectorNav Technologies. Inertial Navigation Primer: Theory of Operation: Inertial Sensors [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/theory-of-operation/theory-inertial. [Accessed 19 November 2022]

5. Majid Dadafshar. Accelerometers and gyroscopes sensors: operation, sensing, and applications [online]. (2014). Available at: https://www.analog.com/media/en/technical-documentation/tech-articles/accelerometer-and-gyroscopes-sensors-operation-sensing-and-applications.pdf. [Accessed 20 Decemeber 2022]

6. VectorNav Technologies. Inertial Navigation Primer: Theory of Operation: MEMS Operation [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/theory-of-operation/theory-mems. [Accessed 20 December 2022]

7. Watson J. MEMS Gyroscope Proved Precision Inertial Sensing in Harsh, High Temperature Enviroments [online]. (2016). Available at: https://www.analog.com/media/en/technical-documentation/tech-articles/MEMS-Gyroscope-Provides-Precision-Inertial-Sensing-in-Harsh-High-Temps.pdf. [Accessed 27 December 2022]

8. Bill Earl. Adafruit Triple Axis Gyro Breakout [online]. (2013). Available at: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-triple-axis-gyro-breakout.pdf. [Accessed 6 November 2022]

9. White C. Starting the Tuning Fork Gyroscope, Background [online]. (2020). Available at: https://cwcircuits.com/post/starting-the-tuning-fork-gyroscope-background/. [Accessed 20 December 2022]

10. Dahai Ren; Lingqi Wu; Meizhi Yan; Mingyang Cui; Zheng You; Muzhi Hu. Design and Analyses of a MEMS Based Resonant Magnetometer [online]. Beijing, China: Tsinghua University; (2009). Available at: https://www.mdpi.com/1424-8220/9/9/6951. [Accessed 16 November 2022]

11. Bill Earl. LSM303 Accelerometer + Compass Breakout [online]. (n.d.). Available at: https://cdn-learn.adafruit.com/downloads/pdf/lsm303-accelerometer-slash-compass-breakout.pdf. [Accessed 6 Novemeber 2022]

12. Noordin A; Basri M. A; Mohamed Z. Sensor Fusion Algorithm by Complementary Filter for Attitude Estimation of Quadrotor with Low-cost IMU [online]. Johor, Malaysia: Universiti Teknologi Malaysia; Melaka, Malaysia: Universiti Teknikal Malaysia Melaka; (2016). Available at: http://telkomnika.uad.ac.id/index.php/TELKOMNIKA/article/viewFile/9020/4435. [Accessed 19 Novemeber 2022]

13. VectorNav Technologies. Inertial Navigation Primer: Math Fundamentals: Filtering basics [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/math-fundamentals/math-filtering. [Accessed 19 December 2022]

14. Chaithanya K. M; Frederic Philips Peter Leo; Keshav Deep Verma; Shivaji Kasireddy; Philip M. Scholl; Jochen Kempfle; Kristof Van Laerhoven. Real-Time and Embdedded Detection of Hand Gestures with an IMU-Based Glove [online]. Freiburg im Breisgau, Germany: University of Freiburg; Siegen, Germany: University of Siegen; (2018). Available at: https://www.mdpi.com/2227-9709/5/2/28/htm. [Accessed 20 Novemeber 2022]

15. Townsend K. Adafruit 9-DOF IMU Breakout [online]. (2014). Available at: https://cdn-learn.adafruit.com/downloads/pdf/adafruit-9-dof-imu-breakout.pdf. [Accessed 6 November 2022]

16. Pololu. LSM303DLHC 3D Compass and Accelerometer Carrier with Voltage Regulator [online]. (n.d.). Available at: https://www.pololu.com/product/2124. [Accessed 28 Decemeber 2022]

17. Pololu. L3GD20H 3-Axis Gyro Carrier with Voltage Regulator [online]. (n.d.). Available at: https://www.pololu.com/product/2129. [Accessed 28 Decemeber 2022]

18. Arduino. Wire library documentation [online]. (2021). Available at: https://www.arduino.cc/reference/en/language/functions/communication/wire/. [Accessed 3 January 2023]

19. Adafruit. I2C device address list [online]. (2022). Available at: https://learn.adafruit.com/i2c-addresses/the-list. [Accessed 11 November 2022]

20. Arduino. I2C device scanner code [online]. (2015). Available at: https://playground.arduino.cc/Main/sourceblock_1/index.txt?action=sourceblock&num=1. [Accessed 11 November 2022]

21. STMicroelectronics. LSM303DLHC datasheet [online]. (2013). Available at: https://www.pololu.com/file/0J564/lsm303dlhc.pdf. [Accessed 11 November 2022]

22. STMicroelectronics. L3GD20H datasheet [online]. (2013). Available at: https://www.pololu.com/file/0J731/L3GD20H.pdf. [Accessed 11 November 2022]

23. VectorNav Technologies. Inertial Navigation Primer: Specifications & Error Budgets: Magnetometer Hard & Soft Iron calibration [online]. (n.d.). Available at: https://www.vectornav.com/resources/inertial-navigation-primer/specifications—and--error-budgets/specs-hsicalibration. [Accessed 29 December 2022]

24. Talat Ozyagcilar. Calibrating an eCompass in the Presence of Hard- and Soft-Iron Interference [online]. (2015). Available at: https://www.nxp.com/docs/en/application-note/AN4246.pdf. [Accessed 8 November 2022]

25. Schmidt P. Complimentary Filter Example: Quaternion Based IMU for Accel+Gyro sensor [online]. (2015). Available at: http://philstech.blogspot.com/2015/06/complimentary-filter-example-quaternion.html. [Accessed 1 August 2022]

26. Schmidt P. Fast Quaternion Integration for Attitude Estimation [online]. (2015). Available at: http://philstech.blogspot.com/2014/09/fast-quaternion-integration-for.html. [Accessed 1 August 2022]

27. Wikipedia. Aircraft principal axes [online]. (2023). Available at: https://en.wikipedia.org/wiki/Aircraft_principal_axes. [Accessed 30 January 2023]

28. OpenGL-Tutorial. Tutorial 17: Rotations [online]. (n.d.). Available at: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/. [Accessed 30 January 2023]

Appendix 1

Arduino code

```
#include <Wire.h>
#include "filter.h"

//#define PLOT_MODE     // mode for GNUplot

#ifndef PLOT_MODE
#define INTERVAL 0
#else
#define INTERVAL 100 // time steps in ms for gnuplot
#endif

#define PRINT_ORIENTATION // define one
//#define PRINT_DATA_ACCEL
//#define PRINT_DATA_MAG
//#define PRINT_DATA_GYRO

#define LSM303_ADDRESS_ACCEL 0x19
#define LSM303_REGISTER_CTRL_REG1_A 0x20
#define LSM303_REGISTER_OUT_X_L_A 0x28 | 0x80 // MSB set to 1 to enable auto-
increment

#define LSM303_ADDRESS_MAG 0x1E
#define LSM303_REGISTER_MR_M 0x02
#define LSM303_REGISTER_CRA_M 0x00
#define LSM303_REGISTER_OUT_X_H_M 0x03 // auto-increment enabled by default

#define L3GD20H_ADDRESS 0x69
#define L3GD20H_REGISTER_CTRL_REG1 0x20
#define L3GD20H_REGISTER_OUT_X_L 0x28 | 0x80 // MSB set to 1 to enable auto-
increment

#define LINEAR_ACCELERATION_SENSITIVITY 0.001f // selected scale +/- 2g
#define GYRO_SENSITIVITY 0.00875f // selected scale 245 dps
#define MAGNETIC_GAIN_XY 1100.0f  // selected scale +/- 1.3 gauss
#define MAGNETIC_GAIN_Z 980.0f    // selected scale +/- 1.3 gauss

#define DPS_TO_RADS 0.017453293f // degrees/s to rad/s multiplier
#define RAD_TO_DGS 180 / M_PI        // radians to degrees multiplier

static int xlo, xhi, ylo, yhi, zlo, zhi;
static int raw_x, raw_y, raw_z;
float ax, ay, az;
float gx, gy, gz;
float mx, my, mz;
static float offset_ax, offset_ay, offset_az;
static float offset_gx, offset_gy, offset_gz;

float dt;
static int time;
static unsigned long time_start, time_print,  time_end;
static quat q;                  // quaternion attitude representation
```

```
void setup()
{
            Wire.begin();
            Serial.begin(115200);
            delay(100);

            if (!start_sensor(LSM303_ADDRESS_ACCEL)) {
                        Serial.print("Accelometer failed to start.\n");
                        while (1);
            }

            if (!start_sensor(LSM303_ADDRESS_MAG)) {
                        Serial.print("Magnetometer failed to start.\n");
                        while (1);
            }

            if (!start_sensor(L3GD20H_ADDRESS)) {
                        Serial.print("Gyroscope failed to start.\n");
                        while (1);
            }

            calib();
}

void loop()
{
            time_start = millis();
            read_acc();
            read_mag();
            read_gyro();

            if (time_start - time_print > INTERVAL) {
#ifdef PLOT_MODE
                        Serial.print(time);
                        Serial.print(" ");
                        time+=1;
#endif
#if defined(PRINT_ORIENTATION) && defined(USE_EULERS)
                        print(roll, pitch, yaw);
#elif defined(PRINT_ORIENTATION)
                        print(q.x, q.y, q.z);
#elif defined(PRINT_DATA_ACCEL)
                        print(ax, ay, az);
#elif defined(PRINT_DATA_MAG)
                        print(mx, my, mz);
#elif defined(PRINT_DATA_GYRO)
                        print(gx, gy, gz);
#endif
                        time_print = time_start;
            }

            get_quaternion();
            delay(15); // wait for visualization software
            time_end = millis();
            dt = (float)(time_end - time_start) * 10E-4;
}
```

```
void print(float x, float y, float z)
{
            Serial.print(q.w);
            Serial.print(" ");
            Serial.print(x);
            Serial.print(" ");
            Serial.print(y);
            Serial.print(" ");
            Serial.println(z);
}

bool start_sensor(byte sensor_i2c_address)
{
            byte reg_value;

            switch (sensor_i2c_address) {
            case LSM303_ADDRESS_ACCEL:
                        // set normal mode 100 hz
                        write_byte(sensor_i2c_address,
LSM303_REGISTER_CTRL_REG1_A, 0x57);

                        // check if powered up
                        reg_value        =        read_byte(sensor_i2c_address,
LSM303_REGISTER_CTRL_REG1_A);
                        if (reg_value != 0x57)
                                    return false;

                        break;
            case LSM303_ADDRESS_MAG:
                        // set continous mode
                        write_byte(sensor_i2c_address,    LSM303_REGISTER_MR_M,
0x00);

                        // check if powered up
                        reg_value =
                                    read_byte(sensor_i2c_address,
LSM303_REGISTER_CRA_M);
                        if (reg_value != 0x10)
                                    return false;

                        break;
            case L3GD20H_ADDRESS:
                        // set normal mode 95 hz
                        write_byte(sensor_i2c_address,
L3GD20H_REGISTER_CTRL_REG1, 0x0F);

                        // check if powered up
                        reg_value        =        read_byte(sensor_i2c_address,
L3GD20H_REGISTER_CTRL_REG1);
                        if (reg_value != 0x0F)
                                    return false;
            }
            return true;
}

// read sensor output
void read_sensor(byte sensor_i2c_address, byte output_register)
{
```

```
Wire.beginTransmission(sensor_i2c_address);
Wire.write(output_register);
Wire.endTransmission();

Wire.requestFrom(sensor_i2c_address, (uint8_t)6);

while (Wire.available() < 6);

switch (sensor_i2c_address) {
case LSM303_ADDRESS_ACCEL:
            xlo = Wire.read();
            xhi = Wire.read();
            ylo = Wire.read();
            yhi = Wire.read();
            zlo = Wire.read();
            zhi = Wire.read();
            raw_x = (int16_t)(xlo | (xhi << 8)) >> 4;
            raw_y = (int16_t)(ylo | (yhi << 8)) >> 4;
            raw_z = (int16_t)(zlo | (zhi << 8)) >> 4;

            // data in g's
            ax = (float)raw_x * LINEAR_ACCELERATION_SENSITIVITY;
            ay = (float)raw_y * LINEAR_ACCELERATION_SENSITIVITY;
            az = (float)raw_z * LINEAR_ACCELERATION_SENSITIVITY;

            // zero level bias correction
            ax -= offset_ax;
            ay -= offset_ay;
            az -= offset_az;

            break;
case LSM303_ADDRESS_MAG:
            xhi = Wire.read();
            xlo = Wire.read();
            zhi = Wire.read();
            zlo = Wire.read();
            yhi = Wire.read();
            ylo = Wire.read();
            raw_x = (int16_t)(xlo | (xhi << 8));
            raw_y = (int16_t)(ylo | (yhi << 8));
            raw_z = (int16_t)(zlo | (zhi << 8));

            // data in Gauss
            mx = (float)raw_x / MAGNETIC_GAIN_XY;
            my = (float)raw_y / MAGNETIC_GAIN_XY;
            mz = (float)raw_z / MAGNETIC_GAIN_Z;

            break;
case L3GD20H_ADDRESS:
            xlo = Wire.read();
            xhi = Wire.read();
            ylo = Wire.read();
            yhi = Wire.read();
            zlo = Wire.read();
            zhi = Wire.read();
            raw_x = (int16_t)(xlo | (xhi << 8));
            raw_y = (int16_t)(ylo | (yhi << 8));
            raw_z = (int16_t)(zlo | (zhi << 8));
```

```
                        // angular velocities deg/s
                        gx = (float)raw_x * GYRO_SENSITIVITY;
                        gy = (float)raw_y * GYRO_SENSITIVITY;
                        gz = (float)raw_z * GYRO_SENSITIVITY;

#ifdef USE_EULERS
                        gyro_angle_x += gx * dt;
                        gyro_angle_y += gy * dt;
                        gyro_angle_z += gz * dt;
#else
                        gx *= DPS_TO_RADS;
                        gy *= DPS_TO_RADS;
                        gz *= DPS_TO_RADS;
#endif
                        // zero level bias correction
                        gx -= offset_gx;
                        gy -= offset_gy;
                        gz -= offset_gz;

                        break;
            }
}

void read_acc()
{
            read_sensor(LSM303_ADDRESS_ACCEL, LSM303_REGISTER_OUT_X_L_A);
}

void read_mag()
{
            read_sensor(LSM303_ADDRESS_MAG, LSM303_REGISTER_OUT_X_H_M);
}

void read_gyro()
{
            read_sensor(L3GD20H_ADDRESS, L3GD20H_REGISTER_OUT_X_L);
}

void get_quaternion()
{
            q = update_filter(ax, ay, az, gx, gy, gz, mx, my, mz, dt);
}

// calculate zero-rate levels
void calib()
{
            //=================
            // ACCELEROMETER
            //=================

            int n = 100;
            static float sumx, sumy, sumz;

            for (int i = 1; i <= n; i++) {
                        read_acc();
                        sumx += ax;
                        sumy += ay;
```

```
                sumz += az;
        }

        offset_ax = (float)sumx / n;
        offset_ay = (float)sumy / n;
        offset_az = 1 - ((float)sumz / n);

        //=================
        // GYROSCOPE
        //=================

        sumx = 0;
        sumy = 0;
        sumz = 0;

        for (int i = 1; i <= n; i++) {
                read_gyro();
                sumx += gx;
                sumy += gy;
                sumz += gz;
        }

        offset_gx = (float)sumx / n;
        offset_gy = (float)sumy / n;
        offset_gz = (float)sumz / n;

        //=================
        // MAGNETOMETER
        //=================

        // not implemented
}

void get_hard_iron()
{
        // to be done
}

void get_soft_iron()
{
        // to be done
}

void write_byte(byte sensor_i2c_address, byte reg, byte value)
{
        Wire.beginTransmission(sensor_i2c_address);
        Wire.write(reg);
        Wire.write(value);
        Wire.endTransmission();
}

byte read_byte(byte sensor_i2c_address, byte reg)
{
        byte value;

        Wire.beginTransmission(sensor_i2c_address);
        Wire.write(reg);
        Wire.endTransmission();
```

```
        Wire.requestFrom(sensor_i2c_address, (uint8_t)1);

        while (Wire.available() < 1);

        value = Wire.read();

        return value;
}
```

Appendix 2

Quaternion-based complementary filter code

```
// this code is implementation of filter design by Philip Schmidt
// http://philstech.blogspot.com/2014/09/fast-quaternion-integration-for.html

#include "filter.h"

static vec3 gyro_vec;
static vec3 accel_body, accel_world;
static vec3 mag_body, mag_world;
static vec3 correction_world, correction_world_a, correction_world_m,
correction_body;
static quat gyro_quat, acc_magn_quat;
static quat orientation, orientation_last;

quat update_filter(float ax, float ay, float az, float gx, float gy, float gz,
                           float mx, float my, float mz, float dt)
{
            gyro_vec.x = gx;
            gyro_vec.y = gy;
            gyro_vec.z = gz;

            //accelerometer corrections
            //-------------------------

            accel_body.x = ax;
            accel_body.y = ay;
            accel_body.z = az;

            // rotate accel data from body frame to world frame
            accel_world = rotate(orientation_last, accel_body);

            // pitch & roll corrections from accel
            correction_world_a = cross_product_earth(accel_world);

#if 0
            //magnetometer corrections
            //-----------------------

            mag_body.x = mx;
            mag_body.y = my;
            mag_body.z = mz;

            mag_world = rotate(orientation_last, mag_body);
            mag_world.z = 0.0f;
            normalize(&mag_world);
            correction_world_m = cross_product_north(mag_world);

            // fuse with accelerometer corrections
            correction_world = sum(correction_world_a, correction_world_m);
#else
            correction_world = correction_world_a;
#endif
```

```
            //fuse corrections with gyroscope readings
            //-----------------------------------------

            // rotate correction vector to body frame
            correction_body = rotate(correction_world, orientation_last);

            // add correction vector to gyro
            vec3 gyro_vec_corrected = sum(gyro_vec, correction_body);

            // convert corrected gyroscope data to quaternion and integrate to
            // previous orientation quaternion by multiplication

#if 1       // filtered quaternion
            gyro_quat = gyro_to_quat(&orientation_last, gyro_vec_corrected.x,
                                                    gyro_vec_corrected.y,
gyro_vec_corrected.z, dt);
#else       // unfiltered gyroscope-based quaternion
            gyro_quat = gyro_to_quat(&orientation_last, gyro_vec.x,
                                                    gyro_vec.y, gyro_vec.z, dt);
#endif

            orientation = gyro_quat;
            orientation_last = gyro_quat;

            // normalize the gyro quat to remove any errors
            normalize(&orientation);

            return orientation;
}

vec3 rotate(quat q, vec3 v)
{
            vec3 r, res, res1;
            r.x = q.x;
            r.y = q.y;
            r.z = q.z;

            res1 = cross_product(sum(r, r), sum(cross_product(r, v), mult(q.w,
v)));
            res = sum(v, res1);
            return res;
}

vec3 rotate(vec3 v, quat q)
{
            vec3 r, res, res1;
            r.x = -q.x;
            r.y = -q.y;
            r.z = -q.z;

            res1 = cross_product(sum(r, r), sum(cross_product(r, v), mult(q.w,
v)));
            res = sum(v, res1);
            return res;
}

vec3 cross_product_earth(vec3 v1)
```

```
{
            // VERTICAL = {0.0f, 0.0f, 1.0f}
            vec3 res;
            res.x = v1.y * 1.0f - v1.z * 0.0f;
            res.y = v1.z * 0.0f - v1.x * 1.0f;
            res.z = v1.x * 0.0f - v1.y * 0.0f;
            return res;
}

vec3 cross_product(vec3 v1, vec3 v2)
{
            vec3 res;
            res.x = v1.y * v2.z - v1.z * v2.y;
            res.y = v1.z * v2.x - v1.x * v2.z;
            res.z = v1.x * v2.y - v1.y * v2.x;
            return res;
}

quat mult(quat *a, quat *b)
{
            quat res;
            res.w = a->w * b->w - a->x * b->x - a->y * b->y - a->z * b->z;
            res.x = a->w * b->x + a->z * b->y - a->y * b->z + a->x * b->w;
            res.y = a->w * b->y + a->x * b->z + a->y * b->w - a->z * b->x;
            res.z = a->y * b->x - a->x * b->y + a->z * b->z + a->z * b->w;
            return res;
}

vec3 mult(float a, vec3 v)
{
            vec3 res;
            res.x = v.x * a;
            res.y = v.y * a;
            res.z = v.z * a;
            return res;
}

quat sum(quat q, vec3 v)
{
            quat res;
            res.x = q.x + v.x;
            res.y = q.y + v.y;
            res.z = q.z + v.z;
            res.w = q.w;
            return res;
}

vec3 sum(vec3 v1, vec3 v2)
{
            vec3 res;
            res.x = v1.x + v2.x;
            res.y = v1.y + v2.y;
            res.z = v1.z + v2.z;
            return res;
}
```

```
quat gyro_to_quat(quat *last, float gx, float gy, float gz, float dt)
{
            float t_2 = dt * 0.5f;

            quat res;
            res.w = last->w - t_2 * gx * last->x - t_2 * gy * last->y -
                        t_2 * gz * last->z;
            res.x = last->x + t_2 * gx * last->w - t_2 * gy * last->z +
                        t_2 * gz * last->y;
            res.y = last->y + t_2 * gx * last->z + t_2 * gy * last->w -
                        t_2 * gz * last->x;
            res.z = last->z - t_2 * gx * last->y + t_2 * gy * last->x +
                        t_2 * gz * last->w;
            return res;
}

void normalize(quat *q)
{
            float norm =
                        sqrtf(q->w * q->w + q->x * q->x + q->y * q->y + q->z *
q->z);

            if (norm == 0.0f)
                        return;

            q->w /= norm;
            q->x /= norm;
            q->y /= norm;
            q->z /= norm;
}
```

Appendix 3

Visualizer code

```
#include <stdio.h>
#include <fstream>
#include <iostream>
#include <cstdlib>
#include <math.h>

#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include "serial.h"

#define READ_SERIAL
#define LSM303DHLC

static char msg[256];
static float roll, pitch, yaw;
static float w, x, y, z;
static GLfloat matrix[16];

Serial serial;

static GLFWwindow *window;

void quat_to_matrix(float w, float x, float y, float z)
{
            float x2 = x * x; float y2 = y * y; float z2 = z * z;
            float wx = w * x; float wy = w * y; float wz = w * z;
            float xy = x * y; float xz = x * z; float yz = y * z;

            matrix[0] = 1 - 2 * (y2 + z2); matrix[4] = 2 * (xy - wz);
matrix[8] = 2 * (xz + wy);      matrix[12] = 0;
            matrix[1] = 2 * (xy + wz);      matrix[5] = 1 - 2 * (x2 + z2);
matrix[9] = 2 * (yz - wx);      matrix[13] = 0;
            matrix[2] = 2 * (xz - wy);       matrix[6] = 2 * (yz + wx);
matrix[10] = 1 - 2 * (x2 + y2); matrix[14] = 0;
            matrix[3] = 0;                             matrix[7] = 0;
matrix[11] = 0;                 matrix[15] = 1;
}

void gl_perspective(GLdouble fovY, GLdouble aspect, GLdouble zNear,
                           GLdouble zFar)
{
            GLdouble fW, fH;
            fH = tan(fovY / 360 * M_PI) * zNear;
            fW = fH * aspect;
            glFrustum(-fW, fW, -fH, fH, zNear, zFar);
}

void resize_callback(GLFWwindow *window, int width, int height)
{
            glViewport(0, 0, width, height);
```

```
            glMatrixMode(GL_PROJECTION);
            glLoadIdentity();
            gl_perspective(45, width / height, 0.1, 100.0);
}

void key_callback(GLFWwindow *window, int key, int scancode, int action,
                          int mods)
{
            if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
                        glfwSetWindowShouldClose(window, GLFW_TRUE);
}

bool create_window()
{
            window = glfwCreateWindow(640, 480, "IMU orientation visualizer",
NULL, NULL);
            if (!window) {
                        glfwTerminate();
                        return false;
            }

            glfwMakeContextCurrent(window);
            glfwSetKeyCallback(window, key_callback);
            glfwSetFramebufferSizeCallback(window, resize_callback);
            return true;
}

void draw()
{
            glMatrixMode(GL_MODELVIEW);

#ifndef USE_EULERS
            glPushMatrix();
            glTranslatef(0.0f, 0.0f, -7.0f);
            glMultMatrixf(matrix);
#else
            glLoadIdentity();
            glTranslatef(0.0f, 0.0f, -7.0f);
            glRotatef(roll, 1, 0, 0);
            glRotatef(-pitch, 0, 0, 1);
            glRotatef(yaw, 0, 1, 0);
#endif

            glBegin(GL_QUADS);
            // Top face (y = 1.0f)
            // Define vertices in counter-clockwise (CCW) order with normal
pointing out
            glColor3f(0.0f, 1.0f, 0.0f); // Green
            glVertex3f(1.5f, 0.25f, -1.0f);
            glVertex3f(-1.5f, 0.25f, -1.0f);
            glVertex3f(-1.5f, 0.25f, 1.0f);
            glVertex3f(1.5f, 0.25f, 1.0f);

            // Bottom face (y = -1.0f)
            glColor3f(1.0f, 0.5f, 0.0f); // Orange
            glVertex3f(1.5f, -0.25f, 1.0f);
            glVertex3f(-1.5f, -0.25f, 1.0f);
            glVertex3f(-1.5f, -0.25f, -1.0f);
```

```
            glVertex3f(1.5f, -0.25f, -1.0f);

            // Front face  (z = 1.0f)
            glColor3f(1.0f, 0.0f, 0.0f); // Red
            glVertex3f(1.5f, 0.25f, 1.0f);
            glVertex3f(-1.5f, 0.25f, 1.0f);
            glVertex3f(-1.5f, -0.25f, 1.0f);
            glVertex3f(1.5f, -0.25f, 1.0f);

            // Back face (z = -1.0f)
            glColor3f(1.0f, 1.0f, 0.0f); // Yellow
            glVertex3f(1.5f, -0.25f, -1.0f);
            glVertex3f(-1.5f, -0.25f, -1.0f);
            glVertex3f(-1.5f, 0.25f, -1.0f);
            glVertex3f(1.5f, 0.25f, -1.0f);

            // Left face (x = -1.0f)
            glColor3f(0.0f, 0.0f, 1.0f); // Blue
            glVertex3f(-1.5f, 0.25f, 1.0f);
            glVertex3f(-1.5f, 0.25f, -1.0f);
            glVertex3f(-1.5f, -0.25f, -1.0f);
            glVertex3f(-1.5f, -0.25f, 1.0f);

            // Right face (x = 1.0f)
            glColor3f(1.0f, 0.0f, 1.0f); // Magenta
            glVertex3f(1.5f, 0.25f, -1.0f);
            glVertex3f(1.5f, 0.25f, 1.0f);
            glVertex3f(1.5f, -0.25f, 1.0f);
            glVertex3f(1.5f, -0.25f, -1.0f);
            glEnd();

#ifndef USE_EULERS
            glPopMatrix();
#endif
}

void get_orientation()
{
            if (!serial.readLine(msg, sizeof(msg) - 1))
                        return;

            char *start = msg;
            char *ptr;
            char *end = msg + sizeof(msg);

            if (!(ptr = strchr(start, ' '))) {
                        return;
            } else {
                        *ptr = '\0';
                        w = strtof(start, NULL);
            }

            start = ptr + 1;
            if (!(ptr = strchr(start, ' '))) {
                        return;
            } else {
                        *ptr = '\0';
                        x = strtof(start, NULL);
```

```
		}

		start = ptr + 1;
		if (!(ptr = strchr(start, ' '))) {
			return;
		} else {
			*ptr = '\0';
			y = strtof(start, NULL);
		}

		z = strtof(ptr + 1, NULL);
		printf("w: %f x: %f y: %f z: %f\n", w, x, y, z);

		quat_to_matrix(w, x, z, -y);
}

bool init()
{
		if (!glfwInit()) {
			printf("Window initialization failed\n");
			return false;
		}

		if (!create_window()) {
			printf("Window creation failed\n");
			return false;
		}

		if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress)) {
			printf("Failed to load GLAD\n");
			return false;
		}

		resize_callback(window, 640, 480);
		glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
		glClearDepth(1.0f);
		glEnable(GL_DEPTH_TEST);
		glDepthFunc(GL_LEQUAL);
		glShadeModel(GL_SMOOTH);
		glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

#ifdef READ_SERIAL
		serial.openSerial();
#endif
		return true;
}

int main()
{
		if (!init())
			return -1;

		while (!glfwWindowShouldClose(window)) {
			get_orientation();
			glfwPollEvents();
			glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
			draw();
```

```
                glfwSwapBuffers(window);
        }

#ifdef READ_SERIAL
        serial.closeSerial();
#endif
        glfwDestroyWindow(window);
        glfwTerminate();
        return 0;
}
```