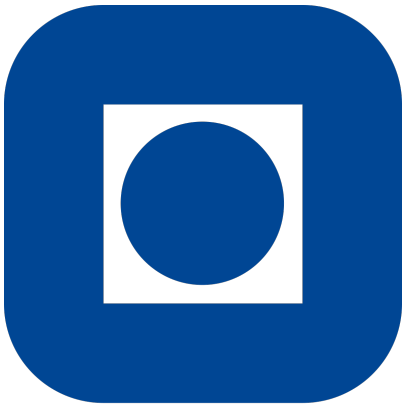
ADCS Sensor Suite Calibration and Signal Processing

Orienteringsbestemmelse: Sensorkalibrering og Signalbehandling

Edvard Birkeland
Eivind Bjørnebøle
Erlend Hestvik

A thesis presented for the degree of
Bachelor of Electrical Engineering



NTNU

Department of Electronic Systems
NTNU Trondheim
Norway
Spring 2020

ADCS Sensor Suite Calibration and Signal Processing

Orienteringsbestemmelse: Sensorkalibrering og Signalbehandling

Edvard Birkeland
Eivind Bjørnebøle
Erlend Hestvik

Abstract

Controlling a spacecraft requires highly precise and calculated action, to know how to move a satellite you must first know where the satellite is, and it's current attitude. Attitude determination with three degrees of freedom requires multiple specialized sensors working together, both to eliminate individual sensor errors, and to strengthen the confidence in our results. Using a microcontroller unit and an IMU with a gyroscope, accelerometer and magnetometer we can implement advanced fusion algorithms such as Extended Kalman Filters or Madgwick filter to calculate the current attitude of a satellite with the level of precision necessary for orbital maneuvers. The microcontroller software itself must be written to efficiently gather all the sensory data, process the signals with the fusion algorithm, and pass the data on to the actuators. All this while running at a sufficient speed to ensure the data is as correct as it can be.

Preface

This bachelor thesis is written by three students from the electrical engineering program at NTNU Trondheim in the spring of 2020. The project involved research and development of an attitude determination system for Orbit NTNU's current mission, the SelfieSat.

When deciding on a subject for our bachelor thesis we, as a group, had many possibilities and options to choose. We selected this thesis, ADCS Sensor suite calibration and signal processing, because it sounds like an interesting and challenging subject matter. Orbital mechanics and satellites are not everyday subjects when studying for electrical engineering, and so we saw this as a great learning opportunity in a field which we have little experience in, but interests us greatly. We also see the bachelor thesis as a learning experience regarding long unsupervised projects in which we will have to determine and decide on our own what the best course of action is.

The thesis, explaining our workflow, and the implemented C-code attached as an appendix, will be our final product delivered to Orbit NTNU.

This is a bachelor thesis written by students at the third and final year of electrical engineering at NTNU. Hence it is written in a fashion that a fellow student would be able to follow along.

If you have any questions or inputs regarding the project, report or code, feel free to contact us.

We would like to thank Orbit NTNU for providing the project, and Håkon Grønning for supervising us. Special thanks to Andreas Westre and Naveed Ahmed at Orbit NTNU.

Contents

List of Figures

Glossary

ADCS Attitude Determination and Control Systems

IMU Inertial Measurement Unit

LSM9DS1 An IMU module

Arduino uno A microcontroller board

ATmega328p The microcontroller chip embedded in an Arduino uno

MEMS Micro-Electro-Mechanical Systems

SPI Serial Peripheral Interface

USART Universal Synchronous/Asynchronous Receiver/Transmitter

Attitude Orientation in space

Quaternions A four-dimensional orientation representation

Euler angles An intuitive representation of orientation

ODR Output Data Rate

DoF Degrees of Freedom

Chapter 1

Introduction

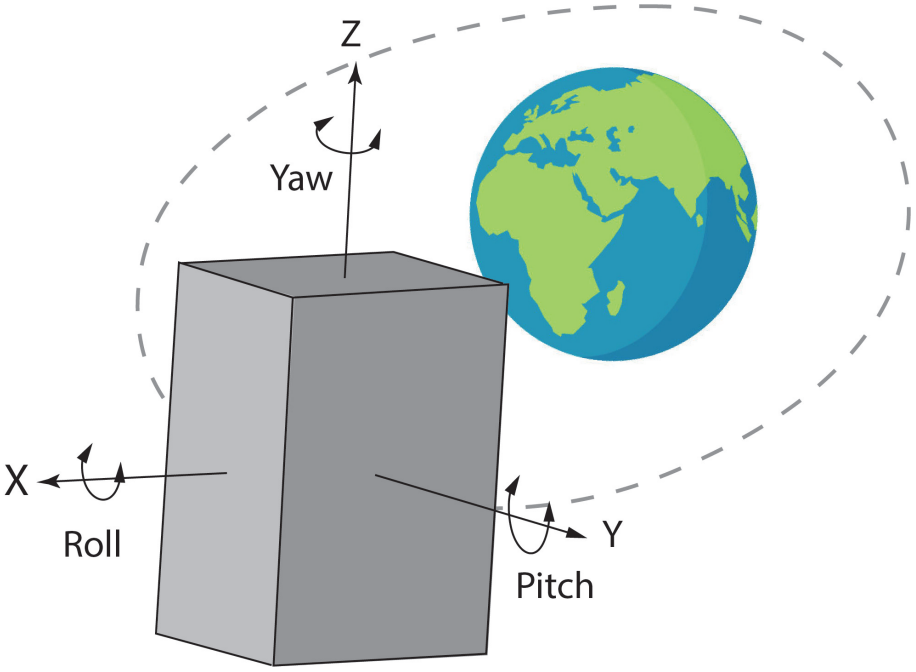


Figure 1.1: Illustration of a satellite with its axes orbiting earth.

1.1 Background

Humans has always longed to see what is beyond their horizon. In the late 50's we moved that horizon to space, to explore thet unknown. The following decades the spacecraft industry has taken huge technologically steps, as the never ending horizon moves further and further away. From interstellar missions into the outer space, to micro satellites in low earth orbit, the common goal is to explore and increase our collective knowledge. A sailor uses a compass to determine his ships orientation in the ocean, the same way a spacecraft engineer uses an IMU to determine the spacecrafts orientation in space see [fig 1.1].
Orbit NTNU is a non-profit student organization that are currently designing and building a small cube satellite, with the goal of launching their "SelfieSat" into space. Their motivation is to create the next generation of space engineers, by working on complex space projects [1]. The SelfieSat is equipped with a camera pointing at a mounted display, this will let people upload a picture to the screen, and then take a

selfie with Earth in the background. For this to work it needs to maintain a constant direction facing the planet, or at the very least maintain a set angle for the duration the earth is visible. To achieve this the SelfieSat needs to know its orientation in relation to earth.
The field involved with this work is called "attitude determination" and it's a part of a broader field, Attitude Determination and Control Systems, see [fig 1.2] for a full flow chart of the field. With on-board actuators on the Selfiesat, the team can control its attitude. But the actuators needs to know the current attitude of the satellite in order to know how much thrust to give. Our job is to find this attitude as precisely as we can.

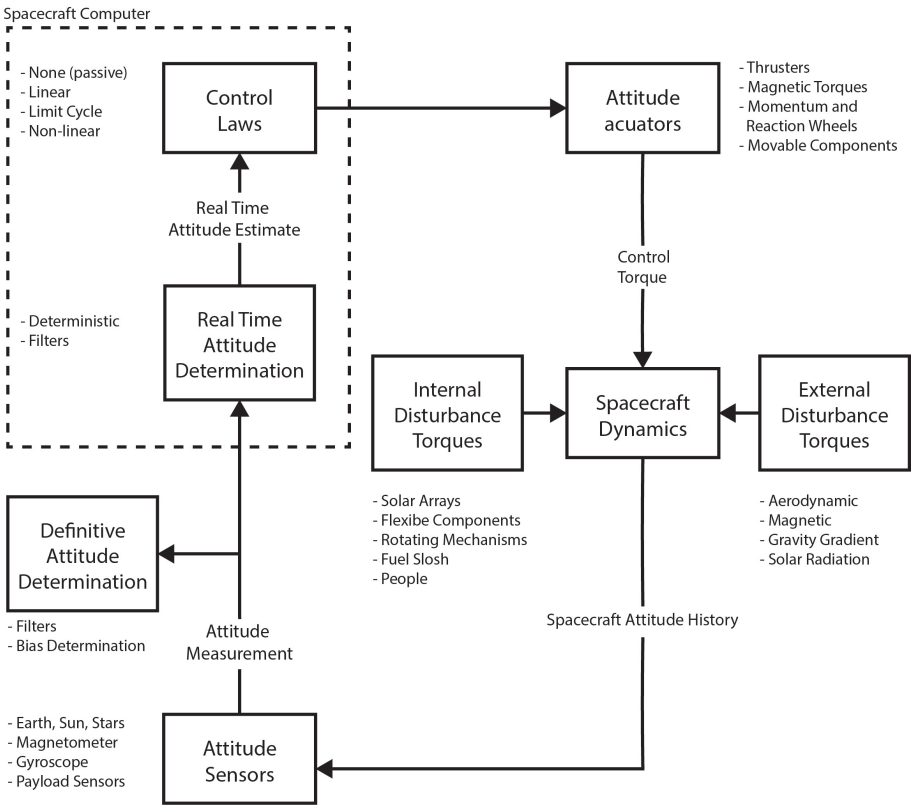


Figure 1.2: ADCS diagram taken from a NASA document [2], reproduced by us for improved resolution.

1.2 Problem definition

In order for the Selfiesat to maintain the same spacial orientation it needs to know it's current attitude. An Inertial Measurement Unit, IMU for short, will in conjunction with another sensor provide the satellite with the data necessary for attitude determination. The IMU has no processing power on it's own and will need a microcontroller capable of high speed short distance hardware communication and mathematical operations in order to successfully output the satellite's orientation.

1.3 Our tasks

The assignment is centered specifically around the IMU itself, and no other hardware component included in the ADCS umbrella. The module of choice is the LSM9DS1, and we will be interfacing it with an Arduino UNO. The first order of operation is learning about and gaining a good understanding of IMUs in general and then specifically how the LSM9DS1 works, then figure out how the IMU is used within the context of ADCS. This will involve finding and studying relevant literature and other media we find that are applicable.
Next we must learn to interface the LSM9DS1 with our Arduino UNO using a Master-slave SPI setup. This will let us conduct the high speed hardware communication required to gather all the necessary data in a timely fashion.
After successfully interfacing the LSM9DS1 module with the Arduino we must learn to calibrate the sensory systems on board the module, primarily the gyroscope and the magnetometer, but the accelerometer should also be considered if possible. Calibration will likely require the use of conventional algorithms for magnetometer, gyroscope and accelerometers, these will have to be researched and implemented by us.
Determining the attitude will require the implementation of some method to fuse the sensory data together, for example a Kalman filter or Madgwick filter.
Signal processing is also a part of the assignment, the sensor suite will experience noise and drift which has to be accounted for with filters. We may also have to look at signal conditioning in case the LSM9DS1 module outputs a signal which is unfit for interfacing with the Arduino. The module may also experience Allan Variance that will have to be researched. Finding the noise characteristics of the IMU will be an important part of designing the digital filters necessary for the ADCS system to work.

1.4 Our plan

Our plan is broken down to multiple stages, starting with research and familiarization, then moving on to calibration, system verification, signal processing and finally the writing of this very paper. See [fig 1.3]

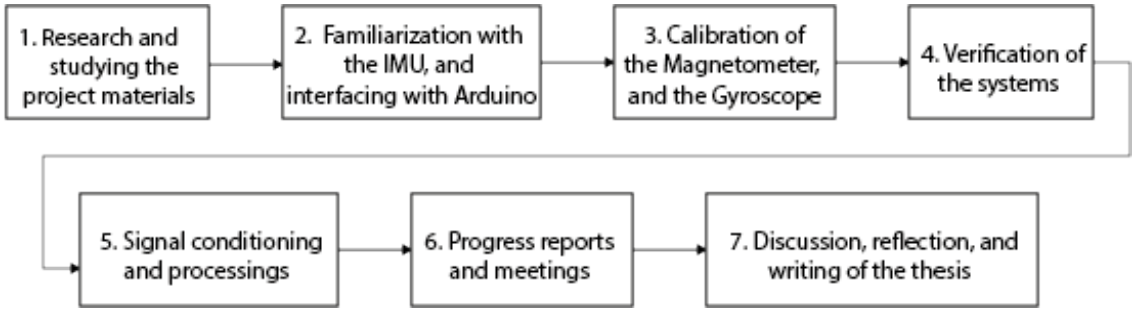


Figure 1.3: Diagram of the work packages.

1.5 Unfinished business

This segment was written at the end of the project period.
Some of our plans didn't end up leading to useful discoveries or productive code. Things that did not go after the plan includes: Allan Variance, temperature dependency, fault detection, in orbit calibrations, in system TWOSTEP algorithms for magnetometer calibration, noise analysis, frequency filter design. We also ended up not implementing any Kalman filter, instead we went with a Madgwick filter.
Therefore, these topics will not be covered much in the main chapters despite being mentioned in the introduction. The discussion chapter will bring up some of these points and why we ended up not following through with out plans.

1.6 The Thesis structure

The thesis is divided into an introduction, five main chapters, and finally a conclusion, each chapter is further divided into sections and subsections. The first chapter is the theory, here we attempt to explain all the pre-requisite information you would need to understand the rest of the chapters, if you are already well versed in the world of ADCS you will most likely not need to read the theory chapter. Next up are two chapters related to the process we underwent during this project, these are the Interfacing and Signal Processing chapters. In these chapters we explain the whole process we went through, from start to finish, in a language that should be easy to follow even if you are not familiar with the subject. Interfacing and Signal Processing contain the meat of the work we have done over the course of this project. The next two chapters are closely related, Results and Discussion. Results is a brief showcasing of the success we had with calibration, our experiments with the fusion filter feedback gain, and the accuracy of our attitude determination. Discussion is an overview of the project from our perspective after it's completion, then a reflection on and discussion of our results. And lastly a write up of future work or planned work packages we ended up not following through with. The closing word is our conclusion, here we make a brutally honest summary of what we did and did not accomplish during the project period. The conclusion offer no excuses or defense for our shortcomings, and no celebration or bragging of our success, it is simply a summary to wrap everything up.

Chapter 2

Theory

This chapter will lay the groundwork for understanding all the hardware, software and terminology that will be used later in the thesis. Here we are concerned with how and why things work, and less concerned with the implementation, that will be covered in later chapters. The goal of this chapter is to explain the fundamentals in a way that makes the main chapters easier to digest, and hopefully this chapter combined with the main chapters should contain all the information required to fully understand the thesis.

2.1 Interfacing

For any hardware unit to communicate with the outside world a means of communicating must first be established. The ubiquitous term for this membrane is "interface", and it can be thought of as a sort of language. For two units to communicate they must speak the same language, some components can speak many languages and some only speak one. The languages are very different from each other and all have their pros, cons, and specialized use cases. There are two important interfaces we will be using for this project. SPI for communication between the Arduino UNO and the IMU, and USART for communication between the Arduino UNO and the computer.

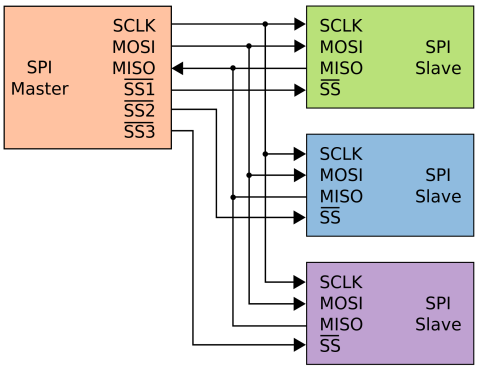


Figure 2.1: Example of a typical SPI setup with 1 master and 3 slaves.

2.1.1 SPI

Serial Peripheral Interface (SPI), is a bi-directional communication interface well suited for short distance communication between two or more hardware components. The core idea is to set up a master-slave relationship between the units; there can be many slaves, but only one master, see [fig 2.1]. The master unit is responsible for setting the clock speed and for initiating communication. There are two ways to connect the units when setting up SPI, 4-wire mode is the standard and what we will be using, but it's also possible or sometimes necessary to use a 3-wire setup. In 4-wire setups the wires are as following: Serial Clock (SCK), Slave Select / Chip Select (SS/CS), Master Out Slave In (MOSI), Master in Slave Out (MISO). One common mistake when hooking up units for SPI is connecting MOSI to MISO and vice versa, mistakenly thinking the slave would consider itself a master. When you move data with SPI the transaction happens both ways simultaneously, this can be imagined as a shift register shifting one bit at a time between the master and the slave register. See [fig 2.2] [3]

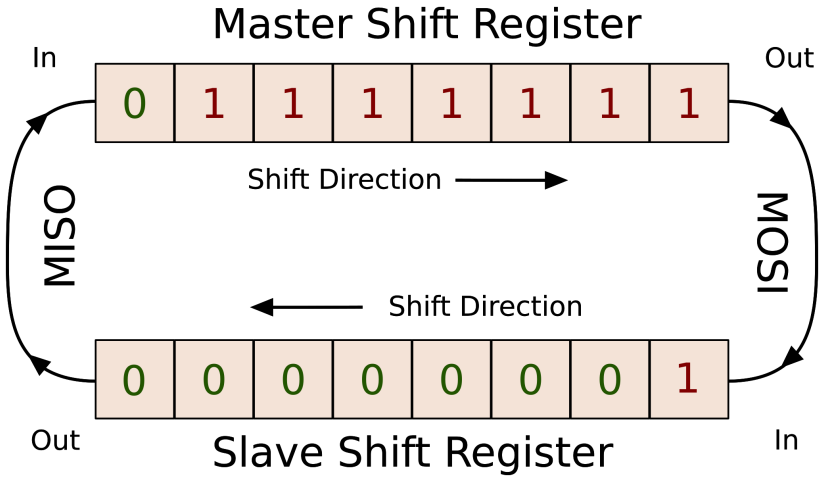


Figure 2.2: SPI shifts data one bit at a time and receives data the same way.

2.1.2 USART

Universal Synchronous/Asynchronous Receiver/Transmitter (USART) is another bi-directional serial communication interface. Serial means that the bits are transferred one by one over a single line, and synchronous / asynchronous denotes whether or not a clock signal is present to sync the signal. In an asynchronous setup there is no common clock signal between the USART units, and so they must both be configured with a pre-determined baud rate, which is how many bits per second the system will send/recieve. To initiate communication a start bit is sent, and to end communication one or two stop bits are used, see [fig 2.3].

2.2 IMU

An Inertial Measurement unit, IMU for short, is an electronic sensory device mainly used for monitoring and detecting spatial orientation, the attitude, of an object.

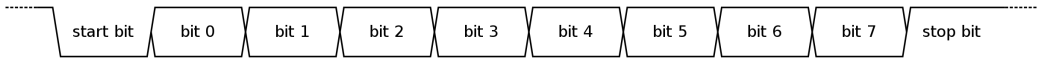


Figure 2.3: USART timing diagram with one start and one top bit. [4]

IMUs usually have an accelerometer and a gyroscope, and sometimes they also include a magnetometer. The gyroscope and accelerometer are enough to monitor full six degrees of freedom (DoF) movement of any object, but by themselves they struggle with sensor drift. The magnetometer allows us to align the yaw of the object with magnetic north to find the heading, the magnetometer also combats the sensor drift of the accelerometer and gyroscope because the earths magnetic field is a comparatively stable field and thus does not experience much drift of its own.
While IMUs can be used to track an object's translation in 3DoF via a method known as dead reckoning, this is not something we have looked at for this project, we are mainly concerned with the rotational movement; Pitch, Roll, and Heading.

2.2.1 Accelerometer

An accelerometer measures linear acceleration, the unit measured is denoted as g. With 1g being the standard gravity, the acceleration of an object free falling in a vacuum near earths surface. Typically, in a small form-factor IMU, the accelerometer will be a MEMS sensor. This can be thought of as a tiny metal plate with a certain mass, called proof mass, suspended by springs close by a fixed electrode. When the metallic mass moves the proof mass will move closer or further away from the electrode, and the resulting capacitance can be used to deduce the applied force. [5]

2.2.2 Gyroscope

A gyroscope measures angular rate, the speed at which an object rotates around it's own axis, the unit measured is simply degrees per second. MEMS gyroscopes are structurally quite similar to MEMS accelerometers, a proof mass is suspended by springs and moves when rotational force is applied, the displacement of the mass is detected with probing electrodes or other similar methods, and rotational force is deduced. See [fig 2.4]. [2]

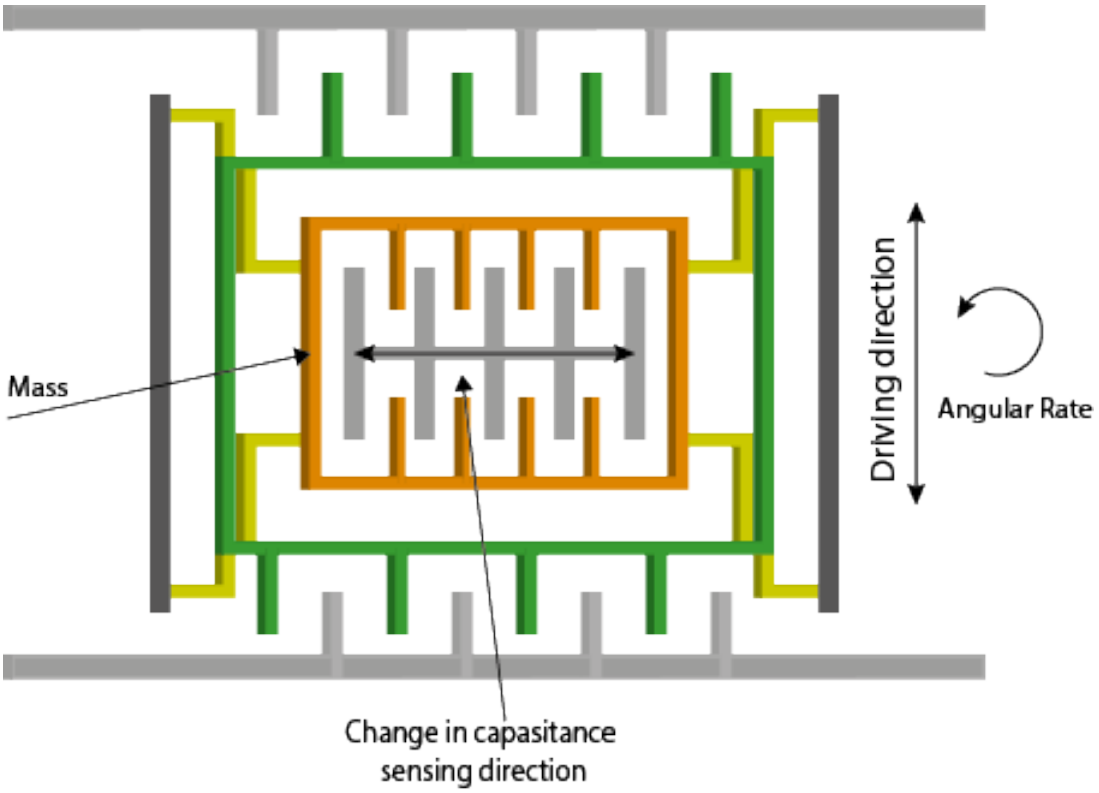


Figure 2.4: Illustration of a MEMS Gyroscope [6]. This figure is remade by us for better clarity.

2.2.3 Magnetometer

A magnetometer measures magnetic field strength in the unit tesla or gauss, where one tesla is equal to 10^4 gauss. A MEMS magnetometer can function in a couple of ways, where the most used method utilize the Hall effect.

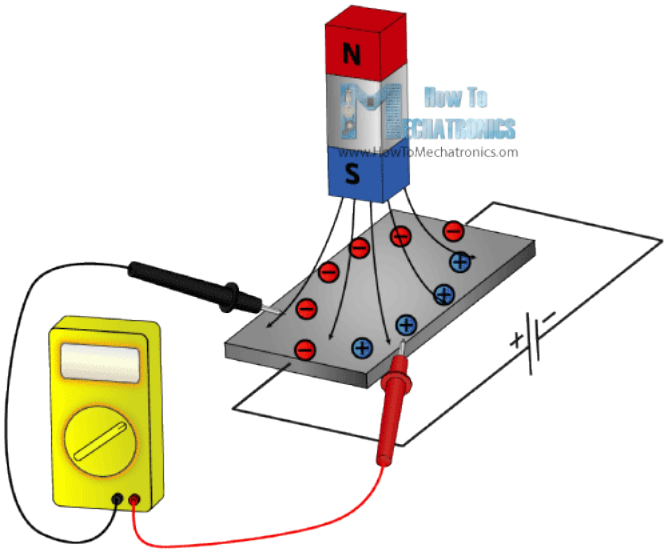


Figure 2.5: Illustration of how the Hall effect works, and how it helps us measure magnetic fields. [6]

Which is when magnetic fields create a voltage potential across a conductive plate perpendicular to the magnetic field's direction. See [fig 2.5]. [6]

2.2.4 Sensor characteristics: Sensitivity vs Range

Sensors are not perfect, no sensor can measure an infinite range of values with infinite sensitivity. Sensitivity refers to how much change is measured per change in bit value on the sensor outputs. Let's use the LSM9DS1's gyroscope as an example: a sensitivity of 8.75 mdps/LSB (milli-degrees per seconds / Least Significant Bit) means that a bit value of one on the output equals a rotation of 0.00875 degrees per second, and a bit value of two equals 0.01750 degrees per second. The gyroscope outputs it's measurement to a 16-bit register, therefore we can see that the maximum possible dps the gyroscope can measure at this sensitivity is $0.00875 * 2^{16} = 573.44dps$, but since we need to be able to measure rotation in both positive and negative direction we only get about half of that, 286 dps, for each direction. However if we bumped the sensitivity down to 0.07 mdps/LSB we could measure a maximum angular rate of
$0.07 * 2^{16} = 4587.52dps$, or about ±2250 dps since we need both directions. One important caveat is that we cannot set the sensitivity to any value we want however, if we look at the data sheet for the LSM9DS1 we can see that the gyroscope has 3 sensitivity settings based on the range you want: $\pm 245dps, \pm 500dps, \pm 2000dps$ [Fig 2.6]

G_So	Angular rate sensitivity	Angular rate FS = ±245 dps		8.75		mdps/ LSB
		Angular rate FS = ±500 dps		17.50		
		Angular rate FS = ±2000 dps		70		

Figure 2.6: The three sensitivity settings for the Gyroscope we're using.[7]

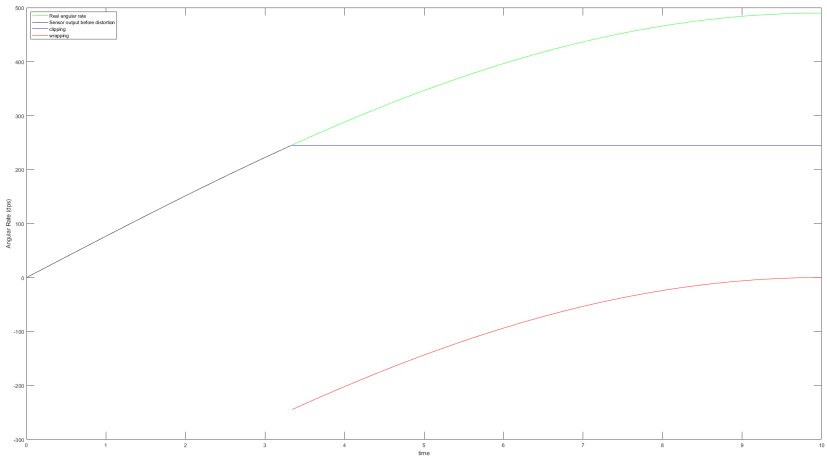


Figure 2.7: A graph showing the effects of clipping and wrapping

FS stands for Full Scale, so if we want the Full Scale of the Gyro to be ±245 dps the sensitivity is set to 8.75 mdps/LSB. If the actual speed of the angular rate exceeds the full scale it's very likely to cause what is called clipping, which is a

distortion caused by the truncation of the output signal due to physical limitations. E.g. if the angular rate over a ten second period rises from 0 dps to 490 dps, the output signal would match the curve until it reached 245 dps and then flatten out until the real angular rate came back down to < 245. The alternative to clipping is called wrapping and is even less desirable, in this scenario the output waveform would wrap around to -245 and continue to climb up to 0 dps. See [fig 2.7].

2.2.5 Control registers

When talking about configuring integrated circuits for SPI and other functionalities, we're really talking about setting up the control registers on a chip. Many integrated circuits are capable of multiple functionalities or different ways of doing the same thing, and the way you set this up is by configuring the control registers by setting bit values to either 1 or 0 depending on what you want. As an example, for the Arduino UNO to "know" that it's even supposed to be using SPI communication we need to at the very least enable SPI and set it to master mode in the SPCR register by writing a 1 to the 6th and 4th bit, see [fig 2.8].

SPCR – SPI Control Register

Bit	7	6	5	4	3	2	1	0	
0x2C (0x4C)	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0	SPCR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2.8: SPCR - SPI control register. [4]

2.3 Signal processing

Signal processing is the process of analyzing and modifying a signal to better fit some criteria of choice. It can for example be to reduce unwanted noise or amplify/attenuate parts of the signal.

2.3.1 Calibration

In an ideal world every sensor would work perfectly straight out of the package. In the real world sensors are full of errors and flaws, some are minor and easily accounted for, others can be very problematic. Calibration is the first step in fighting these flaws, but it's also an important tool for optimizing the sensor for the work you want it to pull off. Calibration can be things like setting the sensitivity of the sensor, measuring and accounting for static offsets or other measurement faults like bias.

Hard and Soft iron errors for magnetometer

The raw magnetometer data is not usable as it is, and need to be calibrated before sensor fusing. When rotating an ideal magnetometer covering all angles and plotting the result, it would look like a perfect sphere with center in origin and radius equal

to the magnetic field strength.
The two main sources to errors in the magnetometer readings are called hard and soft iron effect.
Hard iron effect is caused by a nearby object that generates a magnetic field of its own, this can be a natural or electrical magnet. This field is added to the earths magnetic field and creates an offset error on the measured data. When plotted, it will make the origin of the sphere shifted. This effect is the most impactful error source for a magnetometer.
Soft iron effect occurs when a nearby ferromagnetic material distorts the magnetic field that is already present. This type of error will stretch the ideal sphere to look like an ellipsoid when plotted. See [fig 2.9] and [fig 2.10] [8]

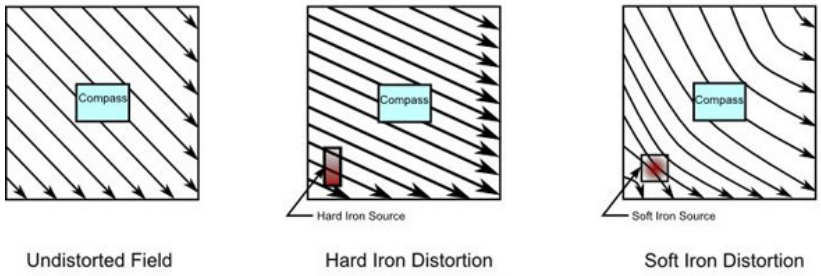


Figure 2.9: Illustrations of the hard iron and soft iron sources.[9]

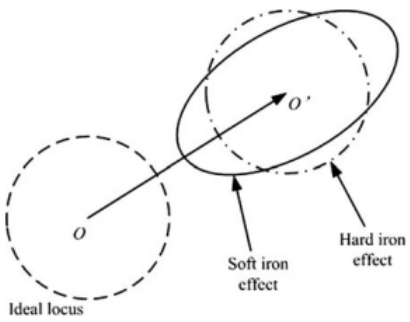


Figure 2.10: 2D illustration of the hard and soft iron effects.[10]

2.3.2 Determining attitude

This segment will assume all sensors are calibrated properly and the data has been scaled to account for the set sensitivity. There are multiple methods one can employ, the simplest method is simply integrating the gyroscope data in a loop:

$$pitch\_gyro = pitch\_gyro + gyro\_x$$

$$roll\_gyro = roll\_gyro + gyro\_y$$

$$yaw\_gyro = yaw\_gyro + gyro\_z$$

One problem here is the complete disconnect between the three axes, for example if the pitch is at 45° and the device yaw is changed by 90° the pitch should become

0° , and the roll should increase to 45°. One way to fix this is to add two lines of codes that changes the pitch and roll if a change on the gyro z axis is made:

$$pitch\_gyro = pitch\_gyro + roll\_gyro \cdot sin(gyro\_z \cdot pi/180)$$

$$roll\_gyro = roll\_gyro + pitch\_gyro \cdot sin(gyro\_z \cdot pi/180)$$

Similarly you'd have to account for how combinations of pitch and roll would affect the yaw. The gyroscope by itself drifts very fast, implementing the accelerometer data into this method would keep the roll, pitch and yaw accurate for a longer period of time. To calculate the roll, pitch and yaw using the accelerometer you first need the total length vector of the accelerometer axes:

$$acc\_length = \sqrt{(acc\_x^2) + (acc\_y^2) + (acc\_z^2)}$$

Pitch can then be found with the formula:

$$Pitch\_acc = asin(acc\_y/acc\_length) \cdot 180/pi$$

Combining the accelerometer and gyroscope values is then done by simply taking a large portion of the gyroscope measured pitch, and a much smaller portion of the accelerometer measured pitch:

$$pitch = pitch\_gyro \cdot 0.996 + pitch\_acc \cdot 0.004$$

Similar operations for roll and yaw, this is not a very elegant solution, and we do not recommend using this for anything beyond measuring if the device is level with the ground.[11]
Now that's a lot of effort spent explaining a method which we do not recommend using, but hopefully it should help with gaining an intuition for how attitude determination works in principle. The maths for the more advanced methods is a lot more complicated.

2.3.3 Madgwick filter

The Madgwick filter is an adaptive filter that uses the sensor measurements to estimate the orientation of the unit. The filter uses a quaternion representation of the attitude, which allows it to use the measurements in a computational effective manner. The filter compensates for gyroscope drift as well as some magnetic distortion. The filter estimates the orientation of the sensor frame relative to the earth frame, ${}^{S}_{E}\mathbf{q}_{est,t}$, by combining two individual orientation estimations, one estimation calculated directly from the gyroscope data ${}^{S}_{E}\mathbf{q}_{\omega,t}$ and one from the accelerometer and magnetometer data ${}^{S}_{E}\mathbf{q}_{\nabla,t}$. $\mathbf{q}$ is a 1-by-4 vector consisting of the quaternions that represents the orientation. The notations ^{E} and ^{S} indicates that the vector describes the earth or sensor frame, while ${}^{S}_{E}$ indicates sensor frame orientation in relation to earth frame.

Angular rate orientation

The orientation estimated from the gyroscope measurement, ${}^{S}_{E}\mathbf{q}_{\omega,t}$, is calculated by integrating the angular rate over time. The angular rates about the x, y, and z axis

are arranged in the vector ${}^{S}\boldsymbol{\omega}$. To be able to work with quaternions, a zero is added into the first position of the vector.

$$ {}^{S}\boldsymbol{\omega} = \begin{bmatrix} 0 & \omega_x & \omega_y & \omega_z \end{bmatrix} $$

By numerical integration of the previous estimation of the attitude, the next one can be calculated. Here $\otimes$ denotes the quaternion product.

$$ {}^{S}_{E}\dot{\mathbf{q}}_{\omega,t} = \frac{1}{2} {}^{S}_{E}\hat{\mathbf{q}}_{\omega,t-1} \otimes {}^{S}\boldsymbol{\omega} $$

$$ {}^{S}_{E}\mathbf{q}_{\omega,t} = {}^{S}_{E}\hat{\mathbf{q}}_{\omega,t-1} + {}^{S}_{E}\dot{\mathbf{q}}_{\omega,t}\Delta t $$

Vector observations orientation

Since the filter operates with quaternions, the accelerometer and magnetometer data can be used in a gradient descent algorithm to estimate the orientation of sensor frame ${}^{S}_{E}\mathbf{q}_{\nabla,t}$.

$$ \mathbf{f}({}^{S}_{E}\hat{\mathbf{q}}, {}^{E}\hat{\mathbf{d}}, {}^{S}\hat{\mathbf{s}}) = {}^{S}_{E}\hat{\mathbf{q}}^{*} \otimes {}^{E}\hat{\mathbf{d}} \otimes {}^{S}_{E}\hat{\mathbf{q}} - {}^{S}\hat{\mathbf{s}} $$

$$ {}^{S}_{E}\hat{\mathbf{q}} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} $$

$$ {}^{E}\hat{\mathbf{d}} = \begin{bmatrix} 0 & d_x & d_y & d_z \end{bmatrix} $$

$$ {}^{S}\hat{\mathbf{s}} = \begin{bmatrix} 0 & s_x & s_y & s_z \end{bmatrix} $$

The next estimation of the attitude can be found by the gradient decent formula, where μ is the step length, and $\mathbf{J}$ is the Jacobian matrix:

$$ {}^{S}_{E}\mathbf{q}_{k+1} = {}^{S}_{E}\hat{\mathbf{q}}_{k} - \mu \frac{\nabla \mathbf{f}({}^{S}_{E}\hat{\mathbf{q}}_{k}, {}^{E}\hat{\mathbf{d}}, {}^{S}\hat{\mathbf{s}})}{\|\nabla \mathbf{f}({}^{S}_{E}\hat{\mathbf{q}}_{k}, {}^{E}\hat{\mathbf{d}}, {}^{S}\hat{\mathbf{s}})\|}, \quad k = 0, 1, 2...n $$

$$ \nabla \mathbf{f}({}^{S}_{E}\hat{\mathbf{q}}_{k}, {}^{E}\hat{\mathbf{d}}, {}^{S}\hat{\mathbf{s}}) = \mathbf{J}^{T}({}^{S}_{E}\hat{\mathbf{q}}_{k}, {}^{E}\hat{\mathbf{d}}) \mathbf{f}({}^{S}_{E}\hat{\mathbf{q}}_{k}, {}^{E}\hat{\mathbf{d}}, {}^{S}\hat{\mathbf{s}}) $$

When using the accelerometer data, we can assume that the sensor only measures gravity, which means that if the earth frame and the sensor frame are aligned, $1g$ will be measured in the z direction and nothing in the other directions. We can substitute ${}^{E}\hat{\mathbf{d}}$ and ${}^{S}\hat{\mathbf{s}}$ with ${}^{E}\hat{\mathbf{g}}$ and ${}^{S}\hat{\mathbf{a}}$ respectively in the equation above to estimate the orientation.

$$ {}^{E}\hat{\mathbf{g}} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix} $$

$$ {}^{S}\hat{\mathbf{a}} = \begin{bmatrix} 0 & a_x & a_y & a_z \end{bmatrix} $$

The magnetometer data can be used in the same manner. We can assume that the sensor only measures earth's magnetic field. This assumption means that when the frames are aligned, the magnetic field strength will measure in the x and z axis, while nothing is measured in the y axis. The z direction is included because of earth's magnetic field inclination. Substituting ${}^{E}\hat{\mathbf{d}}$ and ${}^{S}\hat{\mathbf{s}}$ with ${}^{E}\hat{\mathbf{b}}$ and ${}^{S}\hat{\mathbf{m}}$ enables us to find the orientation of the sensor suite.

$$ {}^{E}\hat{\mathbf{b}} = \begin{bmatrix} 0 & b_x & 0 & b_Z \end{bmatrix} $$

$$ {}^{S}\hat{\mathbf{m}} = \begin{bmatrix} 0 & m_x & m_y & m_z \end{bmatrix} $$

To get an unique orientation of the sensor frame, that includes heading, the accelerometer and magnetometer measurements can be combined and computed in the gradient decent function.

$$\mathbf{f}_{g,b}({}^{S}_{E}\hat{\mathbf{q}}, {}^{S}\hat{\mathbf{a}}, {}^{E}\hat{\mathbf{b}}, {}^{S}\hat{\mathbf{m}}) = \begin{bmatrix} \mathbf{f}_g({}^{S}_{E}\hat{\mathbf{q}}, {}^{S}\hat{\mathbf{a}}) \\ \mathbf{f}_b({}^{S}_{E}\hat{\mathbf{q}}, {}^{E}\hat{\mathbf{b}}, {}^{S}\hat{\mathbf{m}}) \end{bmatrix}$$

$$\mathbf{J}_{g,b}({}^{S}_{E}\hat{\mathbf{q}}, {}^{E}\hat{\mathbf{b}}) = \begin{bmatrix} \mathbf{J}_g^T({}^{S}_{E}\hat{\mathbf{q}}) \\ \mathbf{J}_b^T({}^{S}_{E}\hat{\mathbf{q}}, {}^{E}\hat{\mathbf{b}}) \end{bmatrix}$$

$${}^{S}_{E}\mathbf{q}_{\nabla} = {}^{S}_{E}\hat{\mathbf{q}}_{est,t-1} - \mu_t \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|}$$

It is common to take multiple steps each iteration to find an accurate value when operating with a gradient decent algorithm. However, in order to shorten the number of arithmetic operations, it is acceptable to take just one step if the step length μ_t is at least equal to or greater then the physical orientation change of the sensors. To avoid taking an unnecessary large step, μ_t can be calculated using the physical orientation change rate measured by the gyroscope ${}^{S}_{E}\dot{\mathbf{q}}_{\omega,t}$, the sampling period Δt and an amplifier α.

$$\mu_t = \alpha \|{}^{S}_{E}\dot{\mathbf{q}}_{\omega,t}\| \Delta t, \quad \alpha > 1$$

Fusion of Angular rate and Vector observation orientations

The angular rate and vector observation attitude estimations are fused together in the equation below, where they are weighted by γ_t and $(1 - \gamma_t)$.

$${}^{S}_{E}\mathbf{q}_{est,t} = \gamma_t {}^{S}_{E}\mathbf{q}_{\nabla,t} + (1 - \gamma_t) {}^{S}_{E}\mathbf{q}_{\omega,t}, \quad 0 \leq \gamma_t \leq 1$$

The optimal value of γ_t can be expressed in an equation consisting of the convergence rate of ${}^{S}_{E}\mathbf{q}_{\nabla,t}$, $\frac{\mu_t}{\Delta t}$, and the divergence rate of ${}^{S}_{E}\mathbf{q}_{\omega,t}$, β.

$$\gamma_t = \frac{\beta}{\frac{\mu_t}{\Delta t} + \beta}$$

Assuming that the amplifier α of the step length in the gradient decent algorithm is very large, things can be simplified. This means ${}^{S}_{E}\mathbf{q}_{\nabla,t}$ can be rewritten.

$${}^{S}_{E}\mathbf{q}_{\nabla,t} \approx -\mu_t \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|}$$

The assumption also means that the expression of the fusion weight variable γ_t can be shortened. If μ_t is dominant in the equation, two alternative expressions can be derived.

$$\gamma_t \approx \frac{\beta \Delta t}{\mu_t} \qquad \gamma_t \approx 0$$

Substituting these simplified expressions into the fusion equation, using both variations of γ_t, leaves us with the equation:

$${}^{S}_{E}\mathbf{q}_{est,t} = \frac{\beta \Delta t}{\mu_t} \left(- \mu_t \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|} \right) + (1 - 0) \left({}^{S}_{E}\hat{\mathbf{q}}_{\omega,t-1} + {}^{S}_{E}\dot{\mathbf{q}}_{\omega,t} \Delta t \right)$$

We can simplify the expression one more time. If ${}^{S}_{E}\dot{\mathbf{q}}_{est,t}$ is the rate of orientation change, where ${}^{S}_{E}\hat{\dot{\mathbf{q}}}_{\epsilon,t}$ represents the error direction, the estimated orientation of the sensors ${}^{S}_{E}\mathbf{q}_{est,t}$ can be rewritten.

$$ {}^{S}_{E}\mathbf{q}_{est,t} = {}^{S}_{E}\hat{\mathbf{q}}_{est,t-1} + {}^{S}_{E}\dot{\mathbf{q}}_{est,t}\Delta t $$

$$ {}^{S}_{E}\dot{\mathbf{q}}_{est,t} = {}^{S}_{E}\dot{\mathbf{q}}_{\omega,t} - \beta {}^{S}_{E}\hat{\dot{\mathbf{q}}}_{\epsilon,t} $$

$$ {}^{S}_{E}\hat{\dot{\mathbf{q}}}_{\epsilon,t} = \frac{\nabla \mathbf{f}}{\|\nabla \mathbf{f}\|} $$

Magnetic distortion and Gyroscope drift compensation

The Madgwick filter compensates for magnetic distortion by using the accelerometer measurements as an additional reference to the orientation measured from the magnetometer data. The accelerometer can only help stabilize the errors caused by magnetic disturbance in the vertical plane, because the accelerometer does not have information regarding the horizontal heading. This means that the magnetic disturbance only affects the heading estimations. This operation is described in Group 1 in [fig 2.11]
The Madgwick filter also compensates for gyroscope bias drift, which is a necessary task for all orientation estimation. The filter makes use of the estimated error in orientation rate change ${}^{S}_{E}\hat{\dot{\mathbf{q}}}_{\epsilon,t}$ to correct the gyroscope measurements, using an internal gain ζ. This is illustrated in Group 2 in [fig 2.11]. [12]

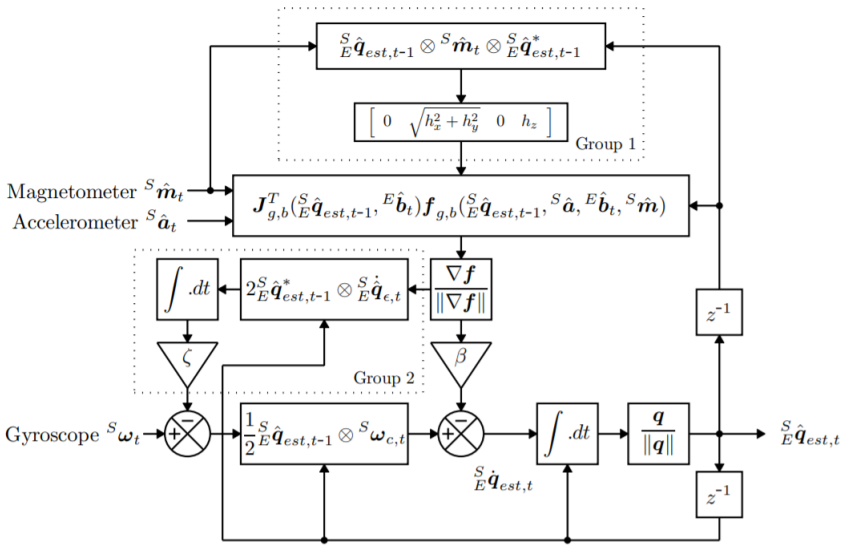


Figure 2.11: An overview of the madgwick filter sensor fusion.[12]

2.4 Hardware Limits

Hardware limitations are the solid walls we have to operate within. It is physically not possible to do any operation that the hardware can not handle

2.4.1 Temperature Dependency

All electronics generate heat. Heat will affect conductivity in components, metal will expand, non-metals might be damaged. On the other end of the spectrum, lack of heat will also affect conductivity in components, materials turn brittle, and functionality might cease to work. Because of this sensors, and electrical components in general, are often rated to work within a certain temperature range, the LSM9DS1 IMU for example is rated to work in the range $[-40°C, 85°C]$. Outside of this range the IMU could still work of course, but mileage may vary.

Chapter 3

Interfacing

3.1 Getting started

When starting the project we first had to familiarize ourselves with the hardware, knowing what you have to work with is helpful for knowing what you can or cannot achieve. There are two pieces of hardware we have to examine, the Arduino UNO and the LSM9DS1, but while we use the Arduino and will refer to the Arduino, we really mean the ATmega328p microcontroller. We will not be using any Arduino specific libraries for this project, it is however easier to work with because the Arduino takes care of the interfacing when you want to program the microcontroller.

3.1.1 Hardware overview

Arduino UNO

The Arduino UNO is a relatively cheap microcontroller board, among its features we find 14 digital pins and of course the ATmega328p microcontroller itself, these two things are really the only parts of the Arduino UNO we are interested in for the purposes of this project. The device runs on 5V power and can be powered either by USB when connected to a PC, or by a power plug if standalone operations. When powered by USB you can also program the microcontroller without having to worry about turning power off or damaging the device, the board takes care of the interfacing as long as you use an IDE that supports this functionality.

LSM9DS1 breakout board

The Sparkfun LSM9SD1 breakout board features the LSM9DS1, and that's pretty much it. the breakout board has labels for the I/O pins and makes the LSM9DS1 accessible without any hassle. The LSM9DS1 itself is a tiny black package that features a MEMS gyroscope, accelerometer, and magnetometer, all in three dimensions for a full six degrees of freedom motion tracking. The sensors can be configured by changing some control registers on board the chip, and the output from the sensors are similarly read from registers. These registers all have their unique address that Sparkfun made a very convenient header file for, so instead of having to look up the numeric address for every register we can write them by name. The IMU supports either I2C or SPI communication, we chose SPI because that's what we're

used to working with. The LSM9DS1 operates on 3V power and is powered by the connecting a supply voltage to the power pin.

Extras

The keen of eyes might have spotted an incompatibility with our two hardware pieces, namely the operating power. The Arduino UNO operates on 5V while the LSM9DS1 operates on 3V. And while the Arduino has a 3V power supply pin, the digital pins on the board will be expecting or supplying 5V when reading or writing, this is mainly a concern for SPI communication. If you want to use I^2C you don't need to worry about this. There are two ways to solve this issue, one way is to under supply and underclock the Arduino, but a far simpler way is to put a bi-directional level converter between the Arduino and the LSM9DS1. A bi-directional level converter does just that, 5V power goes in on one end, 3V comes out the other and vice versa. We ended up buying a couple of Sparkfun manufactured bi-directional level converters with four channels each. Four channels are enough for all the I/O we want to hook up for this project.
Lastly we need some way to connect everything, the best option here would be to make a custom PCB for the level converter and the LSM9DS1, but it's far simpler for our purposes to just use a breadboard. Breadboards are not perfect, especially cheap ones can have problem with stray capacitance and bad connections with the rails. For us these problems aren't a very big deal, but for best results, or if you want to expand upon this project yourself we'd recommend making a PCB.

3.1.2 Hookup guide

Now it's time to hook everything up, the process is actually very simple, see [fig 3.1] for an easy reference. Connect 3V and ground to the power rails, and 5V to the high power side of the level converter and 3V power to the low power side, then connect VDD and GND on the LSM9DS1 to 3V and GND rails. Now the ATmega328p has some pins dedicated to SPI communication, these are on the Arduino UNO labeled as 13 for SCK, 12 for MISO and 11 for MOSI. 10 is suggested as the default CS/SS, but almost any pin can be used for CS and we will actually need to use two CS pins. If you are working directly on the ATmega328p the pins are in the same order: PB5, PB4, PB3, and PB2. Connect SCK, MOSI, and two CS pins to the high side of the level converter, then from the low power side connect them to SCL, SDA, CS AG and CS M respectively. lastly short SDO M and SDO AG together and connect them to MISO, the output from the LSM9DS1 does not need to be converted up to 5V because the Arduino recognizes 3V as HIGH even if it's not ideal.

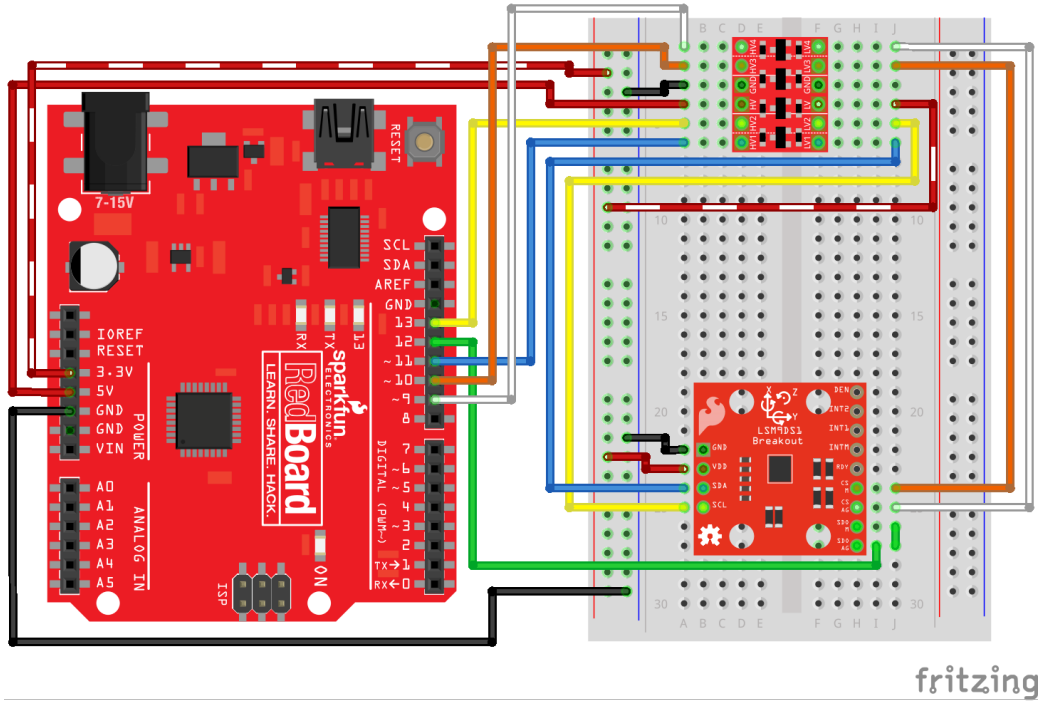


Figure 3.1: Diagram of how to connect the Arduino board and the LSM9DS1 breakout board for SPI communication. [13]

3.1.3 SPI Communication

With all the hardware taken care of we can now start writing some code. And the first thing we need to take care of is communication with the LSM9DS1. To do this we first initialize the SPI on the Arduino with these lines of code:

```
void spiInit(void){
   DDRB = (1<<DDB5)|(1<<DDB3)|(1<<DDB2)|(1<<DDB1); // Set data direction
       for MOSI, SCK, CS_M and CS_AG as output
   PORTB = (1<<PORTB2)|(1<<PORTB1); // CS_M and CS_AG must start HIGH
       for the init to work.
   SPCR = (1<<SPE)|(1<<MSTR)|(1<<CPOL)|(1<<CPHA)|(1<<SPR1)|(1<<SPR0); /*
       SPI enable, Master mode, MSB first, Clockdiv 128
   Clock idle HIGH, Data Captured on Trailing flank.*/
}
```

The last line setting the SPCR register is where the magic really happens so let's break it down. the SPE bit is what tells the microcontroller that SPI needs to be enabled, and MSTR tells the microcontroller that it needs to behave as master. We need to be in master mode because the Arduino is the one that has to start all the communication. MSB first is actually a default mode so we don't write any bit for it, but we want to make sure that it's explicitly mentioned in the code that we need MSB first. CPOL and CPHA set the clock polarity and clock phase, this dictates if the clock idles high or low when not in use, and whether leading or trailing flank of each pulse is setup or action. This is more easily observed a diagram taken from the LSM9DS1 data sheet, see [fig 3.2]

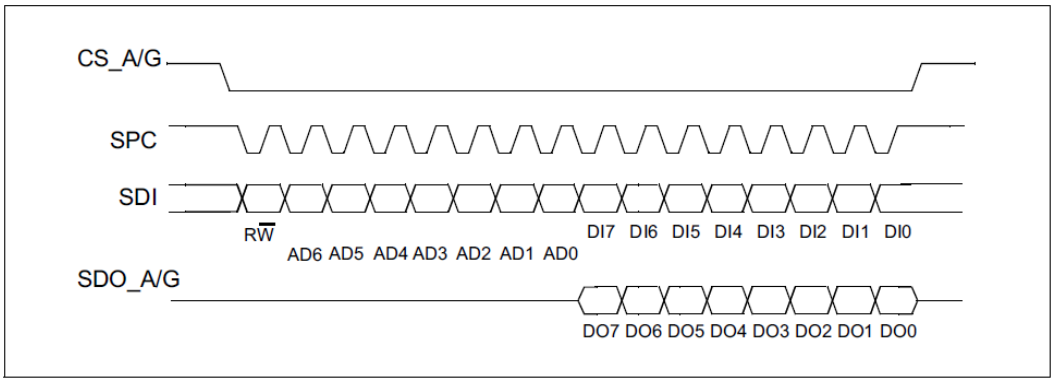


Figure 3.2: Diagram from LSM9DS1's datasheet explaining the read and write protocol for SPI. [7]

Here we can see that the LSM9DS1 expects the clock to idle high, any pin change is driven on the falling(leading) edge of the clock pulse and action is captured on the rising(trailing) edge of the clock pulse. We need to adhere to this, so with CPOL and CPHA both set to one, we configure the Arduino SPI to behave the same way. Lastly we have the clock divider, with SPR1 and SPR0 both set to one we get a clock division of 128. This is the highest clock division we can get and it slows the SPI clock down to a mere 125kHz, which is fast enough for anything we want, but much slower than the maximum possibly 4MHz clock speed. The reason we go slow is because we're on a breadboard, stray capacitance between the rails becomes an increasing risk of interference as you go up in speed, but there really shouldn't be any problems with running this in 4MHz or even faster.
Now that SPI is enabled we need to write some functions to use it. Since SPI sends and receives data simultaneously we can write a single transfer function as our baseline and then call this function in more specific read / write functions, this reduces clutter and makes the code easier to read.

```
uint8_t spiTransfer(uint8_t data) {
   SPDR = data;
   /*
    * The following NOP introduces a small delay that can prevent the
       wait
    * loop form iterating when running at the maximum speed. This gives
    * about 10% more speed, even if it seems counter-intuitive. At lower
    * speeds it is unnoticed. // Ref. Arduino SPI.C
    */
   asm volatile("nop");
   while (!(SPSR & (1<<SPIF))) ; // wait for SPIF flag to be set
   return SPDR;
 }
```

SPDR is the SPI Data Register, and communication is initialized the moment a byte is placed in the register, we then need to wait for communication to finish by looking at the SPIF flag in the SPI Status Register. SPIF is normally zero but when a transaction is complete it's set to one. The flag is then cleared automatically if

we look at the flag and then immediately access SPDR, which is what we end up doing. Now that we can transfer one byte of data we can make our read and write functions. It's useful to separate these because we don't always care about doing both.
For starters we can make our write function, this will be useful for configuring the control registers on board the IMU. However we must first understand how the IMU expects to be written to. looking back to [fig 3.2] we can actually discern quite a lot. When communication starts the first bit on the SDI, that is MOSI, indicates whether or not we're about to execute a read or a write operation. The following seven bits are the address bits, every register on the LSM9DS1 has a unique address, the full list of addresses can be found in the LSM9DS1 data sheet. The last eight bits are the actual byte we're reading or writing, so to write one byte of data to a register we need to send two bytes. Our SPI write function looks like this:

```
void SPIwriteByte(uint8_t csPin, uint8_t subAddress, uint8_t data)
  {
     PORTB &= ~(1<<csPin); // Initiate communication

     // If write, bit 0 (MSB) should be 0
     spiTransfer(subAddress & 0x3F); // Send Address
     spiTransfer(data); // Send data

     PORTB |= (1<<csPin); // Close communication
  }
```

The function takes in arguments for which chip select, what address and the data byte itself and then does two transfer. Dragging the the chip select pin LOW starts SPI communication, we then send the address using the transfer function we wrote above. Since we always want the MSB to be zero when writing, and since there are no target addresses where the following bit is a one, we do an AND operation with 0011 1111 and the target address value to make doubly sure that the first two bits can't be wrong. We then use the SPI transfer function again to send the actual data we want to write into the target register, chip select is toggled back to high again to stop the transaction.
Reading is a slightly more complicated process. Reading from the magnetometer and accelerometer/gyroscope registers is slightly different, and we generally want to read multiple bytes in a burst, rather than just one byte. Let's first write the function, then break it down:

```
uint8_t SPIreadBytes(uint8_t csPin, uint8_t subAddress, uint8_t * dest,
    uint8_t count)
{
  // To indicate a read, set bit 0 (MSB) of first byte to 1
  uint8_t rAddress = 0x80 | (subAddress & 0x3F);
  // Mag SPI is different. If we're reading multiple bytes,
  // set bit 1 to 1. The remaining six bytes are the address to be read
  if ((csPin == PIN_M) && count > 1){
     rAddress |= 0x40;
  }
```

```
    PORTB &= ~(1<<csPin); // Initiate communication
    spiTransfer(rAddress);
    for (int i=0; i<count; i++)
    {
        dest[i] = spiTransfer(0x00); // Read into destination array
    }
    PORTB |= (1<<csPin); // Close communication

    return count;
}
```

When reading multiple bytes from the IMU the address you're reading from will automatically increment by one, this makes our job here a lot easier. So we write the function to support multiple reads by default by having number of reads wanted as an argument, we then output to a destination array of bytes. To read just a single byte then you would just set the count to one when calling the function. On the address byte we need to set MSB to one to indicate that we're doing a read operation. We then do the same AND operation on the address as we did in the write function to make sure the data is sanitized. Then we need to check if we're reading from the magnetometer registers or not. When reading multiple times from the magnetometer registers the address needs to be modified so that the bit following MSB is also one. Now that the address is prepared we can start the transaction, the only difference here from the write function is the small for-loop to iterate through until we've reached the desired amount of bytes read.

Initializing the control registers

Now that we have successfully bridged the communication gap between the Arduino and the IMU we can finally start the final part of the setup: initializing the control registers. The process is quite menial, the accelerometer and gyroscope have three unique control registers each and four shared control registers, the magnetometer has five on its own. In addition to these there are other registers for tuning, adjustments and other functionalities the IMU is capable of. So what we're gonna do is simply start from the top of the data sheet, beginning with the registers that affect the gyroscope and make a "gyro init" function that has variables for all the options we would consider changing, with an explanation in a comment. Then, in the same function, we make a the string of bits that we're gonna end up writing to the register by combining values from the relevant options into a single byte. After the byte is sent we clear the tempvalue byte and make a new one for the next register. It ends up looking something like this:

```
void initGyro(void){

    uint8_t gyroEnableX = 1; // 0 for off, 1 for on
    uint8_t gyroEnableZ = 1; // 0 for off, 1 for on
    uint8_t gyroEnableY = 1; // 0 for off, 1 for on

    // bandwidth is dependent on scaling, choose value between 0-3
    uint8_t gyroBandwidth = 0;
    uint8_t gyroLowPowerEnable = 0; // 0 for off, 1 for on
```

```
  uint8_t gyroHPFEnable = 0; // 0 for off, 1 for on
  // HPF cutoff frequency depends on sample rate
  // choose value between 0-9
  uint8_t gyroHPFCutoff = 0;
  uint8_t gyroFlipX = 0; // 0 for default, 1 for orientation inverted
  uint8_t gyroFlipZ = 0; // ----
  uint8_t gyroFlipY = 0; // ----
  uint8_t gyroOrientation = 0b00000000; // 3-bit value
  uint8_t gyroLatchINT = 0; // 0 for off, 1 for on

  uint8_t tempValue = 0;

  /* CTRL_REG1_G
     bit 7-5: output data rate selection, activates gyroscope when
        written
     bit 4-3: full-scale selection
     bit 2:     always 0, don't write
     bit 1-0: bandwidth selection */
  // Default: CTRL_REG1_G = 0x00;
  tempValue = (gyroSampleRate & 0b00000111) << 5;
  tempValue |= (gyroScale & 0b00000011) << 3;
  tempValue |= (gyroBandwidth & 0b00000011);
  SPIwriteByte(PIN_XG, CTRL_REG1_G, tempValue);
  .
  .
  .
```

Most of these are set and forget once you know what you want them to be, having the option to easily change them by just changing the variable, instead of having to manually change bits around makes it easier to quickly change stuff if needed. We make similar init-functions for the accelerometer and the magnetometer.

3.1.4 USART communication

Now that we can read and write, and have written some code for the initial setup of the IMU, we need to figure out a way to see what's going on, otherwise the whole unit will just operate as a black box. The easiest way to get some insight is by utilizing USART communication to print some statements and values. Initializing USART is quite easy, you need only set a baudrate, enable receiver and transmitter and set the data format. All that is done in the following function.

```
void usart_init( uint16_t ubrr) {
  // Set baud rate
  UBRR0H = (uint8_t)(ubrr>>8);
  UBRR0L = (uint8_t)ubrr;
  // Enable receiver and transmitter
  UCSR0B = (1<<RXEN0)|(1<<TXEN0);
  // Set frame format: 8data, 1stop bit
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);

}
```

We need a few more functions before we're ready to use the print statement. One function is putchar, which simply waits for the USART Data buffer to be clear, and then puts the byte you want to send in the data register, which automatically starts the transmission. Then we need a function to keep looping the putchar until the whole string of data has been sent. Together they look like this.

```
void usart_putchar(char data) {
   // Wait for empty transmit buffer
   while ( !(UCSR0A & (1<<UDRE0)) );
   // Start transmission
   UDR0 = data;
}

void usart_pstr(char *s) {
   // loop through entire string
   while (*s) {
      usart_putchar(*s);
      s++;
   }
}
```

In order to print floating points number we need to conduct some final wizardry. First we need to get the AVR/GNU library libprintf_flt, and then we we need to tell the compiler to use the vprintf library. In Atmel Studio 7 we do this by going into toolchain $\longrightarrow$ AVR/GNU linker and in the general tab tick the use vprintf checkbox, and in the libraries tab we add the library manually. Not quite done yet we need one more function and one more line of code before we are able to print floating points. First we need this function:

```
int usart_putchar_printf(char var, FILE *stream) {
   if (var == '\n') usart_putchar('\r');
   usart_putchar(var);
   return 0;

}
```

and this line declared above everything else in the code:

```
static FILE mystdout = FDEV_SETUP_STREAM(usart_putchar_printf, NULL,
    _FDEV_SETUP_WRITE);
```

This creates the stream of data the USART will transmit, and lets us print everything we would want to by using the function printf. Lastly we need some software that can read the data stream, some IDEs have this as a functionality, but we ended up using the software PuTTY, which has great tools for logging and saving the USART data. With all this setup done, we are finally ready to start reading data from the IMU.

3.2 Data Acquisition

Assuming everything in the Getting started chapter went right, our job here in Data Acquisition is very easy, we only need to write a few functions and do some easy mathematics for time keeping, we will get to that later. Let's start by just extracting useful data from the IMU.

3.2.1 Reading from the IMU

Each sensor has six output registers, one high byte and one low byte for each axis, x, y and z. The order of the registers is fixed and goes low byte x, high byte x, low byte y, high byte y, low byte z, high byte z. As you might remember, reading more than one byte in one go automatically increments the address. This means that in order to read all the data from one sensor, we need only start reading the low byte x register, and read a total of six bytes. We then combine the low and high byte for a 16-bit value and we're done. the function ends up being only four lines due to the proper groundwork done earlier. Reading from the magnetometer looks like this:

```
void readMag(void){
   uint8_t temp[6]; // We'll read six bytes from the mag into temp
   SPIreadBytes(PIN_M, OUT_X_L_M, temp, 6);
   mx = (temp[1] << 8 | temp[0]);
   my = (temp[3] << 8 | temp[2]);
   mz = (temp[5] << 8 | temp[4]);
}
```

Similar functions are made for the gyroscope and accelerometer, making sure to set the correct chip select pin and start at the correct register.

Polling rate

There is one more thing we need keep in mind for data acquisition, the gyroscope has to operate on a fixed polling rate. Since the data from the gyroscope has to be integrated to be useful, we need to know the exact time step between each sample. We actually already set the speed when we initialized the gyroscope control registers, there are six options to choose between, and while faster is better and more precise, it also demands a faster run time. If the software is too slow to keep up everything will go bad very quickly. On the other hand, if the program speed is too fast we need to halt it and wait for a new gyroscope sample to be ready before it does a new round of data gathering. In order to do this we need a clock that runs independently of the program that we can check on. Luckily the Arduino has clocks to spare, we don't even need to set any bits to start them since we're going to be using them in the default configuration. All we need to do is apply a clock divider to start the timer, it doesn't run otherwise. We also don't want the clock to count too fast because then it could overflow before the gyroscope is ready with a new sample. This isn't necessarily a problem, just like how our 24 hour clock resetting at midnight isn't a problem, but life is easier if we don't have to keep track of overflows.
So we set the gyroscope to one of it's six output data rates (ODR) see [fig 3.3], for example 119Hz. Now the main loop of the program has 8.4 milliseconds to complete all its tasks and get ready for a new sample. We can not simply just count to 8.4

Table 46. ODR and BW configuration setting (after LPF1)

ODR_G2	ODR_G1	ODR_G0	ODR [Hz]	Cutoff [Hz](1)
0	0	0	Power-down	n.a.
0	0	1	14.9	5
0	1	0	59.5	19
0	1	1	119	38
1	0	0	238	76
1	0	1	476	100
1	1	0	952	100
1	1	1	n.a.	n.a.

1. Values in the table are indicative and can vary proportionally with the specific ODR value.

Figure 3.3: Table of the the Gyro ODR, taken from the LSM9DS1 data sheet. [7]

milliseconds however, the clock counts in terms of ticks, so the maths for how many ticks we need to count works out to be:

$$\frac{16 \cdot 10^6 Hz}{clc\_div} \cdot \frac{1}{ODR}$$

So with 119Hz ODR and a clock division of 64 we need to count to 2101 every loop. Now we could just put the whole main program inside this while loop:

```
while(TCNT1 < 2101){
.
.
.
TCNT1 = 0;
}
```

But this is inefficient since the program has to constantly check back with the criteria for the whole loop. Instead what we do is reset the timer at the beginning of the main program loop, and at the very end we check if we ran fast enough or if we have to break the program because it missed its time window.

```
timerInit(); // Start the clock
while(1){
.
.
.
   if((TCNT1 > timerticks) | (TIFR1 & (1<<TOV1))) {
         temp = TCNT1;
         printf("Game over! You were too slow! \n");
         printf("Clock cycles lapsed: %u\n", temp);
         printf("Clock cycles limit: %u\n", timerticks);
         while(1); // stop the program
   }
   while(TCNT1 < timerticks); // Wait for the next gyro sample to be
      ready
   TCNT1 = 0x0000; // Reset the timer
```

```
}
```

We also check the timer overflow flag because it's always possible that we ran so slow that the clock had one or multiple overflows, but coincidentally we're less than 2101 ticks into the new timer.

Chapter 4

Signal Processing

This chapter will be focused on how we are manipulating the data in the way that fits our system the most. It will include everything from our sensitivity settings on the IMU, calibration of the sensors, combining measurements in the Madgwick filter, and finally presentation of the measured orientation in roll, pitch and yaw.

4.1 Transforming Data and Calibration

The data stream coming from the IMU is in the form of raw signed 16 bit values, making them range from -32768 to 32767. To extract any useful information from this data we need to transform and scale the numbers so that they properly represent the measurement unit they are. We also need to conduct some calibrations to get the best possible data we can from our sensors.

4.1.1 Gyroscope

Let's start by calibrating the gyroscope. When placed on a stable surface we would expect the gyroscope to output zero on all axes, without calibration this won't be the case, and even with calibration there will be some errors. We have to make do with what we can, so let's write our calibration function.

```
void calibrateGyro(void){
   int32_t gBiasRawTemp[3] = {0, 0, 0};
   printf("Calibrating gyroscope, hold the device still\n");
   for(int i = 0; i<1000; i++){
      if(i % 100 == 0)printf(".");
      readGyro();
      gBiasRawTemp[0] += gx;
      gBiasRawTemp[1] += gy;
      gBiasRawTemp[2] += gz;
      _delay_ms(9); // Wait for guaranteed new data.
   }
   printf("\n");
   gBiasRawX = gBiasRawTemp[0] / 1000;
   gBiasRawY = gBiasRawTemp[1] / 1000;
   gBiasRawZ = gBiasRawTemp[2] / 1000;
```

```
}
```

Basically we read 1000 samples while the gyro is stationary, add up all the readings and then divide by 1000 to get the average bias for each axis. This bias will then later be subtracted from all future readings before we manipulate the data more on the journey to get the attitude of the device. Also note that we have hardcoded in a delay here, this delay has to be changed to reflect the ODR of the gyroscope.
After adjusting for bias with the calibration we need to scale the data to properly reflect the sensitivity, we chose a FS of ± 500 because that lets us keep up with rotations up to about 1.3 full rotations per second, which is pretty fast. That means we have to multiply with the sensitivity factor of 0.0175, and while we're doing multiplication we also multiply the number by pi and divide by 180. The gyroscope data needs to temporarily be represented as rads per second instead of degrees per second, because the Madgwick filter later on is expecting the data to come in this form. The data manipulation ends up looking like this:

```
    readGyro();
    gx -= gBiasRawX;
    gy -= gBiasRawY;
    gz -= gBiasRawZ;
    // convert gyroscope data to rad/s:
    gyro_x = gx * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));
    gyro_y = gy * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));
    gyro_z = gz * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));
```

Here "SENSITIVITY_GYROSCOPE_500" is a defined constant, we have similar constants defined for the two other FS options, this way it's easier to spot where to update the code if we wanted to change the full scale.

4.1.2 Accelerometer

When it comes to the accelerometer we're going to do much of the same as with the gyroscope. An additional requirement for the calibration is that the device has to be placed on a level surface with the z-axis being the only one affected by gravity, any axis can be used instead of the z-axis of course but then you'd have to edit the code so that $1g$ is subtracted from the correct axis during calibration. The calibration function ends up looking like this:

```
void calibrateAccel(void){
  int32_t aBiasRawTemp[3] = {0, 0, 0};

  printf("Calibrating Accelerometer, please put the device on a stable,
      level surface.\n");
  for(int i = 0; i<1000; i++){
    if(i % 100 == 0)printf(".");
    readAccel();
    aBiasRawTemp[0] += ax;
    aBiasRawTemp[1] += ay;
    aBiasRawTemp[2] += az - (int16_t)(1/SENSITIVITY_ACCELEROMETER_8);
    _delay_ms(9); // Wait for guaranteed new data.
  }
```

```
    printf("\n");
    aBiasRawX = aBiasRawTemp[0] / 1000;
    aBiasRawY = aBiasRawTemp[1] / 1000;
    aBiasRawZ = aBiasRawTemp[2] / 1000;
}
```

As you can see, the function is suspiciously similar to the gyroscope calibration function, we only have to subtract $1g$ from the axis we expect to be facing up. Notice the similar delay, when the accelerometer and gyroscope are both turned on the accelerometer ODR is tied to the gyroscope. Going from raw data to gs is then simply the same subtraction of the bias, and then multipliciation with the sensitivity factor, similar to how the gyroscope data is handled.

4.1.3 Magnetometer

A three-axis magnetometer measures the magnetic field strength along three orthogonal axes. The combined magnitude of the axis measurements will in an ideal world be equal to the magnetic field strength, $|b|$. The magnetic field strength should not change when the magnetometer rotates.

$$|b| = \sqrt{{m_x}^2 + {m_y}^2 + {m_z}^2}$$

The magnetic field strength and magnetic declination is dependent on our location. The inclination, which is "dipping" of the magnetic field, also needs to be accounted for depending on location. By examining the readings on the different axis the orientation of the sensor can be determined.

Magnetometer calibration

It is necessary to calibrate the magnetometer to make use of measurements. The two main error sources are called the hard iron and soft iron effects, where hard iron errors are most significant. To compensate for these errors we use an embedded function in Matlab, *magcal*, where we get the values needed to calibrate the magnetometer.

```
% Matlab:
[A,b,magB] = magcal(d);
m_cal = (d-b)*A;
```

The input d is a n-by-3 matrix containing magnetometer measurements. A is a 3-by-3 matrix used to compensate for soft iron distortion, b is a 1-by-3 vector that corrects the hard iron offset error, and $magB$ is a scalar that shows the strength of the magnetic field. [8]

$$d = \begin{bmatrix} m_{x,1} & m_{y,1} & m_{z,1} \\ \vdots & \vdots & \vdots \\ m_{x,n} & m_{y,n} & m_{z,n} \end{bmatrix} \quad A = \begin{bmatrix} A_{xx} & A_{yx} & A_{zx} \\ A_{xy} & A_{yy} & A_{zy} \\ A_{xz} & A_{yz} & A_{zz} \end{bmatrix} \quad b = \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

The calibration is done by collecting a high number of samples of the uncalibrated measurements in the matrix d, while rotating the sensor to cover all orientations.

The values of the outputs A and b are manually transferred to the C program where the calibration process is implemented. Six values of A is sufficient, as the matrix is symmetric and three values repeat themselves.

```
void softIronMag(float xx, float yy, float zz, float xy, float xz, float
    yz, float b1, float b2, float b3){
   // A = [xx,yx,zx ; xy,yy,zy ; xz,yz,zz]
   // b = [b1,b2,b3]
   float m_x = mag_x - b1;
   float m_y = mag_y - b2;
   float m_z = mag_z - b3;
   mag_x = m_x*xx + m_y*xy + m_z*xz;
   mag_y = m_x*xy + m_y*yy + m_z*yz;
   mag_z = m_x*xz + m_y*yz + m_z*zz; }
```

This function calibrates and updates the global measurements mag_x, mag_y and mag_z consecutively as the measurements are made. The soft iron calibration is applied by transferring smaller parts of the two other axes measurements to the axis that is being adjusted. The parts are weighted by the values in A, which has values close to one in the diagonal and zero elsewhere. If the magnetometer does not suffer from any soft iron distortions, A would be an identity matrix.
The function is called softIronMag, even though it compensates for hard iron as well. This is because the hard iron compensation already is implemented into the IMU's magnetometer offset register directly, which accounts for the majority of the error. The hard iron part in this function only adjusts the measurements slightly.
The results of calibration can be seen in [fig 5.3] in the results chapter.

4.2 Madgwick filter

Overview

The Madgwick filter was developed by Sebastian Madgwick as a part of his Ph.D research in 2009. In his work he proposed a new and efficient orientation filter as an alternative to the traditional Kalman-based solutions.
The proposed filter uses tri-axis measurements from the gyroscope, accelerometer and magnetometer, or just from the gyroscope and accelerometer, to estimate the attitude of the sensor suite. A simplified block diagram can be seen in [fig 4.1]
A benefit of this filter is the low computational processing load, which makes it suitable for low power modes or applications that have limited processing power, as well as applications requiring high sampling rates.

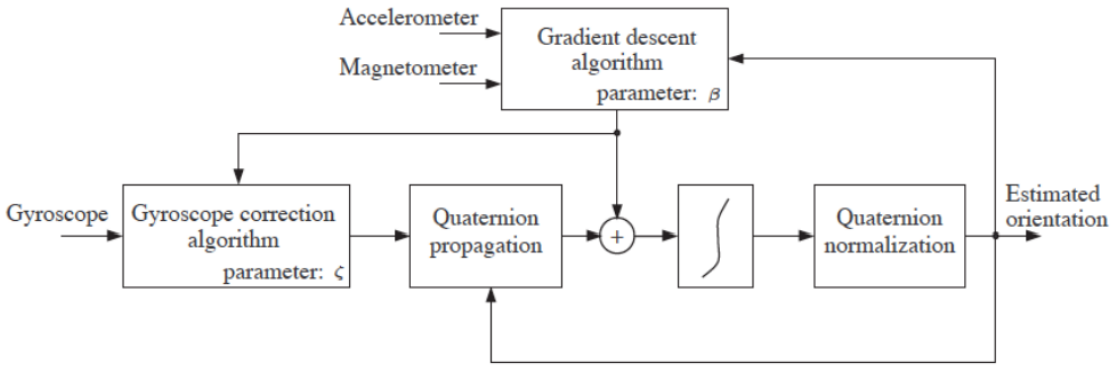


Figure 4.1: Block diagram of Madgwick filter. [14]

Implementation

A C code of the filter is published and free to use. We implemented the code into our own, named madgwick.c. The pre-written function uses the calibrated sensor measurements as input to update the quaternions, that represent the orientation of the sensor suite. The gyroscope input needs to be given in $[radians/sec]$ in order for the filter to function. The unit of the accelerometer and magnetometer measurements do not matter, as they are normalized before use. The sample frequency of the gyroscope data has to be correctly set for the filter to work.

Filter gain

The gain, β, is an adjustable parameter that decides the weighing of the orientation error calculated by the gradient descent algorithm, used to correct the attitude calculated by the gyroscope measurements alone.
The most significant disadvantage of the Madgwick filter is the need of setting the gain manually. The gain influences the results heavily, and has to be tuned in order for the filter to work as intended. Distinct β values have different strengths. A large value makes the filter more capable of handling instant orientation changes, see [fig 4.2] and [fig 4.3]. While a smaller value makes the stationary orientation estimations more stable, see [fig 4.4]. To utilize the strengths of different gain values, it may be beneficial to have a variable β value depending on the rate of orientation change. [12]

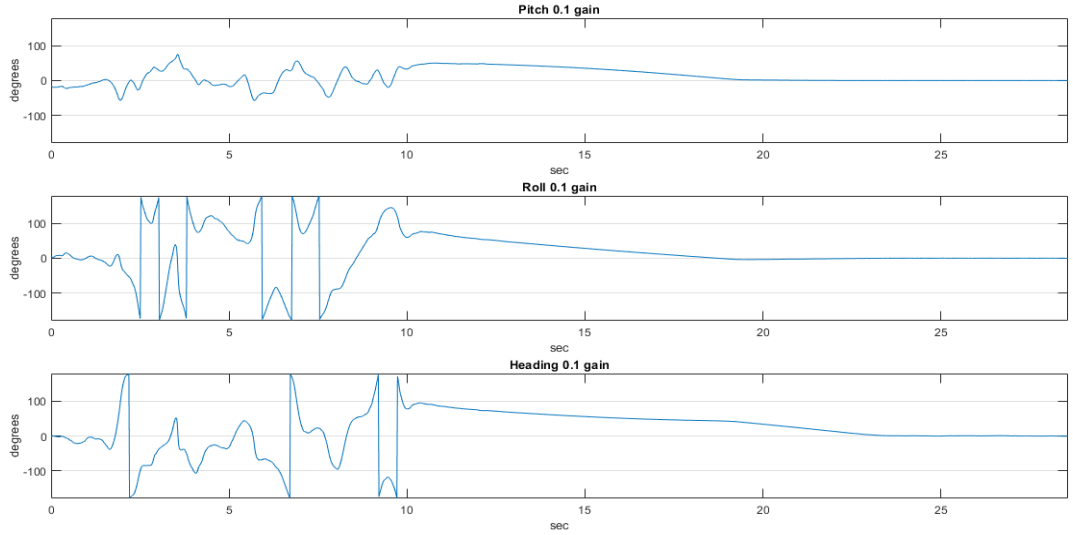


Figure 4.2: 0.1 gain settling time performance. The device is shaken randomly and then placed on a steady surface

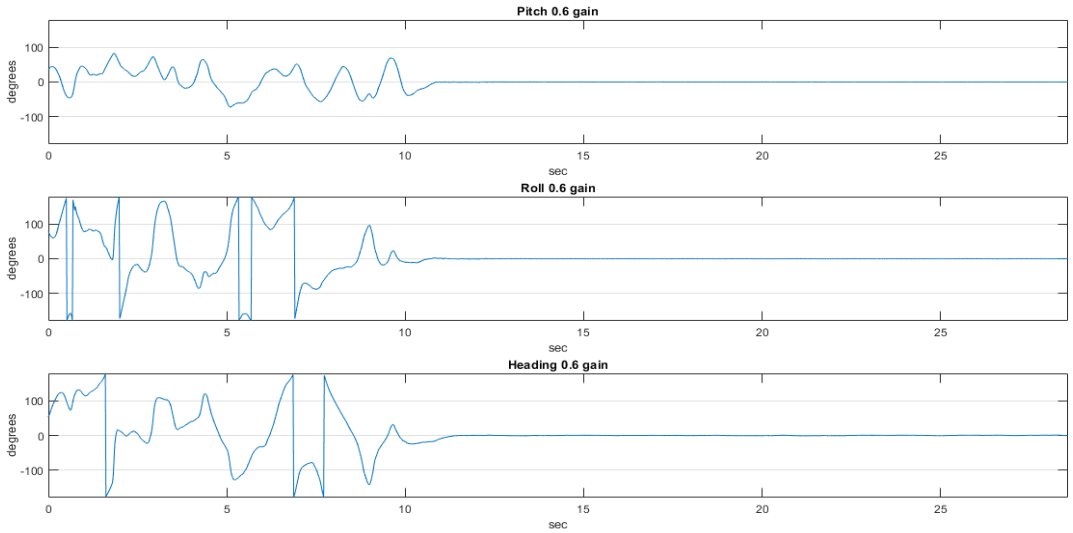


Figure 4.3: 0.6 gain settling time performance. The device is shaken randomly and then placed on a steady surface

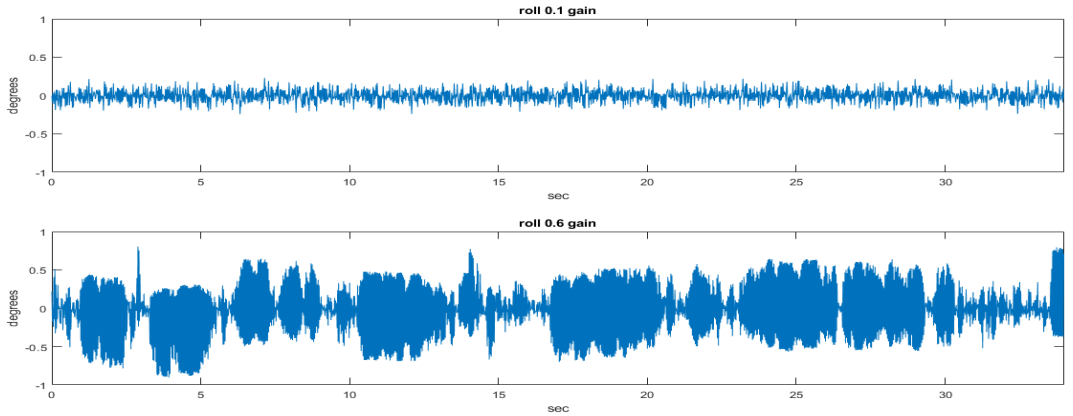


Figure 4.4: Comparison of 0.1 and 0.6 gain applied on stationary measurements

Euler Angles

To get more convenient numbers to look at, the quaternion orientation is transformed to Euler angle representation by using the function "QuaternionsToEuler". Euler angles are the rotations about the x, y and z axis named roll ψ, pitch θ and heading ϕ, and can be found by working out the following equations.

$$\psi = atan2\big(2(q_0q_1 - q_2q_3), 1 - 2({q_1}^2 + {q_2}^2)\big)$$

$$\theta = asin\big(2(q_0q_2 - q_1q_3)\big)$$

$$\phi = atan2\big(2(q_0q_3 + q_1q_2), 1 - 2({q_2}^2 + {q_3}^2)\big)$$

The pitch angle cannot portray angles exceeding $\pm 90°$ because of the nature of the *asin* function. It has no way of detecting quadrant change. An input value of 0.5 would for example always give an output angle of 30°, even if the true angle can be both 30° and 150°. This is not an issue for roll and heading, as they can be calculated using the C function *atan2*.

4.3 Data presentation

The final step for us is data visualisation, if we can't see what the IMU reports, we don't know if it works. Having already converted the quaternions to Euler angles at the end of our program all we need to do now is print them in a tabular format to easily ingest the data in MATLAB. We chose to make all our graphs in MATLAB not only because it's easier, but because making graphs and figures is not a functionality the ADCS device needs. The data would normally be passed on to the next stage of the satellite's control system without any human oversight.

```
// Print Euler angles.
    printf("Pitch: %f\t", angle_pitch);
    printf("Roll:  %f\t", angle_roll);
    printf("Yaw:   %f\n", angle_yaw);
```

We then use the terminal software PuTTY to log and save the data stream to a text file. The file is then imported into MATLAB and then we make a simple script to plot the data points with sensible axis scaling. An example of a script looks like this:

```
%MATLAB code
angles = readtable('axis_rotation.txt');
Pitch = angles{1:size(angles,1),1};
Roll = angles{1:size(angles,1),2};
Heading = angles{1:size(angles,1),3};

Pitchscaled = Pitch(1:2000);
Rollscaled = Roll(1:2000);
Headingscaled = Heading(1:2000);

t = 0:1/59.5:(1/59.5)*(2000-1);
figure
```

```
subplot(3,1,1);
plot(t, Pitchscaled)
title('Pitch');
ylabel('degrees');
xlabel('sec');
xlim([0 max(t)])
ylim([-120 120])
set(gca, 'YGrid', 'on', 'XGrid', 'off') %Horosontal grid

subplot(3,1,2);
plot(t, Rollscaled)
title('Roll');
ylabel('degrees');
xlabel('sec');
xlim([0 max(t)])
ylim([-120 120])
set(gca, 'YGrid', 'on', 'XGrid', 'off')

subplot(3,1,3);
plot(t, Headingscaled)
title('Heading');
ylabel('degrees');
xlabel('sec');
xlim([0 max(t)])
ylim([-120 120])
set(gca, 'YGrid', 'on', 'XGrid', 'off')
```

This plots the pitch, roll and heading each in its own subplot. We start by importing all the data and sorting it into arrays, we then chose how many samples we want to include in the plot. We don't want to use the full length because the last line might have gotten cut off or interrupted half-way through the printing. Lastly we scale the x-axis to properly represent time, as you can see these measurements were taken with an ODR of 59.5Hz.

Chapter 5

Results

This chapter contains the results of the processes from chapter three and four. They will later be discussed in the Discussion in section 6.2.

5.1 Sensor calibrations

These are the results from section 4.1 Transforming Data and Calibration.
Gyroscope and accelerometer
The easiest way to showcase the results of calibration is to look at the offset value when the device is stationary before and after calibration. see [fig 5.1], [fig 5.2].
Magnetometer
The magnetometer calibration figure shows a plot of many magnetometer samples captured while turning the IMU around itself, before and after calibration. see [fig 5.3]

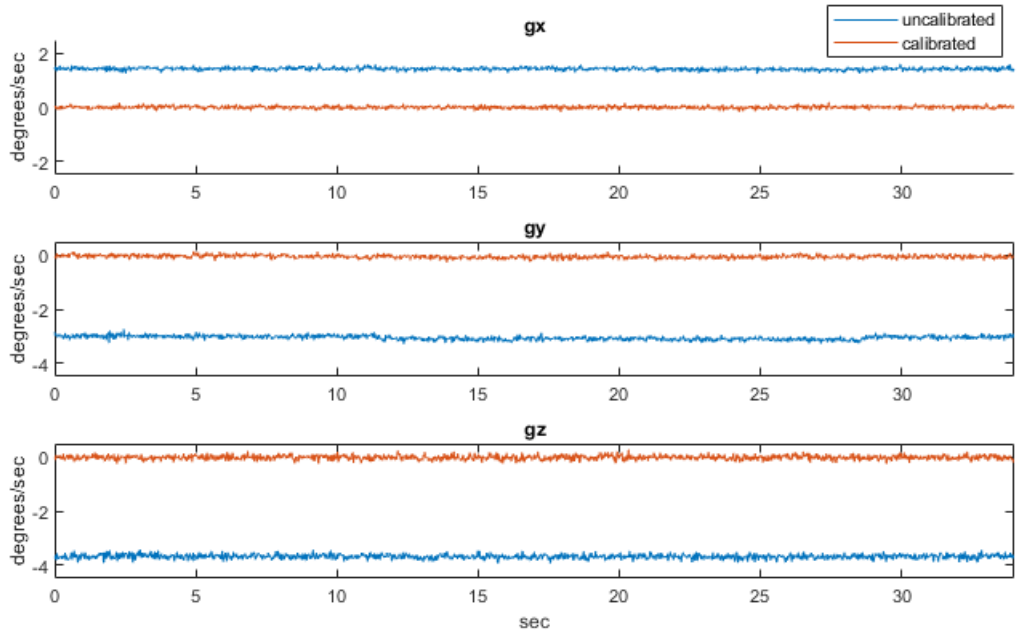


Figure 5.1: Gyroscope calibration, stationary offset on all axes before and after calibration.

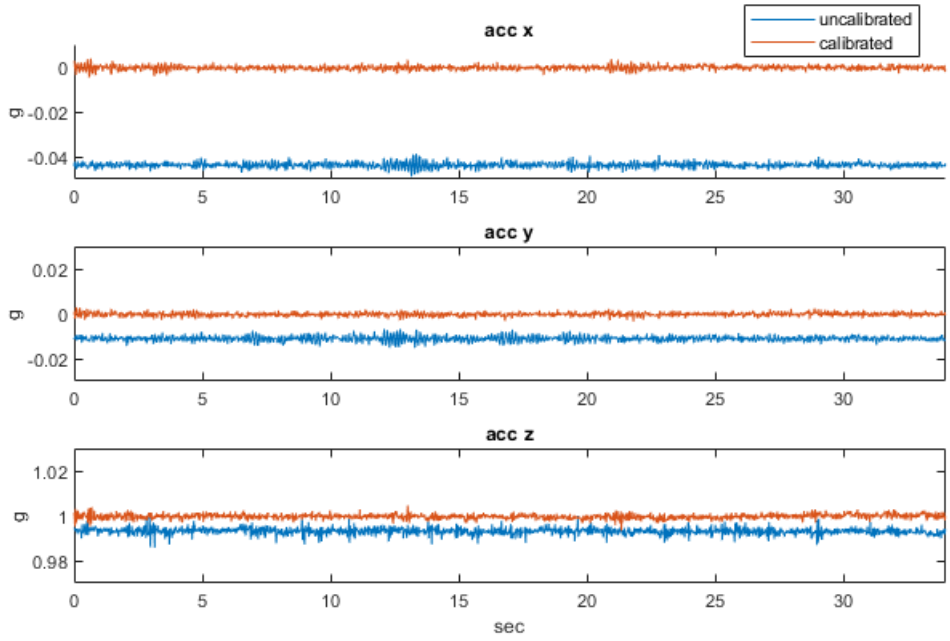


Figure 5.2: Accelerometer calibration, stationary offset on all axes before and after calibration.

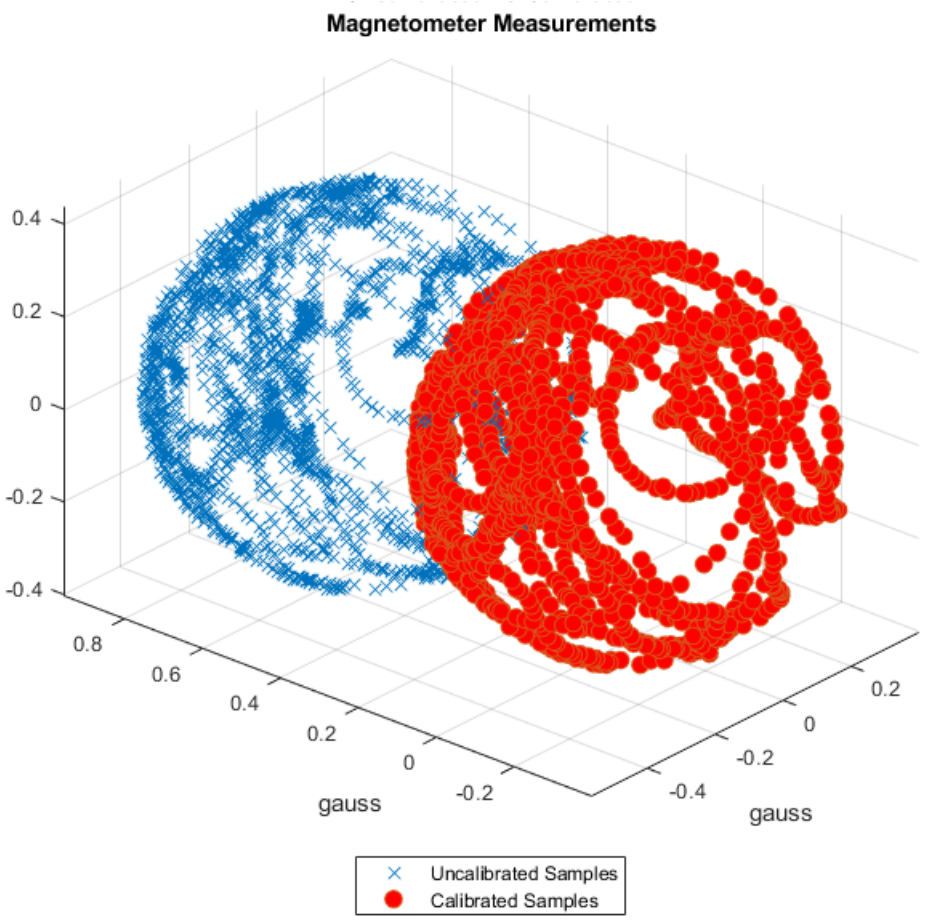


Figure 5.3: Magnetometer calibration, the two spheres origin and shape display offset error and soft iron distortion

5.2 Stationary angle measurements

The next two figures relates to the Madgwick filter feedback gain, showing how the output gets less accurate for stationary values as the filter gain increases. see [fig 5.4] and [fig 5.5].

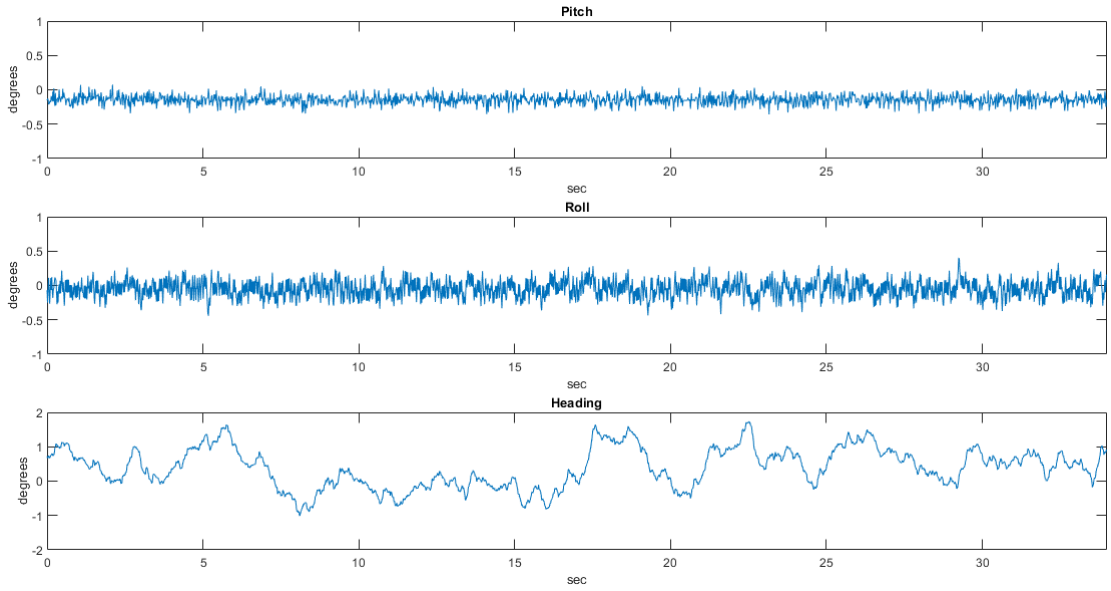


Figure 5.4: Stationary pitch / roll / heading measurements at 0.1β gain

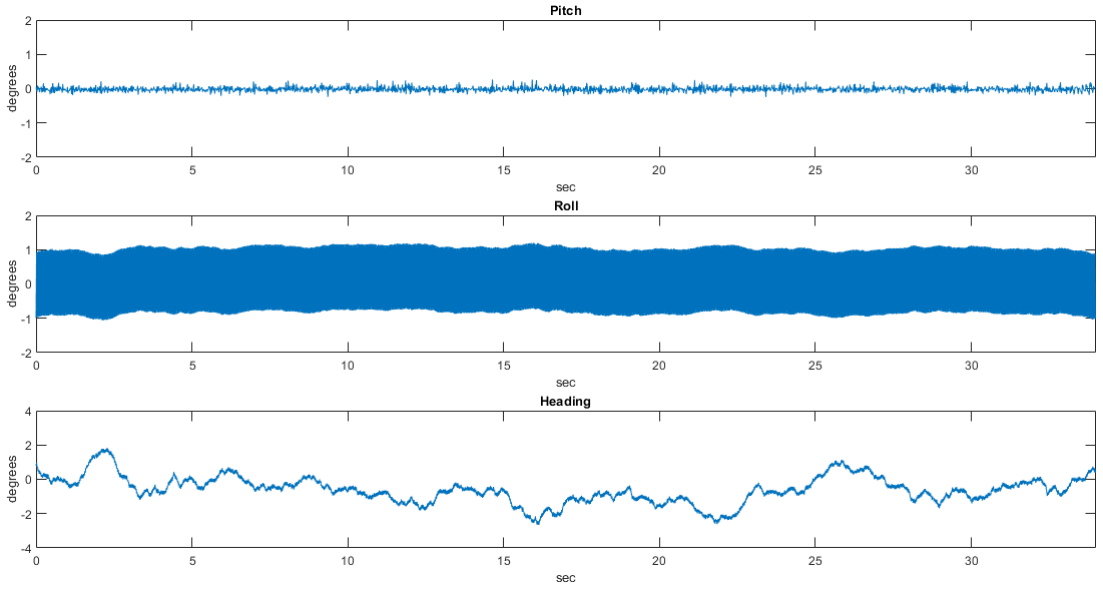


Figure 5.5: Stationary pitch / roll / heading measurements at 1β gain

5.3 Rotated angle measurements

Lastly, to show off the attitude determination capabilities, we have some graphs to show motion over time. In the first four figures the IMU starts stationary at 0° and is rotated in 90° increments until it ends up back at 0°. See [fig 5.6], [fig 5.7],

[fig 5.8] and [fig 5.9]. The last figure shows all three axes as the the device is first rotated 90° on the heading, then 90° on it's roll axis before moving both back to 0° simultaneously, see [fig 5.10]

Note: These rotations are done by hand.

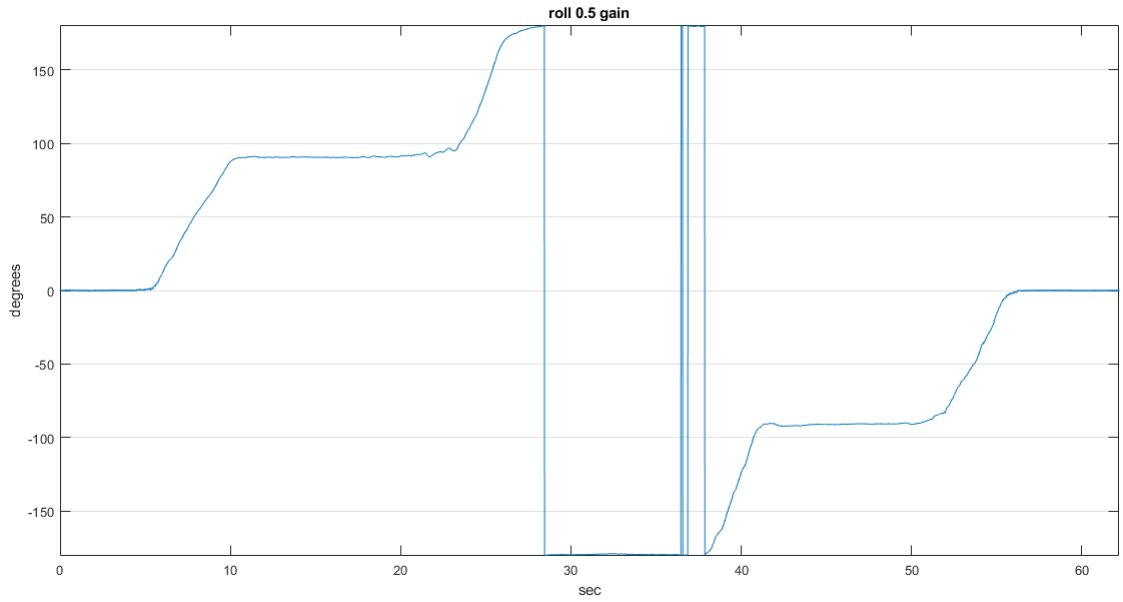


Figure 5.6: Rotated roll at 0.5β gain

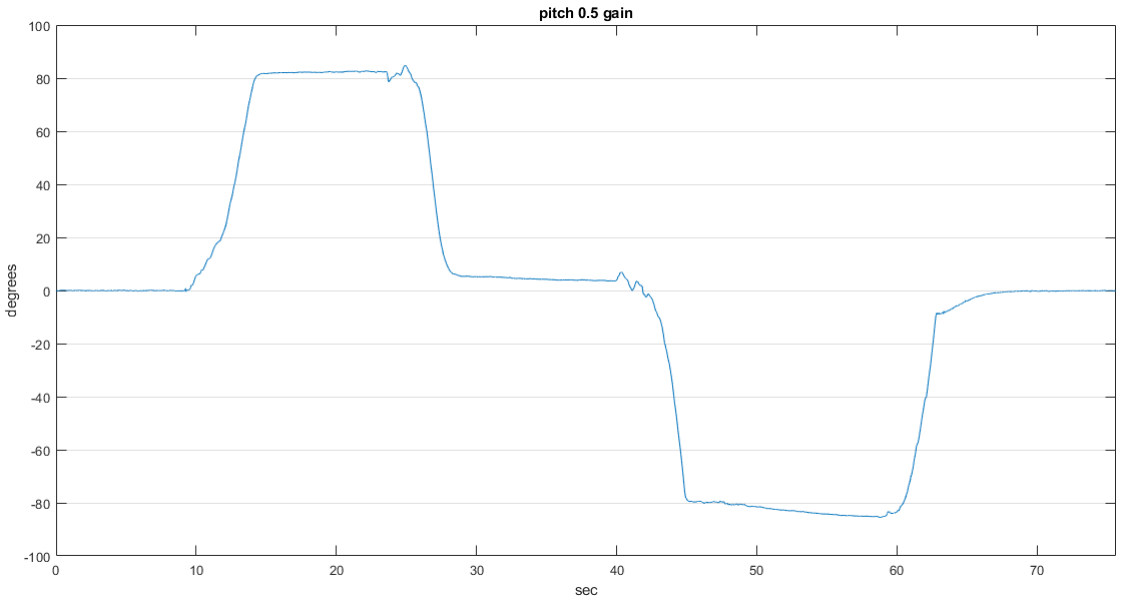


Figure 5.7: Rotated pitch at 0.5β gain

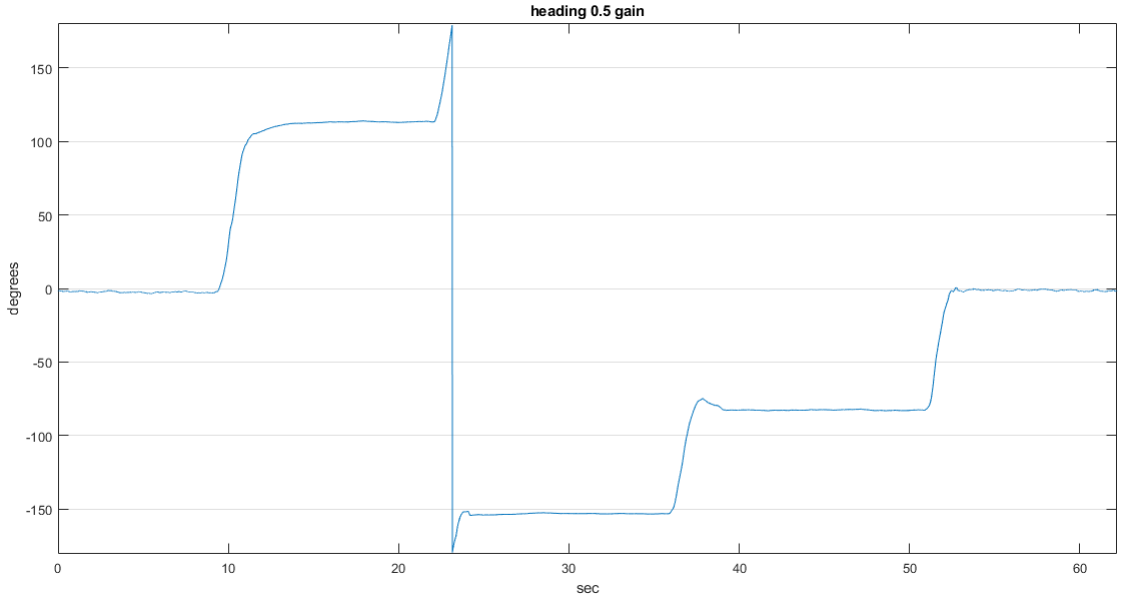


Figure 5.8: Rotated heading at 0.5β gain

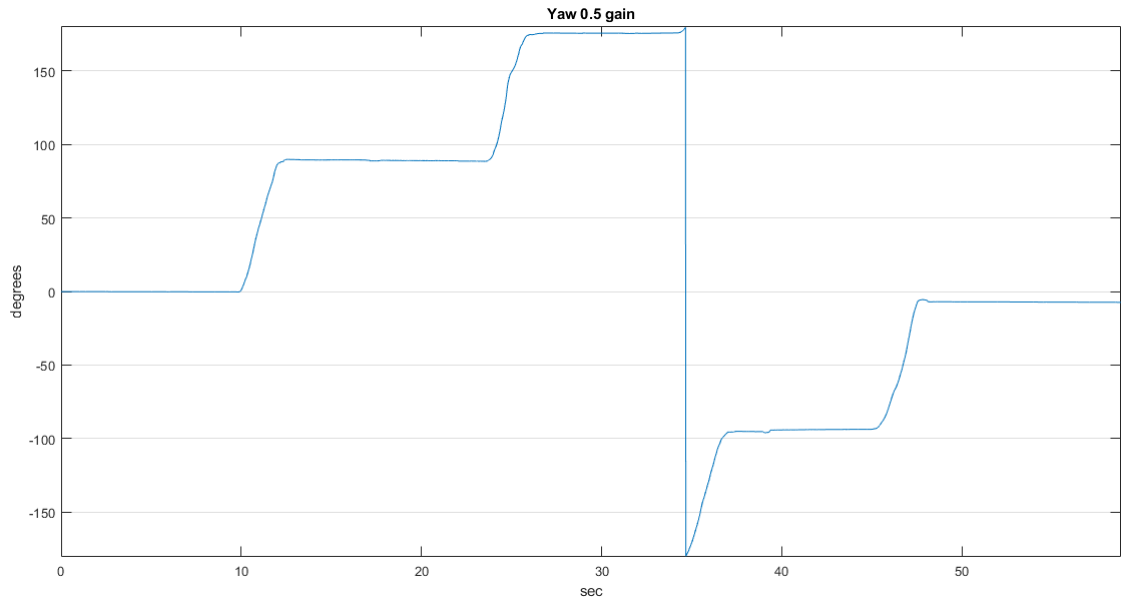


Figure 5.9: Rotated yaw (heading measured without the aide of a magnetometer) at 0.5β gain

The final graph can also be displayed in matrix form

$$\begin{bmatrix}\theta\\ \psi\\ \phi\end{bmatrix} = \begin{bmatrix}0°\\ 0°\\ 0°\end{bmatrix} \Rightarrow \begin{bmatrix}0°\\ 0°\\ -90°\end{bmatrix} \Rightarrow \begin{bmatrix}0°\\ 90°\\ -90°\end{bmatrix} \Rightarrow \begin{bmatrix}0°\\ 0°\\ 0°\end{bmatrix}$$

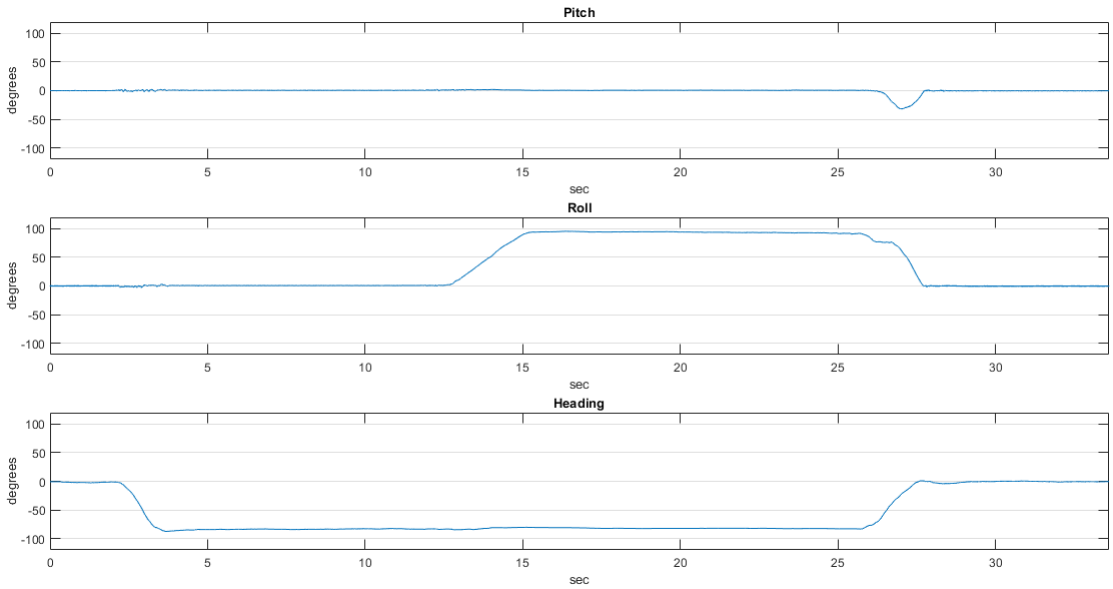


Figure 5.10: Rotation events at 1β gain

Chapter 6

Discussion

Over the course of the five months we've worked on this project we've seen it change quite, some work packages we thought we would spend a lot of time on ended up being almost entirely neglected or trivial, and some things we thought would be a breeze ended up eating way too big a chunk of our time budget. This is probably normal, the nature of the beast is that it's not easy to predict exactly what you're going to get. We would like to take some time here to reflect on the journey, how it differs from what we set out to accomplish, what we currently have in our hands, and what we would work on if we had continued.

6.1 Looking back

Going all the way back to January and early February we wrote this pre-project paper to explain the project we had ahead of us in our own words. This was our understanding of the tasks, work packages, and goals involved in the project. This was what we thought we could accomplish. In hindsight we were quite ambitious, here is a snippet from what we wrote regarding the assignment:

> This thesis will involve researching, developing and testing the needed calibration algorithms for the sensors, as well as processing the sensor measurements. Furthermore the signal processing should include basic filtering of the measurements. This would probably also constitute a noise analysis of the sensors, which could include looking at Allan variance or spectral density estimation.

We wrote this, neither sure what calibration algorithms we were going to use, nor how long it would take to figure out how to "process the sensor measurements". Well it turns out calibrating an accelerometer or gyroscope is quite easy, but calibrating a magnetometer is a lot of work, so much so that we never managed to do it in the embedded system itself, instead we had to use a function included in MATLAB. Hitting a roadblock as early as calibration is demotivating, but more importantly it makes it harder to progress on other parts of the project simultaneously. Interfacing was another part of the project we failed to predict would take so much of our time. Both computer $\iff$ Arduino and Arduino $\iff$ IMU took quite a lot of effort to get working correctly. For example, getting floating points to print properly, something that isn't really a part of the project, just a necessity for seeing the output data, took a lot of googling and digging old AVR forums from the late 2000s

to find a solution. SPI took a lot of time for other reasons. We knew how the logic worked and how to set it up, but you have to write the code exactly right to get it right. At one point we deleted the whole SPI code and rewrote it using the same logic, and that fixed our problems. Further out in the project, before deciding on a "adaptive" filter we tried our hands on manual attitude determination using the sensor outputs and some trigonometry. We found some DIY projects on level-keeping projects and self balancing robots that required the same type of data as we did, but these projects were very much done with a specific goal in mind, and the maths involved didn't hold up well outside the specifications of the DIY projects. The experimentation we went through trying to make this manual method work was great for our understanding and intuition of attitude determination, but ultimately a dead end as far as the project is concerned. When deciding on how to move on from this failure we ended up not going with the ubiquitous Extended Kalman Filter as our fusion algorithm of choice, opting instead for the Madgwick filter we found due to the ease of implementation and the extremely good documentations written for it. Another aspect of the project that was outlined but ended up not being a big focus of ours was noise analysis and sensor characteristics analysis. We started on the noise analysis by making a MATLAB script for taking the FFT of a signal, and we did run some FFT analysis on the sensor outputs. The problem is getting consistent data to analyze. Without a test bench to help us we don't really have a reliable way of making the same movement over and over. If we can't run multiple tests on the same motion we can't identify much beyond white noise, and white noise is something we already know is always present. In the main chapters we completely forego the filtering chapter and skip straight down to the Madgwick. The reason for this is simply that we didn't end up touching filter design very much while working on this project, so it felt disingenuous to write about as if that was something we got very involved in. There are low-pass filters embedded on the LSM9DS1 that are enabled by default, and we left them as they are. Because of this lack of focus on noise, Allan variance also fell to the wayside. When looking at the extra tasks we set up for ourselves we had also wanted to look into the temperature dependency of the IMU, quite a useful thing to know considering how cold space is, but sadly we don't have the testing facilities to simulate sub $-40°$ Celsius temperatures, and since the data sheet rates the IMU as working as normal down to said temperature we don't know how performance realistically deteriorates as temperatures drop further.

6.2 Where we are

As we can see in [fig 5.1], [fig 5.2] and [fig 5.3] in Chapter 5 Results, we were able to successfully calibrate the sensors. The sensors biases are removed from the gyroscope and accelerometer measurements, making the stationary values correct. By examining the magnetometer calibration plot, we can see in the shape of the calibrated data, that it's a little more spherical than the uncalibrated one. Meaning that the small part off soft iron present is accounted for. It is also visible from the plot that a large offset error is present by the location of the uncalibrated measurements origin. This offset error is in part due to magnetometer bias and in part caused by hard iron effect of nearby magnetic field sources. The ratio of bias to hard iron effect is not easy to discern, but it's also not necessary to find, as the compensation method accounts for both simultaneously. We can see that this error

is successfully accounted for as well, as the calibrated sphere of measurements has its origin located in $(x, y, z) = (0, 0, 0)$.
The Euler angles estimations plotted in [fig 5.4] and [fig 5.5] confirm that the stationary angles are relatively stable. The estimation based on a 0.1β madgwick feedback gain have noticeably less fluctuations compared to 1β feedback gain. This is expected, as the gain controls the weighing of the error correction part of the filter fusion, which can influence the performance negatively if the estimations already are close to the truth. Another observation to be made from these graphs is that all drift, commonly associated with gyroscopes, is completely eliminated by the madgwick filter. Only the heading experiences fluctuating values, stemming from magnetic disturbances that are hard to eliminate without a very closed and controlled test environment. The reason this affects heading enough to cause swing, while the other two axes remain untouched is that heading is in direct reference to magnetic north, and so the magnetometer data plays a much bigger role in determining the angle. The fact that the heading is the only angle experiencing instabilities, may point towards the possibility of magnetic noise being the culprit. Another sign supporting this idea is that when attempting to take a measurement at a different apartment we experienced wildly different magnetometer readings. Sadly we discarded these readings, and thus have no plot to back this statement up.
The rotated angle estimations portrayed in [fig 5.6], [fig 5.7] and [fig 5.8] show the performance and accuracy of the attitude determination when the unit is rotated about it's own axis in approximately 90° increments. The heading readings show uncertainties yet again. The graph misses with up to 30° in the absolute worst cases. When running the Madgwick version without the magnetometer data, the angle estimation appear to be more or less correct at all angles, until drifting occurs. Seen in [fig 5.9] which shows yaw with no magnetometer data, and follows the 90° increments more closely, but ends up settling a few degrees below zero due to drift, and this is after only a minute of run time. Assuming that the magnetometer is correctly calibrated, it seems likely that present magnetic noise inside the test environment plays a role in this fluctuation error. The roll estimation behaves as expected at all times, showing expected values at all increments. The pitch estimation only displays angle values up to around ±86°. We would expect it to reach exactly ±90° before going back towards zero. The reason our pitch can not exceed ±90° is already brought up in chapter 4.2: Euler Angles.
The final experiment demonstrates the independent nature of the different axes, the results are found in [fig 5.10]. Here we find that the Madgwick filter performs attitude determination quite well overall. The orientation changes affecting one angle at the time looked like we expected. The last orientation change where two angles were rotated simultaneously shows that this motion correctly affects the final angle, as it is physically impossible to move two axes without affecting the third.
One minor regret of ours when it comes to these graphs is our inconsistency with the high β value, it changes between 0.5, 0.6 and 1, when ideally it should be the same value across all figures. It's not a huge deal in our opinion, as we are trying to show the difference between a low and high β value, and not necessarily trying to make a statement about what value is best.

6.3 Going Forward

Looking ahead, there are many things we could imagine ourselves doing to build upon the work we've started. Some are things we feel we should have been able to do but for various reasons didn't follow through with, other things are ideas we had towards the end of the project and did not have time to implement or work on. First up is the gain on the Madgwick filter, we really should have implemented an adaptive gain functionality that goes higher if the measured angular rate exceeds a certain amount, finding the sweet spot for how fast and how much to crank the gain would take some testing, testing we didn't have time to conduct. From the tuning we did it's clear that a high gain value when the IMU is experiencing a lot of fast rotation would help the madgwick filter quickly hone in on the correct values. The gain could then be dropped down to a more sensible value once the motion slowed down. We think this could be useful specifically for the launch and eventual re-entry if the IMU is to be relied on for these two stages. Next we would want to create a makeshift test bench if no other option was available for us, one idea was re-purposing a bicycle wheel to at least get a proper consistent circular motion for our tests. Obviously ideally we'd get a proper mechanical test bench with all six degrees of freedom, but the circumstances of the Covid-19 outbreak makes it uncertain if that's a realistic possibility any time soon. With a better testing environment we could conduct more precise tests, for example maintaining a rotation of roughly one revolution per second to see how the IMU performs at what Orbit NTNU said we should consider the upper speed limit. With this we could also do FFT analysis of the gyroscope signal as it accelerates and decelerates at a consistent rate, making it easier to find the range of frequencies we would want to keep in an eventual filter design. Furthermore, if we were to continue on this project there are a few more features we'd likely add that are important for the ADCS task, first of them is on-board magnetometer calibration. Being able to run the soft and hard iron calibration algorithm from the board would reduce the time and complexity to setup the system in a new location, this functionality would be a high priority feature. Secondly we should incorporate the temperature sensor on the IMU for diagnostic purposes, being able to print out the current temperature with the measurements would be very useful if we could get access to some method of freezing the IMU down to sub -40°. Then we could start properly mapping the temperature dependency outside of normal operating ranges.
Lastly we're not very happy with how the pitch can't go beyond ±90°, and we had one idea for a potential fix that would look at the accelerometer z-axis, and determine which quadrant we should be in based on if a_z is a positive or negative value. If we imagine that the z-axis aligns with gravity and is 1 when the device is level, then the z-axis would be 0 when the device pitch is ±90° and dip into negative values as you kept rotating until it hit -1 when the pitch value reach ±180°. We had some minor success making a very quick hack, which works when looking at only the pitch, but a complete fix would require a much more thorough check of the current state of the device to account for roll, thrust, and decreased gravitational force in space. Our quick hack was this check:

```
//TEST CODE FOR PITCH
if(az < 0 && angle_pitch > 0){
    angle_pitch = 90 + (90 - angle_pitch);
```

```
}
if(az < 0 && angle_pitch < -0){
   angle_pitch = -90 - (90 + angle_pitch);
}
```

This works if you only rotate the pitch, but roll greater than $\pm 90^\circ$ would instantly break it, similarly a strong enough negative acceleration affecting the accelerometer z-axis could also cause the pitch to jump to wrong numbers. We think it could certainly be possible to make a complete workaround for the shortcomings of the pitch, but we would rather recommend working in quaternions rather than Euler angles if possible.

Chapter 7

Conclusion

In conclusion, we successfully interfaced with the Arduino and the LSM9DS1 and managed to create a calibration algorithm for the gyroscope and accelerometer. The Magnetometer calibration is a bit more involved and we had to resort to external software to pass the hurdle.
The only signal conditioning we did was a voltage level converter to let the 5V operated Arduino work properly with the 3V operated LSM9DS1.
We did not end up focusing on noise analysis or sensory characteristics except for sensitivity and range, because of this decision we also did not implement any frequency filters in our software, relying instead on the built in low-pass filter on board the IMU.
We did not make any code for fault detection or self-diagnosis of the system, nor did we look at Allan Variance. Since magnetometer calibration ended up being done in MATLAB, we didn't end up focusing on developing any on-system alternative.
We did not enable the IMU's built in temperature sensor, and we did not conduct any testing of the IMU operating outside it's recommended temperature range of -40° Celsius to 85° Celsius.
We successfully implemented a Madgwick filter instead of a Kalman filter to take care of sensory data fusion and output quaternions. We then converted these quaternions back to Euler angles to represent pitch, roll and heading. Using this we properly output attitude determination for roll, pitch works, but cannot correctly display angles greater than 90° or lower than -90°. And the heading has a non-linear offset problem.
Using PuTTY to log the data output from the IMU, and MATLAB to create plots we were able to make some useful graphs for visualizing the effects of the Madgwick filter feedback gain, effects of calibration, and overall accuracy of the attitude determination.

Bibliography

[1] Orbit NTNU, "Home," *accessed: May*, 2020. [Online]. Available: https://www.orbitntnu.com/.

[2] S. R. Starin and J. Eterno, "Attitude determination and control systems," pp. 4, 24–25, 2011. [Online]. Available: https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20110007876.pdf.

[3] E. Williams, "What could go wrong: Spi," *accessed: Apr*, 2020. [Online]. Available: https://hackaday.com/2016/07/01/what-could-go-wrong-spi/.

[4] Atmel, "Atmel atmega328p datasheet," *accessed: Jan*, 2020. [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf.

[5] SiliconSensing, "Mems accelerometers," *Accessed: May*, 2020. [Online]. Available: https://www.siliconsensing.com/technology/mems-accelerometers/.

[6] D. Nedelkovski, "Mems accelerometer gyroscope magnetometer arduino," *accessed: May*, 2020. [Online]. Available: https://howtomechatronics.com/how-it-works/electrical-engineering/mems-accelerometer-gyrocope-magnetometer-arduino/.

[7] STMicroelectronics, "Lsm9ds1 data sheet," *accessed: Feb*, 2020. [Online]. Available: https://cdn.sparkfun.com/assets/learn_tutorials/3/7/3/LSM9DS1_Datasheet.pdf.

[8] Mathworks, "Magnetometer calibration," *accessed: Mar*, 2020. [Online]. Available: https://www.mathworks.com/help/nav/ug/magnetometer-calibration.html.

[9] SBG Systems, "How to handle magnetic disturbances?" *accessed: May*, 2020. [Online]. Available: https://www.sbg-systems.com/support/knowledge/handle-magnetic-disturbances/.

[10] B. Yang, C. Li, J. Xu, and H. Sun, "Error compensation of geomagnetic field measurement used in geomagnetic navigation," in. May 2018, pp. 787–797, ISBN: 978-981-13-0028-8. DOI: 10.1007/978-981-13-0029-5_66.

[11] J. Brokking, "Mpu-6050 6dof imu for auto-leveling multicopters," *Accessed: March*, 2020. [Online]. Available: http://www.brokking.net/imu.html.

[12] S. O. H. Madgwick, "An efficient orientation filter for inertial and inertial / magnetic sensor arrays," 2010.

[13] Sparkfun, "Lsm9ds1 hookup guide," *accessed: Feb*, 2020. [Online]. Available: https://learn.sparkfun.com/tutorials/lsm9ds1-breakout-hookup-guide/all.

[14] S. Ludwig, "Optimization of control parameter for filter algorithms for attitude and heading reference systems," Jul. 2018, pp. 1–8. DOI: 10.1109/CEC.2018.8477725.

Appendix A

main.c

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include "header.h"
#include "registers.h"

// This line is needed to print floating point numbers using USART.
static FILE mystdout = FDEV_SETUP_STREAM(usart_putchar_printf, NULL,
    _FDEV_SETUP_WRITE);


int main(void)
{
   uint8_t counter = 0;
   stdout = &mystdout; // related to printf

   // gyro sample rate [Hz]: choose value between 1-6
   // 1 = 14.9   4 = 238
   // 2 = 59.5   5 = 476
   // 3 = 119    6 = 952
   gyroSampleRate = 2; // remember to set in madgwick.c as well!

   // mag scale can be 4, 8, 12, or 16
   magScale = 4;

   // Gyroscope scaling, choose: 0 = 245dps, 1 = 500dps, 3 = 2000dps
   gyroScale = 1;

   autocalc = 1; // Set autocalc enable -- This should always be on

   // Interfacing inits:
   usart_init(MYUBRR);
   spiInit();
   WhoAmICheck();  // Checks if the wires and SPI are correctly
      configured, if this check can't complete the program will not
```

```
    continue, this is to protect the IMU.

  printf("Check complete, all systems are ready to go!\n");

  // Sensor inits:
  initMag(); // Sets the magnetometer control registers, settings can be
      changed in sensorInits.c.
  initGyro(); // Sets the gyroscope control registers, settings can be
      changed in sensorInits.c
  initAccel();

  // Sensor Calibration:
  calibrateOffsetMag(-947, 3720, 175); // Sets offset, calculated in
      MATLAB function, see readme
  calibrateGyro(); // Calculates the average offset value the gyro
      measures. IMU must be held still during this.
  calibrateAccel();

  // Timer inits:
  uint16_t timerticksMax = runTime(gyroSampleRate); // Sets the runtime
      for the repeating loop.
  uint16_t TimerValue = 0;
  timerInit(); // Start the timer.

  while(1){
    counter += 1; // Counter to keep track of iteration count, this is
        for when you don't want to print out data every iteration.
    readMag();
    // convert magnetometer data to Gauss:
    mag_x = mx * SENSITIVITY_MAGNETOMETER_4; // Be sure to use the
        correct sensitivity factor.
    mag_y = my * SENSITIVITY_MAGNETOMETER_4;
    mag_z = mz * SENSITIVITY_MAGNETOMETER_4;
    // compensate for soft iron distortion using values from MATLAB:
    softIronMag(0.99847, 0.98362, 1.01898, 0.02723, 0.00005, 0.00311,
        -0.00084, -0.00821, -0.00468);
//    mag_x = 0; mag_y = 0; mag_z = 0;   // For using madgwick without
    magnetometer.

    readGyro();
    // Subtract the Offset value.
    gx -= gBiasRawX;
    gy -= gBiasRawY;
    gz -= gBiasRawZ;
    // convert gyroscope data to rad/s:
    gyro_x = gx * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));  // Be sure
        to use the correct sensitivity factor.
    gyro_y = gy * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));
    gyro_z = gz * (SENSITIVITY_GYROSCOPE_500 * (PI / 180));

    readAccel();
```

```
      // Subtract the Offset value.
      ax -= aBiasRawX;
      ay -= aBiasRawY;
      az -= aBiasRawZ;
      // convert accelerometer data to g:
      acc_x = ax * SENSITIVITY_ACCELEROMETER_8; // Be sure to use the
          correct sensitivity factor.
      acc_y = ay * SENSITIVITY_ACCELEROMETER_8;
      acc_z = az * SENSITIVITY_ACCELEROMETER_8;


      MadgwickAHRSupdate(gyro_x, gyro_y, gyro_z, acc_x, acc_y, acc_z,
          mag_x, mag_y, mag_z); // Run data through the Madgwick filter,
          this outputs the attitude as Quaternions
      QuaternionsToEuler(q0, q1, q2, q3);
          // Convert back to Euler angles.

      // convert angles from radians to degrees:
      angle_pitch *= (180/PI);
      angle_roll *= (180/PI);
      angle_yaw *= (180/PI);


      if(counter == 1){       // Edit comparison value to set how often
          you want to print out data.

         // Print Quaternions.
//       printf("q0: %f\t", q0);
//       printf("q1: %f\t", q1);
//       printf("q2: %f\t", q2);
//       printf("q3: %f\n", q3);

         // Print Euler angles.
         printf("Pitch: %f\t", angle_pitch);
         printf("Roll: %f\t", angle_roll);
         printf("Yaw:  %f\n", angle_yaw);

         // Print individual sensor outputs.
//       printf("mx: %f\t", mag_x);
//       printf("my: %f\t", mag_y);
//       printf("mz: %f\n", mag_z);
//       printf("gx: %f\t", gyro_x);
//       printf("gy: %f\t", gyro_y);
//       printf("gz: %f\n", gyro_z);
//       printf("ax: %f\t", acc_x);
//       printf("ay: %f\t", acc_y);
//       printf("az: %f\n", acc_z);

         counter = 0;          // Reset the counter.
      }
```

```
      // This code checks if the program ran fast enough to keep up with
          the Gyroscope ODR.
      if((TCNT1 > timerticksMax) | (TIFR1 & (1<<TOV1))){
         TimerValue = TCNT1;
         printf("Game over! You were too slow! \n");
         if(TIFR1 & (1<<TOV1)) printf("Timer overflow happened\r\n");
         printf("Clock cycles lapsed: %u\n", TimerValue);
         printf("Clock cycles limit: %u\n", timerticksMax);
         while(1);
      }
      while(TCNT1 < timerticksMax);  // Wait for new Gyroscope sample to
          be ready
      TCNT1 = 0x0000;             // Reset the clock and go again.
   }
}
```

Appendix B

header.h

```
/* Global Variables */

//Raw values from Gyroscope
int16_t gx;
int16_t gy;
int16_t gz;

float gyro_x;
float gyro_y;
float gyro_z;

//Raw values from Accelerometer
int16_t ax;
int16_t ay;
int16_t az;

float acc_x;
float acc_y;
float acc_z;

int32_t a_total_vector;
float angle_pitch_acc;
float angle_roll_acc;

//Raw values from Magnetometer
int16_t mx;
int16_t my;
int16_t mz;

float mag_x;
float mag_y;
float mag_z;

//Madgwick
extern volatile float beta;         // algorithm gain
extern volatile float q0, q1, q2, q3; // quaternion of sensor frame
    relative to auxiliary frame
```

```
//Euler Angles
float angle_pitch;
float angle_roll;
float angle_yaw;

//Config values
uint8_t autocalc;
uint8_t set_gyro_angles;
uint8_t magScale;     // mag scale can be 4, 8, 12, or 16
uint8_t gyroScale;       // Gyroscope scaling, choose: 0 = 245dps, 1 =
    500dps, 3 = 2000dps
uint8_t gyroSampleRate;  // choose: 1=14,9Hz 2=59,5Hz 3=119Hz 4=238Hz
    5=476Hz 6=952Hz

//Calibration Values
int16_t gBiasRawX;
int16_t gBiasRawY;
int16_t gBiasRawZ;

int16_t aBiasRawX;
int16_t aBiasRawY;
int16_t aBiasRawZ;


// Sensor Sensitivity Constants
// Values set according to the typical specifications provided in
// table 3 of the LSM9DS1 data sheet. (pg 12)
#define SENSITIVITY_ACCELEROMETER_2 0.000061
#define SENSITIVITY_ACCELEROMETER_4 0.000122
#define SENSITIVITY_ACCELEROMETER_8 0.000244
#define SENSITIVITY_ACCELEROMETER_16 0.000732
#define SENSITIVITY_GYROSCOPE_245 0.00875
#define SENSITIVITY_GYROSCOPE_500 0.0175
#define SENSITIVITY_GYROSCOPE_2000 0.07
#define SENSITIVITY_MAGNETOMETER_4 0.00014
#define SENSITIVITY_MAGNETOMETER_8 0.00029
#define SENSITIVITY_MAGNETOMETER_12 0.00043
#define SENSITIVITY_MAGNETOMETER_16 0.00058

#define BAUD 76800
#define MYUBRR F_CPU/16/BAUD-1
#define PI 3.141592

#ifndef BAUD                        /* if not defined in Makefile... */
#define BAUD 9600                  /* set a safe default baud rate */
#endif

#define PIN_XG PB1
#define PIN_M PB2
```

```
   //USART functions

void usart_init(uint16_t ubrr);

char usart_getchar(void);

void usart_putchar(char data);

void usart_pstr (char *s);

unsigned char usart_kbhit(void);

int usart_putchar_printf(char var, FILE *stream);


   //Timer functions

void timerInit(void);

uint16_t runTime(uint8_t gyroSampleRate);


   // SPI functions

/* spiInit - Initializes the SPI.

*/
void spiInit(void);

/* SPIreadByte - reads one byte of data from the desired registry via the
    hardware SPI.
   INPUTS:
      -csPin = chip select, chose between Accelerometer/Gyroscope or
          Magnetometer
      -subAdress = the desired registry to read from
*/
uint8_t SPIreadByte(uint8_t csPin, uint8_t subAddress);


/* SPIreadBytes - reads a desired amount of bytes, incrementing the
    target registry by one for every read.
   INPUTS:
      -csPin = Chip Select, chose between XL/G or M
      -subADress = The desired start address
      -dest = Destination array for storing data
      -count = desired amount of bytes to read
*/
uint8_t SPIreadBytes(uint8_t csPin, uint8_t subAddress, uint8_t * dest,
    uint8_t count);
```

```
/* SPIwriteByte - writes one byte of data to desired registry via SPI
   INPUTS:
      -csPin = chip select, chose between Accelerometer/Gyroscope or
          Magnetometer
      -subAdress = the desired registry to write to
      -data = byte to send
*/
void SPIwriteByte(uint8_t csPin, uint8_t subAddress, uint8_t data);

/* spiTransfer - Transmits one byte of data over SPI, reads one byte back.
   INPUTS:
      -data = Byte to send.
*/
uint8_t spiTransfer(uint8_t data);


   // General IMU functions

/* WhoAmICheck - Checks the WHO_AM_I registers of both magnetometer and
    Accelerometer/Gyro
            to see if they match the expected response, holds the program
               until the check passes.
*/
void WhoAmICheck(void);

/*  constrainScales - Constrains the scales of all sensors to make sure
    they're not out of spec, defaults the value to lowest valid scale.

*/
void constrainScales(void);

/* getFIFOSamples - Reads the FIFO_SRC register on the desired chip
    select.
   INPUTS:
      -csPin = chip select.
*/
uint8_t getFIFOSamples();

/* setFIFO - Set desired FIFO mode and threshold.
   INPUTS:
      -fifoMode = 3 bit value to set desired mode:
        0 = Bypass mode. FIFO is turned off. 1 = FIFO mode. Stops
            collecting data when FIFO is full
        2 = reserved.                3 = Continuous mode until trigger
            is deasserted, then FIFO mode.
        4 = Bypass mode until trigger is deasserted, then Continuous mode.
        5 = Continuous mode. If the FIFO is full, the new sample
            over-writes the older sample.
      -Threshold = ??? FIFO threshold, max value = 31.
*/
void setFIFO(uint8_t fifoMode, uint8_t fifoThs);
```

```
/* enableFIFO - enables FIFO mode
   INPUTS:
      enable - Any value other than 0 enables FIFO.
*/
void enableFIFO(uint8_t enable);

/* getGyroIntSrc() - Get status of inactivity interrupt

*/
uint8_t getInactivity(void);

/* configInactivity -
   Input:
      - duration = Inactivity duration - actual value depends on gyro ODR
      - threshold = Activity Threshold
      - sleepOn = Gyroscope operating mode during inactivity.
        1: gyroscope in sleep mode
        0: gyroscope in power-down
*/
void configInactivity(uint8_t duration, uint8_t threshold, uint8_t
    sleepOn);

/* configInt - Configure INT1 or INT2 (Gyro and Accelerometer Interrupts
    only)
   Input:
      - interrupt = Select INT1_CTRL or INT2_CTRL
        Possible values: INT1_CTRL or INT2_CTRL
      - generator = OR'd combination of interrupt generators.
        Possible values: INT_DRDY_XL, INT_DRDY_G, INT1_BOOT (INT1 only),
            INT2_DRDY_TEMP (INT2 only)
        INT_FTH, INT_OVR, INT_FSS5, INT_IG_XL (INT1 only), INT1_IG_G (INT1
            only), INT2_INACT (INT2 only) --???? work on this.
      - activeLow = Interrupt active configuration
         1: Active HIGH
         0: Active lOW
      - pushPull = Push-pull or open drain interrupt configuration --
          ???? work on this.
        Can be either INT_PUSH_PULL or INT_OPEN_DRAIN
*/
void configInt(uint8_t interrupt_select, uint8_t generator, uint8_t
    activeLow, uint8_t pushPull);


   // Gyroscope functions

/* initGyro - Initializes the Gyroscope control registers.

*/
void initGyro(void);
```

```
/* interuptGyro - Initializes the gyroscope interrupt registers.

*/
void interuptGyro(void);

/* sleepGyro - tucks the gyroscope into bed.
   INPUTS:
      enable - ANy value other than 0 enables sleep.
*/
void sleepGyro(uint8_t enable);

/* getGyroIntSrc - Reads the gyro interrupt source register

*/
uint8_t getGyroIntSrc();

/* configGyroThs() -- Configure the threshold of a gyroscope axis
   Input:
      threshold = Interrupt threshold. Possible values: 0-0x7FF.
      Value is equivalent to raw gyroscope value.
      axis = Axis to be configured. Either 1 for X, 2 for Y, or 3 for Z
      duration = Duration value must be above or below threshold to
          trigger interrupt
      wait = Wait function on duration counter
         1: Wait for duration samples before exiting interrupt
         0: Wait function off
*/
void configGyroThs(int16_t threshold, uint8_t axis, uint8_t duration,
    uint8_t wait);

/* configGyroInt() -- Configure Gyroscope Interrupt Generator
   Input:
      - generator = Interrupt axis/high-low events
         Any OR'd combination of ZHIE_G, ZLIE_G, YHIE_G, YLIE_G, XHIE_G,
             XLIE_G
      - aoi = AND/OR combination of interrupt events
         1: AND combination
         0: OR combination
      - latch: latch gyroscope interrupt request.
*/
void configGyroInt(uint8_t generator, uint8_t aoi, uint8_t latch);

/* readGyro - reads the desired gyroscope axis.
   INPUTS:
      axis_address = desired low byte address to start on, chose between:
         OUT_X_L_G
         OUT_Y_L_G
         OUT_Z_L_G
*/
void readGyro(void);
```

```
/* availableGyro = 1 when data available.
            = 0 when data not available
*/
uint8_t availableGyro(void);

/* calibrateGyro - Calibrates the gyroscope on the IMU module. Stores the
    bias in respective variables.

*/
void calibrateGyro(void);


   // Accelerometer functions

/* initAccl - Initializes the accelerometer control registers.

*/
void initAccel(void);

/* readAccel - reads the XYZ output from the accelerometer.

*/
void readAccel(void);

/* calibrateAccel - Finds and stores the offset on the Accelerometer.

*/
void calibrateAccel(void);


   // Magnetometer functions

/* initMag - Initializes the magnetometer control registers, values can
    be changed in sensorInits.c

*/
void initMag(void);

/* interruptMap - Sets up the interrupt flags for magnetometer, set
    values in sensorInits.c

*/
void interuptMag(void);

/* getGyroIntSrc() -- Get contents of magnetometer interrupt source
    register

*/
uint8_t getMagIntSrc();

/* configMagThs - Sets the magnetometer interrupt threshold
```

```
   INPUTS:
      -threshold = 16-bit value for the desired threshold.
*/
void configMagThs(uint16_t threshold);

/* configMagInt - configures the magnetometer interrupt register
   INPUTS:
      - generator = Interrupt axis/high-low events
        Any OR'd combination of ZIEN, YIEN, XIEN
      - activeLow = Interrupt active configuration
        Can be either 0 for Active LOW or 1 for active HIGH
      - latch: latch gyroscope interrupt request.
*/
void configMagInt(uint8_t generator, uint8_t activeLow, uint8_t latch);

/* readMag - Reads the XYZ values from the magnetometer, scales the raw
    value according to the set sensitivity and stores the values in mx,
    my, and mz.
   *This function depends on the variable magScale being set correctly,
      it must be either 4, 8, 12 or 16.
*/
void readMag(void);

/* availableMag = 1 when data available.
           = 0 when data not available
   INPUTS: 0 for X-axis
        1 for Y-axis
        2 for Z-axis
        3 for X, Y, and Z- axis
*/
uint8_t availableMag(uint8_t axis);

/* calibrateMag - Calibrates the magnetometer by storing the offset in
    the registers present on the IMU.
   INPUTS:    raw offset found by calmag() function in Matlab
*/
void calibrateOffsetMag(int16_t offsetX, int16_t offsetY, int16_t
    offsetZ);

/* OffsetMag - set the offset of the magnetometer, this function is
    called in calibrateMag
   INPUTS:    inputs are selected in another function, should not be
      touched.
*/
void offsetMag(uint8_t axis, int16_t offset);

/* softIronMag - compensates for soft iron distortion using values
    calculated in MATLAB function
   INPUTS:    A: 3x3 matrix consisting of x-, y- and z-values
           b: 3x1 matrix consisting of b-values
*/
```

```
void softIronMag(float xx, float yy, float zz, float xy, float xz, float
    yz, float b1, float b2, float b3);

   // Madgwick filter

/* MadgwickAHRSupdate - updates orientation in quaternions (q0, q1, q2,
    q3)
   INPUTS:    gyroscope, accelerometer and magnetometer data (float)
           gyroscope data needs to be in rad/s, rest is your choice
*/
void MadgwickAHRSupdate(float gx, float gy, float gz, float ax, float ay,
    float az, float mx, float my, float mz);

/* MadgwickAHRSupdateIMU - called in MadgwickAHRSupdate if no
    magnetometer data is available
   INPUTS:    gyroscope and accelerometer data (float)
*/
void MadgwickAHRSupdateIMU(float gx, float gy, float gz, float ax, float
    ay, float az);

/* QuaternionsToEuler - converts quaternions to Euler angles
   INPUTS:    quaternions (q0, q1, q2, q3)
*/
void QuaternionsToEuler(volatile float q0, volatile float q1, volatile
    float q2, volatile float q3);
```

Appendix C

functions.c

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include "header.h"
#include "registers.h"

   // USART

void usart_init( uint16_t ubrr) {
   // Set baud rate
   UBRR0H = (uint8_t)(ubrr>>8);
   UBRR0L = (uint8_t)ubrr;
   // Enable receiver and transmitter
   UCSR0B = (1<<RXEN0)|(1<<TXEN0);
   // Set frame format: 8data, 1stop bit
   UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);

}

void usart_putchar(char putchar) {
   // Wait for empty transmit buffer
   while ( !(UCSR0A & (1<<UDRE0)));
   // Start transmission
   UDR0 = putchar;
}

char usart_getchar(void) {
   // Wait for incoming data
   while ( !(UCSR0A & (1<<RXC0)));
   // Return the data
   return UDR0;
}

void usart_pstr(char *s) {
   // loop through entire string
```

```
   while (*s) {
      usart_putchar(*s);
      s++;
   }
}

unsigned char kbhit(void) {
   //return nonzero if char waiting polled version
   unsigned char b;
   b=0;
   if(UCSR0A & (1<<RXC0)) b=1;
   return b;
}

//*********** required for printf *********//
int usart_putchar_printf(char var, FILE *stream) {
   if (var == '\n') usart_putchar('\r');
   usart_putchar(var);
   return 0;

}


   // Timer

void timerInit(void){
   TCCR1A = 0x00;          // We don't need to set any bits, we will use
       normal mode.
   TCCR1B = (1<<CS11)|(1<<CS10); // Clock Divide 64 on prescaler
}

uint16_t runTime(uint8_t gyroSampleRate){
   // Values need to get adjusted if the Clock Divide prescaler is changed
   uint16_t temp = 0;
   switch (gyroSampleRate){
      case 1:
      temp = 16779;
      break;
      case 2:
      temp = 4202;
      break;
      case 3:
      temp = 2101;
      break;
      case 4:
      temp = 1051;
      break;
      case 5:
      temp = 526;
      break;
      case 6:
```

```
      temp = 263;
      break;
   }
   return temp;
}

   // SPI

 void spiInit(void){
    DDRB = (1<<DDB5)|(1<<DDB3)|(1<<DDB2)|(1<<DDB1);              // MOSI,
        SCK, CS_M and CS_AG output
    PORTB = (1<<PORTB2)|(1<<PORTB1);                            // CS_M and
        CS_AG start HIGH
    SPCR = (1<<SPE)|(1<<MSTR)|(1<<CPOL)|(1<<CPHA)|(1<<SPR1)|(1<<SPR0);
        // * SPI enable, Master mode, MSB first, Clock div 128.
    // * Clock idle HIGH, Data Captured on Rising edge. SPI mode 3.
 }

uint8_t SPIreadByte(uint8_t csPin, uint8_t subAddress)
{
   uint8_t temp;
   // Use the multiple read function to read 1 byte.
   // Value is returned to `temp`.
   SPIreadBytes(csPin, subAddress, &temp, 1);
   return temp;
}

uint8_t SPIreadBytes(uint8_t csPin, uint8_t subAddress, uint8_t * dest,
    uint8_t count)
{
   // To indicate a read, set bit 0 (MSB) of first byte to 1
   uint8_t rAddress = 0x80 | (subAddress & 0x3F);
   // Mag SPI port is different. If we're reading multiple bytes,
   // set bit 1 to 1. The remaining six bytes are the address to be read
   if ((csPin == PIN_M) && count > 1)
   rAddress |= 0x40;

   PORTB &= ~(1<<csPin); // Initiate communication
   spiTransfer(rAddress);
   for (int i=0; i<count; i++)
   {
      dest[i] = spiTransfer(0x00); // Read into destination array
   }
   PORTB |= (1<<csPin); // Close communication

   return count;
}

uint8_t spiTransfer(uint8_t data) {
    SPDR = data;
    /*
```

```
     * The following NOP introduces a small delay that can prevent the
        wait
     * loop form iterating when running at the maximum speed. This gives
     * about 10% more speed, even if it seems counter-intuitive. At lower
     * speeds it is unnoticed. // Ref. Arduino SPI.C
     */
    asm volatile("nop");
    while (!(SPSR & (1<<SPIF))) ; // wait
    return SPDR;
  }

void SPIwriteByte(uint8_t csPin, uint8_t subAddress, uint8_t data)
   {
      PORTB &= ~(1<<csPin); // Initiate communication

      // If write, bit 0 (MSB) should be 0
      spiTransfer(subAddress & 0x3F); // Send Address
      spiTransfer(data); // Send data

      PORTB |= (1<<csPin); // Close communication
   }


   // General

uint8_t getFIFOSamples(void){
   return (SPIreadByte(PIN_XG, FIFO_SRC) & 0x3F);
}

void setFIFO(uint8_t fifoMode, uint8_t fifoThs){
   // Limit threshold - 0x1F (31) is the maximum. If more than that was
      asked
   // limit it to the maximum.
   uint8_t threshold = fifoThs <= 0x1F ? fifoThs : 0x1F;
   SPIwriteByte(PIN_XG, FIFO_CTRL, ((fifoMode & 0x7) << 5) | (threshold &
      0x1F));
}

void enableFIFO(uint8_t enable){
   uint8_t temp = SPIreadByte(PIN_XG, CTRL_REG9);
   if (enable) temp |= (1<<1);
   else temp &= ~(1<<1);
   SPIwriteByte(PIN_XG, CTRL_REG9, temp);
}

uint8_t getInactivity(void){
   uint8_t temp = SPIreadByte(PIN_XG, STATUS_REG_0);
   temp &= (0x10);
   return temp;
}
```

```
void configInactivity(uint8_t duration, uint8_t threshold, uint8_t
    sleepOn){
   uint8_t temp = 0;

   temp = threshold & 0x7F;
   if (sleepOn) temp |= (1<<7);
   SPIwriteByte(PIN_XG, ACT_THS, temp);

   SPIwriteByte(PIN_XG, ACT_DUR, duration);
}

void configInt(uint8_t interrupt_select, uint8_t generator, uint8_t
    activeLow, uint8_t pushPull){
   // Write to INT1_CTRL or INT2_CTRL. [interrupt] should already be one
      of
   // those two values.
   // [generator] should be an OR'd list of values from the
      interrupt_generators enum
   SPIwriteByte(PIN_XG, interrupt_select, generator);

   // Configure CTRL_REG8
   uint8_t temp;
   temp = SPIreadByte(PIN_XG, CTRL_REG8);

   if (activeLow) temp |= (1<<5);
   else temp &= ~(1<<5);

   if (pushPull) temp &= ~(1<<4);
   else temp |= (1<<4);

   SPIwriteByte(PIN_XG, CTRL_REG8, temp);
}

void WhoAmICheck(void){
     uint8_t testM = 0x00;
     uint8_t testXG = 0x00;
     uint16_t whoAmICombined = 0x0000;
     testXG = SPIreadByte(PIN_XG, WHO_AM_I_XG);
     testM = SPIreadByte(PIN_M, WHO_AM_I_M);
     printf("Conducting WHO_AM_I check, please wait...\n");
     whoAmICombined = (testXG << 8) | testM;
     printf("WHO_AM_I reads 0x%X, expected 0x683D\n", whoAmICombined);
     if(whoAmICombined != ((WHO_AM_I_AG_RSP << 8) | WHO_AM_I_M_RSP)){
        printf("test failed. \n");
        printf("double-check wiring and retry, program will not run as
            long as the check fails.\n");
        while(1);
     }
  }
```

```
   // Gyroscope

void sleepGyro(uint8_t enable){
   uint8_t temp = SPIreadByte(PIN_XG, CTRL_REG9);
   if (enable) temp |= (1<<6);
   else temp &= ~(1<<6);
   SPIwriteByte(PIN_XG, CTRL_REG9, temp);
}

void readGyro(void){
   uint8_t temp[6]; // We'll read six bytes from the mag into temp
   SPIreadBytes(PIN_XG, OUT_X_L_G, temp, 6);
   gx = (temp[1] << 8 | temp[0]);
   gy = (temp[3] << 8 | temp[2]);
   gz = (temp[5] << 8 | temp[4]);
}

uint8_t getGyroIntSrc(void){
   uint8_t intSrc = SPIreadByte(PIN_XG, INT_GEN_SRC_G);

   // Check if the IA_G (interrupt active) bit is set
   if (intSrc & (1<<6))
   {
      return (intSrc & 0x3F);
   }

   return 0;
}

void configGyroInt(uint8_t generator, uint8_t aoi, uint8_t latch){
   // Use variables from accel_interrupt_generator, OR'd together to
       create
   // the [generator]value.
   uint8_t temp = generator;
   if (aoi) temp |= 0x80;
   if (latch) temp |= 0x40;
   SPIwriteByte(PIN_XG, INT_GEN_CFG_G, temp);
}

void configGyroThs(int16_t threshold, uint8_t axis, uint8_t duration,
    uint8_t wait){
   uint8_t buffer[2];
   buffer[0] = (threshold & 0x7F00) >> 8;
   buffer[1] = (threshold & 0x00FF);
   // Write threshold value to INT_GEN_THS_?H_G and INT_GEN_THS_?L_G.
   // axis will be 0, 1, or 2 (x, y, z respectively)
   SPIwriteByte(PIN_XG, INT_GEN_THS_XH_G + (axis * 2), buffer[0]);
   SPIwriteByte(PIN_XG, INT_GEN_THS_XH_G + 1 + (axis * 2), buffer[1]);

   // Write duration and wait to INT_GEN_DUR_XL
   uint8_t temp;
```

```
   temp = (duration & 0x7F);
   if (wait) temp |= 0x80;
   SPIwriteByte(PIN_XG, INT_GEN_DUR_G, temp);
}

uint8_t availableGyro(void){
   uint8_t status = SPIreadByte(PIN_XG, STATUS_REG_1);
   return ((status & 0b00000010) >> 1);
}

   // Accelerometer

void readAccel(void){
   uint8_t temp[6]; // We'll read six bytes from the mag into temp
   SPIreadBytes(PIN_XG, OUT_X_L_XL, temp, 6);
   ax = (temp[1] << 8 | temp[0]);
   ay = (temp[3] << 8 | temp[2]);
   az = (temp[5] << 8 | temp[4]);
}

   // Magnetometer

uint8_t getMagIntSrc(void){
   uint8_t intSrc = SPIreadByte(PIN_M, INT_SRC_M);

   // Check if the INT (interrupt active) bit is set
   if (intSrc & (1<<0)){
      return (intSrc & 0xFE);
   }

   return 0;
}

void configMagThs(uint16_t threshold){
   // Write high eight bits of [threshold] to INT_THS_H_M
   SPIwriteByte(PIN_M, INT_THS_H_M, ((threshold & 0x7F00) >> 8));
   // Write low eight bits of [threshold] to INT_THS_L_M
   SPIwriteByte(PIN_M, INT_THS_L_M, (threshold & 0x00FF));
}

void configMagInt(uint8_t generator, uint8_t activeLow, uint8_t latch){
   // Mask out non-generator bits (0-4)
   uint8_t config = (generator & 0xE0);
   // IEA bit is 0 for active-low, 1 for active-high.
   if (activeLow == 0) config |= (1<<2);
   // IEL bit is 0 for latched, 1 for not-latched
   if (!latch) config |= (1<<1);
   // As long as we have at least 1 generator, enable the interrupt
   if (generator != 0) config |= (1<<0);

   SPIwriteByte(PIN_M, INT_CFG_M, config);
```

```
}

void readMag(void){
   uint8_t temp[6]; // We'll read six bytes from the mag into temp
   SPIreadBytes(PIN_M, OUT_X_L_M, temp, 6);
   mx = (temp[1] << 8 | temp[0]);
   my = (temp[3] << 8 | temp[2]);
   mz = (temp[5] << 8 | temp[4]);
}

uint8_t availableMag(uint8_t axis){
   uint8_t status = SPIreadByte(PIN_M, STATUS_REG_M);
   return ((status & (1<<axis)) >> axis);
}
```

Appendix D

sensorInits.c

```
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include "header.h"
#include "registers.h"



/* ---------------------- GYROSCOPE -------------------- */

void initGyro(void){

   uint8_t gyroEnableX = 1; // 0 for off, 1 for on
   uint8_t gyroEnableZ = 1; // 0 for off, 1 for on
   uint8_t gyroEnableY = 1; // 0 for off, 1 for on

   // bandwidth is dependent on scaling, choose value between 0-3
   uint8_t gyroBandwidth = 0;
   uint8_t gyroLowPowerEnable = 0; // 0 for off, 1 for on
   uint8_t gyroHPFEnable = 0; // 0 for off, 1 for on
   // HPF cutoff frequency depends on sample rate
   // choose value between 0-9
   uint8_t gyroHPFCutoff = 0;
   uint8_t gyroFlipX = 0; // 0 for default, 1 for orientation inverted
   uint8_t gyroFlipZ = 0; // ----
   uint8_t gyroFlipY = 0; // ----
   uint8_t gyroOrientation = 0b00000000; // 3-bit value
   uint8_t gyroLatchINT = 0; // 0 for off, 1 for on

   uint8_t tempValue = 0;

   /* CTRL_REG1_G
      bit 7-5: output data rate selection, activates gyroscope when
          written
      bit 4-3: full-scale selection
```

```
   bit 2:     always 0, don't write
   bit 1-0: bandwidth selection */
// Default: CTRL_REG1_G = 0x00;
tempValue = (gyroSampleRate & 0b00000111) << 5;
tempValue |= (gyroScale & 0b00000011) << 3;
tempValue |= (gyroBandwidth & 0b00000011);
SPIwriteByte(PIN_XG, CTRL_REG1_G, tempValue);

/* CTRL_REG2_G
   bit 7-4: always 0
   bit 3-2: INT (interrupt) selection configuration
   bit 1-0: OUT selection configuration */
// Default: CTRL_REG2_G = 0x00;
SPIwriteByte(PIN_XG, CTRL_REG2_G, 0x00);

/* CTRL_REG3_G
   bit 7:     low power mode enable
   bit 6:     high-pass filter enable
   bit 5-4: always 0
   bit 3-0: high-pass filter cutoff frequency selection */
// Default: CTRL_REG3_G = 0x00;
tempValue = 0;
tempValue = (gyroLowPowerEnable & 0x01) << 7;
tempValue |= (gyroHPFEnable & 0x01) << 6;
tempValue |= (gyroHPFCutoff & 0b0000111);
SPIwriteByte(PIN_XG, CTRL_REG3_G, tempValue);

/* CTRL_REG4
   bit 5:     yaw axis (Z) output enable
   bit 4:     roll axis (Y) output enable
   bit 3:     pitch axis (X) output enable
   bit 1:     latched interrupt
   bit 0:     4D option on interrupt
   rest:   always 0 */
// Default: CTRL_REG4 = 0x00;
tempValue = 0;
tempValue = (gyroEnableZ & 0x01) << 5;
tempValue |= (gyroEnableY & 0x01) << 4;
tempValue |= (gyroEnableX & 0x01) << 3;
tempValue |= (gyroLatchINT & 0x01) << 1;
SPIwriteByte(PIN_XG, CTRL_REG4, tempValue);

/* ORIENT_CFG_G
   bit 7-6: always 0
   bit 5:     pitch axis (X) angular rate sign
   bit 4:     roll axis (Y) angular rate sign
   bit 3:     yaw axis (Z) angular rate sign
   bit 2-0: directional user orientation selection */
// default = ORIENT_CFG_G = 0x00;
tempValue = 0;
tempValue = (gyroFlipX & 0x01) << 5;
```

```
   tempValue |= (gyroFlipY & 0x01) << 4;
   tempValue |= (gyroFlipZ & 0x01) << 3;
   tempValue |= (gyroOrientation & 0b00000111);
   SPIwriteByte(PIN_XG, ORIENT_CFG_G, tempValue);
}

void calibrateGyro(void){
   // int32_t aBiasRawTemp[3] = {0, 0, 0}; //Not yet implemented
   int32_t gBiasRawTemp[3] = {0, 0, 0};

   printf("Calibrating gyroscope, hold the device still\n");
   for(int i = 0; i<1000; i++){
      if(i % 100 == 0)printf(".");
      readGyro();
      gBiasRawTemp[0] += gx;
      gBiasRawTemp[1] += gy;
      gBiasRawTemp[2] += gz;
      _delay_ms(9); // Wait for guaranteed new data.
   }
   printf("\n");
   gBiasRawX = gBiasRawTemp[0] / 1000;
   gBiasRawY = gBiasRawTemp[1] / 1000;
   gBiasRawZ = gBiasRawTemp[2] / 1000;

}

/* ---------------------- MAGNETOMETER -------------------- */


void initMag(void){

   uint8_t tempRegValue = 0x00;

// uint8_t magEnable = 0x01;

   // mag data rate can be 0-7
   // 0 = 0.625 Hz 4 = 10 Hz
   // 1 = 1.25 Hz  5 = 20 Hz
   // 2 = 2.5 Hz   6 = 40 Hz
   // 3 = 5 Hz     7 = 80 Hz
   uint8_t magSampleRate = 7;
   uint8_t magTempCompensationEnable = 0x00;
   // magPerformance can be any value between 0-3
   // 0 = Low power mode    2 = high performance
   // 1 = medium performance 3 = ultra-high performance
   uint8_t magXYPerformance = 3;
   uint8_t magZPerformance = 3;
   uint8_t magLowPowerEnable = 0x00;
   // magOperatingMode can be 0-2
   // 0 = continuous conversion
   // 1 = single-conversion
```

```
// 2 = power down
uint8_t magOperatingMode = 0;

/* CTRL_REG1_M
   bit 7:     temperature compensation enable
   bit 6-5: X and Y axis operation mode selection
   bit 4-2: output data rate selection
   bit 1:     enables higher data rates than 80Hz
   bit 0:     self-test enable */
// Default CTRL_REG1_M = 0x10
if(magTempCompensationEnable){tempRegValue = (1<<7);}
tempRegValue |= (magXYPerformance & 0x3) << 5;
tempRegValue |= (magSampleRate & 0x7) << 2;
SPIwriteByte(PIN_M, CTRL_REG1_M, tempRegValue);

/* CTRL_REG2_M
   bit 6-5: full scale configuration
   bit 3:     reboot memory content
   bit 2:     configuration registers and user register reset function
   rest:   always 0 */
// Default CTRL_REG2_M = 0x00
tempRegValue = 0;
switch (magScale){
   case 8:
   tempRegValue |= (0x1 << 5);
   break;
   case 12:
   tempRegValue |= (0x2 << 5);
   break;
   case 16:
   tempRegValue |= (0x3 << 5);
   break;
   // Otherwise we'll default to 4 gauss (00)
}
SPIwriteByte(PIN_M, CTRL_REG2_M, tempRegValue);

/* CTRL_REG3_M
   bit 7:     I2C disable
   bit 5:     low-power mode configuration
   bit 2:     SPI mode selection
   bit 1-0: operation mode selection
   rest:   always 0 */
// Default CTRL_REG3_M = 0x03
tempRegValue = 0;
if(magLowPowerEnable){ tempRegValue = (1<<5);}
tempRegValue |= (magOperatingMode & 0x3);
SPIwriteByte(PIN_M, CTRL_REG3_M, tempRegValue);

/* CTRL_REG4_M
   bit 3-2: Z-axis operation mode selection
     00:low-power mode, 01:medium performance
```

```
        10:high performance, 10:ultra-high performance
      bit 1:     Endian data selection
      rest:    always 0 */
   // Default CTRL_REG4_M = 0x00
   tempRegValue = 0;
   tempRegValue = (magZPerformance & 0x3) << 2;
   SPIwriteByte(PIN_M, CTRL_REG4_M, tempRegValue);

   /* CTRL_REG5_M
      bit 7:  fast read enable
      bit 6:  block data update for magnetic data
      rest: always 0 */
   // Default CTRL_REG5_M = 0x00
   tempRegValue = 0;
   SPIwriteByte(PIN_M, CTRL_REG5_M, tempRegValue);
}


void calibrateOffsetMag(int16_t offsetX, int16_t offsetY, int16_t
    offsetZ){
   // hard iron distortion:
   int16_t mBiasRaw[3] = {offsetX, offsetY, offsetZ};
   int j;
   for (j=0; j<3; j++){
      offsetMag(j, mBiasRaw[j]);
   }
}

void offsetMag(uint8_t axis, int16_t offset){
   /* Offset values. This values acts on the magnetic output data values
      in order to subtract the environmental offset */
   if (axis > 2) return; // make sure we don't write to wrong address
   uint8_t msb = (offset & 0xFF00) >> 8;
   uint8_t lsb = offset & 0x00FF;
   SPIwriteByte(PIN_M, OFFSET_X_REG_L_M + (2 * axis), lsb);
   SPIwriteByte(PIN_M, OFFSET_X_REG_H_M + (2 * axis), msb);
}


void softIronMag(float xx, float yy, float zz, float xy, float xz, float
    yz, float b1, float b2, float b3){
   // A = [xx,yx,zx ; xy,yy,zy ; xz,yz,zz]  order does not matter (xy =
       yx)
   // b = [b1,b2,b3]
   // MagCalib = (Mag - b) * A
   float m_x = mag_x - b1;    // subtracts remaining offset error
   float m_y = mag_y - b2;
   float m_z = mag_z - b3;
   mag_x = m_x*xx + m_y*xy + m_z*xz;
   mag_y = m_x*xy + m_y*yy + m_z*yz;
   mag_z = m_x*xz + m_y*yz + m_z*zz;
```

```
}


/* --------------------- ACCELEROMETER -------------------- */

void initAccel(void){
   //test code only for debugging, fix this later.
   SPIwriteByte(PIN_XG, CTRL_REG6_XL, 0b00011000); // set Accel scale to
       +-8 g.
}

void calibrateAccel(void){
   int32_t aBiasRawTemp[3] = {0, 0, 0};

   printf("Calibrating Accelerometer, please put the device on a stable,
       level surface.\n");
   for(int i = 0; i<1000; i++){
      if(i % 100 == 0)printf(".");
      readAccel();
      aBiasRawTemp[0] += ax;
      aBiasRawTemp[1] += ay;
      aBiasRawTemp[2] += az - (int16_t)(1./SENSITIVITY_ACCELEROMETER_8);
      _delay_ms(9); // Wait for guaranteed new data.
   }
   printf("\n");
   aBiasRawX = aBiasRawTemp[0] / 1000;
   aBiasRawY = aBiasRawTemp[1] / 1000;
   aBiasRawZ = aBiasRawTemp[2] / 1000;

}
```

Appendix E

Madgwick.c

```
//=====================================================================
// MadgwickAHRS.c
//=====================================================================
//
// Implementation of Madgwick's IMU and AHRS algorithms.
// See: http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms
//
// Date       Author         Notes
// 29/09/2011 SOH Madgwick  Initial release
// 02/10/2011 SOH Madgwick Optimized for reduced CPU load
// 19/02/2012 SOH Madgwick Magnetometer measurement is normalized
//
//=====================================================================

//---------------------------------------------------------------------------------------------------
// Header files
#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include "header.h"
#include "registers.h"
//---------------------------------------------------------------------------------------------------
// Definitions

#define sampleFreq  59.5f  // sample frequency in Hz
#define betaDef    0.1f  // 2 * proportional gain

//---------------------------------------------------------------------------------------------------
// Variable definitions

// beta should be: sqrt(3/4) * estimated mean zero gyroscope measurement
    error
volatile float beta = betaDef;                    // 2 * proportional gain
volatile float q0 = 1.0f, q1 = 0.0f, q2 = 0.0f, q3 = 0.0f; // quaternion
    of sensor frame relative to auxiliary frame
```

```
//---------------------------------------------------------------------------------------------------
// Function declarations

float invSqrt(float x);

//====================================================================================================
// Functions

//---------------------------------------------------------------------------------------------------
// AHRS algorithm update

void MadgwickAHRSupdate(float gx, float gy, float gz, float ax, float ay,
    float az, float mx, float my, float mz) {
   float recipNorm;
   float s0, s1, s2, s3;
   float qDot1, qDot2, qDot3, qDot4;
   float hx, hy;
   float _2q0mx, _2q0my, _2q0mz, _2q1mx, _2bx, _2bz, _4bx, _4bz, _2q0,
       _2q1, _2q2, _2q3, _2q0q2, _2q2q3, q0q0, q0q1, q0q2, q0q3, q1q1,
       q1q2, q1q3, q2q2, q2q3, q3q3;

   // Use IMU algorithm if magnetometer measurement invalid (avoids NaN
       in magnetometer normalization)
   if((mx == 0.0f) && (my == 0.0f) && (mz == 0.0f)) {
      MadgwickAHRSupdateIMU(gx, gy, gz, ax, ay, az);
      return;
   }

   // Rate of change of quaternion from gyroscope
   qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
   qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
   qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
   qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);

   // Compute feedback only if accelerometer measurement valid (avoids
       NaN in accelerometer normalization)
   if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

      // Normalize accelerometer measurement
      recipNorm = invSqrt(ax * ax + ay * ay + az * az);
      ax *= recipNorm;
      ay *= recipNorm;
      az *= recipNorm;

      // Normalize magnetometer measurement
      recipNorm = invSqrt(mx * mx + my * my + mz * mz);
      mx *= recipNorm;
      my *= recipNorm;
      mz *= recipNorm;
```

```
// Auxiliary variables to avoid repeated arithmetic
_2q0mx = 2.0f * q0 * mx;
_2q0my = 2.0f * q0 * my;
_2q0mz = 2.0f * q0 * mz;
_2q1mx = 2.0f * q1 * mx;
_2q0 = 2.0f * q0;
_2q1 = 2.0f * q1;
_2q2 = 2.0f * q2;
_2q3 = 2.0f * q3;
_2q0q2 = 2.0f * q0 * q2;
_2q2q3 = 2.0f * q2 * q3;
q0q0 = q0 * q0;
q0q1 = q0 * q1;
q0q2 = q0 * q2;
q0q3 = q0 * q3;
q1q1 = q1 * q1;
q1q2 = q1 * q2;
q1q3 = q1 * q3;
q2q2 = q2 * q2;
q2q3 = q2 * q3;
q3q3 = q3 * q3;

// Reference direction of Earth's magnetic field
hx = mx * q0q0 - _2q0my * q3 + _2q0mz * q2 + mx * q1q1 + _2q1 * my
    * q2 + _2q1 * mz * q3 - mx * q2q2 - mx * q3q3;
hy = _2q0mx * q3 + my * q0q0 - _2q0mz * q1 + _2q1mx * q2 - my *
    q1q1 + my * q2q2 + _2q2 * mz * q3 - my * q3q3;
_2bx = sqrt(hx * hx + hy * hy);
_2bz = -_2q0mx * q2 + _2q0my * q1 + mz * q0q0 + _2q1mx * q3 - mz *
    q1q1 + _2q2 * my * q3 - mz * q2q2 + mz * q3q3;
_4bx = 2.0f * _2bx;
_4bz = 2.0f * _2bz;

// Gradient decent algorithm corrective step
s0 = -_2q2 * (2.0f * q1q3 - _2q0q2 - ax) + _2q1 * (2.0f * q0q1 +
    _2q2q3 - ay) - _2bz * q2 * (_2bx * (0.5f - q2q2 - q3q3) + _2bz *
    (q1q3 - q0q2) - mx) + (-_2bx * q3 + _2bz * q1) * (_2bx * (q1q2 -
    q0q3) + _2bz * (q0q1 + q2q3) - my) + _2bx * q2 * (_2bx * (q0q2 +
    q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
s1 = _2q3 * (2.0f * q1q3 - _2q0q2 - ax) + _2q0 * (2.0f * q0q1 +
    _2q2q3 - ay) - 4.0f * q1 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az)
    + _2bz * q3 * (_2bx * (0.5f - q2q2 - q3q3) + _2bz * (q1q3 -
    q0q2) - mx) + (_2bx * q2 + _2bz * q0) * (_2bx * (q1q2 - q0q3) +
    _2bz * (q0q1 + q2q3) - my) + (_2bx * q3 - _4bz * q1) * (_2bx *
    (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
s2 = -_2q0 * (2.0f * q1q3 - _2q0q2 - ax) + _2q3 * (2.0f * q0q1 +
    _2q2q3 - ay) - 4.0f * q2 * (1 - 2.0f * q1q1 - 2.0f * q2q2 - az)
    + (-_4bx * q2 - _2bz * q0) * (_2bx * (0.5f - q2q2 - q3q3) + _2bz
    * (q1q3 - q0q2) - mx) + (_2bx * q1 + _2bz * q3) * (_2bx * (q1q2
    - q0q3) + _2bz * (q0q1 + q2q3) - my) + (_2bx * q0 - _4bz * q2) *
    (_2bx * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
```

```
        s3 = _2q1 * (2.0f * q1q3 - _2q0q2 - ax) + _2q2 * (2.0f * q0q1 +
            _2q2q3 - ay) + (-_4bx * q3 + _2bz * q1) * (_2bx * (0.5f - q2q2 -
            q3q3) + _2bz * (q1q3 - q0q2) - mx) + (-_2bx * q0 + _2bz * q2) *
            (_2bx * (q1q2 - q0q3) + _2bz * (q0q1 + q2q3) - my) + _2bx * q1 *
            (_2bx * (q0q2 + q1q3) + _2bz * (0.5f - q1q1 - q2q2) - mz);
        recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
            normalize step magnitude
        s0 *= recipNorm;
        s1 *= recipNorm;
        s2 *= recipNorm;
        s3 *= recipNorm;

        // Apply feedback step
        qDot1 -= beta * s0;
        qDot2 -= beta * s1;
        qDot3 -= beta * s2;
        qDot4 -= beta * s3;
    }

    // Integrate rate of change of quaternion to yield quaternion
    q0 += qDot1 * (1.0f / sampleFreq);
    q1 += qDot2 * (1.0f / sampleFreq);
    q2 += qDot3 * (1.0f / sampleFreq);
    q3 += qDot4 * (1.0f / sampleFreq);

    // Normalise quaternion
    recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
    q0 *= recipNorm;
    q1 *= recipNorm;
    q2 *= recipNorm;
    q3 *= recipNorm;
}

//---------------------------------------------------------------------------------------------------
// IMU algorithm update

void MadgwickAHRSupdateIMU(float gx, float gy, float gz, float ax, float
    ay, float az) {
    float recipNorm;
    float s0, s1, s2, s3;
    float qDot1, qDot2, qDot3, qDot4;
    float _2q0, _2q1, _2q2, _2q3, _4q0, _4q1, _4q2 ,_8q1, _8q2, q0q0,
        q1q1, q2q2, q3q3;

    // Rate of change of quaternion from gyroscope
    qDot1 = 0.5f * (-q1 * gx - q2 * gy - q3 * gz);
    qDot2 = 0.5f * (q0 * gx + q2 * gz - q3 * gy);
    qDot3 = 0.5f * (q0 * gy - q1 * gz + q3 * gx);
    qDot4 = 0.5f * (q0 * gz + q1 * gy - q2 * gx);
```

```
// Compute feedback only if accelerometer measurement valid (avoids
    NaN in accelerometer normalization)
if(!((ax == 0.0f) && (ay == 0.0f) && (az == 0.0f))) {

   // Normalize accelerometer measurement
   recipNorm = invSqrt(ax * ax + ay * ay + az * az);
   ax *= recipNorm;
   ay *= recipNorm;
   az *= recipNorm;

   // Auxiliary variables to avoid repeated arithmetic
   _2q0 = 2.0f * q0;
   _2q1 = 2.0f * q1;
   _2q2 = 2.0f * q2;
   _2q3 = 2.0f * q3;
   _4q0 = 4.0f * q0;
   _4q1 = 4.0f * q1;
   _4q2 = 4.0f * q2;
   _8q1 = 8.0f * q1;
   _8q2 = 8.0f * q2;
   q0q0 = q0 * q0;
   q1q1 = q1 * q1;
   q2q2 = q2 * q2;
   q3q3 = q3 * q3;

   // Gradient decent algorithm corrective step
   s0 = _4q0 * q2q2 + _2q2 * ax + _4q0 * q1q1 - _2q1 * ay;
   s1 = _4q1 * q3q3 - _2q3 * ax + 4.0f * q0q0 * q1 - _2q0 * ay - _4q1
      + _8q1 * q1q1 + _8q1 * q2q2 + _4q1 * az;
   s2 = 4.0f * q0q0 * q2 + _2q0 * ax + _4q2 * q3q3 - _2q3 * ay - _4q2
      + _8q2 * q1q1 + _8q2 * q2q2 + _4q2 * az;
   s3 = 4.0f * q1q1 * q3 - _2q1 * ax + 4.0f * q2q2 * q3 - _2q2 * ay;
   recipNorm = invSqrt(s0 * s0 + s1 * s1 + s2 * s2 + s3 * s3); //
      normalize step magnitude
   s0 *= recipNorm;
   s1 *= recipNorm;
   s2 *= recipNorm;
   s3 *= recipNorm;

   // Apply feedback step
   qDot1 -= beta * s0;
   qDot2 -= beta * s1;
   qDot3 -= beta * s2;
   qDot4 -= beta * s3;
}

// Integrate rate of change of quaternion to yield quaternion
q0 += qDot1 * (1.0f / sampleFreq);
q1 += qDot2 * (1.0f / sampleFreq);
q2 += qDot3 * (1.0f / sampleFreq);
q3 += qDot4 * (1.0f / sampleFreq);
```

```
   // Normalize quaternion
   recipNorm = invSqrt(q0 * q0 + q1 * q1 + q2 * q2 + q3 * q3);
   q0 *= recipNorm;
   q1 *= recipNorm;
   q2 *= recipNorm;
   q3 *= recipNorm;
}

//---------------------------------------------------------------------------------------------------
// Fast inverse square-root
// See: http://en.wikipedia.org/wiki/Fast_inverse_square_root

float invSqrt(float x) {
   float halfx = 0.5f * x;
   float y = x;
   long i = *(long*)&y;
   i = 0x5f3759df - (i>>1);
   y = *(float*)&i;
   y = y * (1.5f - (halfx * y * y));
   return y;
}

//=====================================================
// END OF CODE
//=====================================================
// Quaternions to Euler angles conversion:

// See:
    https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles
void QuaternionsToEuler(float q0, float q1, float q2, float q3){
   // roll:
   double sinr_cosp = 2*(q0*q1 + q2*q3);
   double cosr_cosp = 1 - 2*(q1*q1 + q2*q2);
   angle_roll = atan2(sinr_cosp, cosr_cosp);

   // pitch:
   double sinp = 2*(q0*q2 - q3*q1);
   if (fabs(sinp) >= 1)
      angle_pitch = copysign(PI/2, sinp); // use 90 degrees if out of
         range

   else // only works to +-pi/2
      angle_pitch = asin(sinp);

   // yaw:
   double siny_cosp = 2*(q0*q3 + q1*q2);
   double cosy_cosp = 1 - 2*(q2*q2 + q3*q3);
   angle_yaw = atan2(siny_cosp, cosy_cosp);
}
```