2018

Design and Evaluation of Novel Attitude Estimation System Using MEMS Sensors for Indoor UAS

Joshua Bruce Milam
WVU, jomilam@mail.wvu.edu

Follow this and additional works at: https://researchrepository.wvu.edu/etd
Part of the Navigation, Guidance, Control and Dynamics Commons

Recommended Citation
https://researchrepository.wvu.edu/etd/3711

This Thesis is brought to you for free and open access by The Research Repository @ WVU. It has been accepted for inclusion in Graduate Theses, Dissertations, and Problem Reports by an authorized administrator of The Research Repository @ WVU. For more information, please contact ian.harmon@mail.wvu.edu.
Design and Evaluation of Novel Attitude Estimation System Using MEMS Sensors for Indoor UAS

Joshua B. Milam

Thesis submitted
to the Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of

Master of Science in
Aerospace Engineering

Marcello Napolitano, Ph.D., Chair
Jason Gross, Ph.D.
Andres Velasquez, Ph.D.

Department of Mechanical and Aerospace Engineering

Morgantown, West Virginia
2018

Keywords: Attitude Estimation, IMU, IMU Array, MEMS, Sensor Fusion, Sensor Array, UAS
Copyright 2018 Joshua Milam
Abstract

Design and Evaluation of Novel Attitude Estimation System Using MEMS Sensors for Indoor UAS

Joshua Milam

Most small unmanned aerial systems in use today, employ extended Kalman filter sensor fusion algorithms in order to provide accurate estimations of attitude or orientation. These complex algorithms use measurements from GPS receivers and magnetometer sensors that can be rendered useless in GPS denied environments or areas of significant magnetic interference, such as inside buildings or other structures. The complexity of these algorithms makes them inaccessible for some researchers and hobbyists who wish to code their own attitude estimation algorithms. This complexity is also computationally expensive and requires processors that are powerful enough to operate the algorithms along with any command and control functions required by the application. In contrast, there are simple sensor fusion algorithms such as the complementary filter or linear Kalman filter, that are commonly used by hobbyists because they are relatively easy to implement and computationally lightweight. However, these methods are not as accurate as the extended Kalman filter and therefore, are not adequate for some of the emerging precision applications in aerial robotics.

The goal of this research is to investigate an attitude estimation algorithm that uses two separate inertial measurement units (IMUs), each consisting of tri-axis accelerometers and tri-axis gyroscopes. This dual or twin IMU (TIMU) algorithm is compared to several common algorithms that only use one IMU, such as the complementary filter and linear Kalman filter. Analysis of a one degree of freedom experiment shows that the TIMU algorithm provides a more accurate attitude estimate. The analysis also shows that distance between the IMU and the rotating body’s center of gravity can have an inverse effect on attitude accuracy. The ability of the algorithms to provide an accurate estimate of the rate of attitude change is used as a performance metric, in addition to the accuracy of attitude estimates. The complexity of the twin IMU algorithm is kept to a minimum. It is presented in a way that can be easily programed by the layman and has a small computational footprint.
ACKNOWLEDGMENTS

I owe the members of my committee, Dr. Napolitano, Dr. Gross, and Dr. Velasquez, each many thanks for their help and support through this process.

Dr. Napolitano, I have learned much from you over the past several years in your undergraduate classes and afterwards from your advice for me during graduate school. The opportunity you gave me when you hired me as a research assistant to work at WVRTC, is one of the most influential events that will shape my career. I am extremely thankful for this opportunity that funded my graduate education and helped me join the workforce. Without you, this process would have likely been much more difficult for me and my family. I am so glad to have had a professor that loves airplanes as much as you do.

Dr. Gross, thank you for your advice and willingness to work with me on finishing this thesis. If I ever go on to pursue a position academia, you will be one of my role models.

Dr. Velasquez, I have learned so much about many things over the past few years working with you at WVRTC. I owe you for much of my programming skills and interests in using open-source software as an engineering tool. Again, I have learned so much that I can’t describe it all. But, I can say that I am a much better, more capable engineer because of it. This thesis is what it is because of your ideas and input. It was born from our pursuits of your vision of aerial robotics. Thank you for your patience and guidance along the way.

I must thank my parents/step-parents. You all have blessed me beyond my merits and I would not be who I am or where I am without your selfless support throughout my life. I am extremely lucky to be your son. I hope that in the future, I can be as good of a parent as you all are to me. Grandma, all that goes for you too! I love you all very much.

I want to thank my sister Lindsay. Thank you for understanding that I have been working on this thesis and not able to spend time with you. I love you and am so proud of how you are doing in college. I think that you are on track to be very successful.

Lastly, but most importantly, I want to thank my lovely wife. Sarah, you make my life better many times over. I am so extremely thankful for your sacrifice while I have worked on this thesis. I hope that I can support you the way you support me. I love you so much and cannot wait to start the next chapter of our lives.
TABLE OF CONTENTS

Acknowledgments........................................................................................................................................... iii
List of Figures .................................................................................................................................................. vi
List of Tables ................................................................................................................................................ viii
List of Symbols ............................................................................................................................................. ix
List of Abbreviations .................................................................................................................................. x

Chapter 1: Introduction and Background.................................................................................................. 1
  1.1 Introduction ........................................................................................................................................... 1
  1.2 MEMS Sensors ................................................................................................................................... 2
  1.3 Sensor Fusion ...................................................................................................................................... 3
  1.4 Research Goals ................................................................................................................................. 5
  1.5 Document Structure .......................................................................................................................... 5

Chapter 2: Literature Review ....................................................................................................................... 7

Chapter 3: Experimental Design and Apparatus ................................................................................... 12
  3.1 The Experimental Model of Angular Motion .................................................................................... 12
  3.2 The Algorithms Used for Comparison: ............................................................................................. 14
    3.2.1 Algorithm #1: Integration of Angular Rate .................................................................................. 14
    3.2.2 Algorithm #2: Accelerometers Only ........................................................................................... 15
    3.2.3 Algorithm #3: Complementary Filter ......................................................................................... 15
    3.2.4 Algorithm #4: Kalman Filter ...................................................................................................... 16
    3.2.5 Algorithm #5: Madgwick Filter .................................................................................................. 18
    3.2.6 Algorithm #6: Mahony Filter .................................................................................................... 19
    3.2.7 Algorithms #7, #8, and #9: Novel/TIMU Filters ....................................................................... 21
  3.3 Experimental Apparatus and Data Processing ................................................................................. 23

Chapter 4: Results ....................................................................................................................................... 32
  4.1 Results: Pitch Estimates .................................................................................................................... 33
  4.2 Results: Pitch Rate Estimates .......................................................................................................... 48
  4.3 Results: Numerical Pitch and Pitch Rate Errors ............................................................................ 58
  4.4 Results: Algorithm Run Times ....................................................................................................... 62

Chapter 5: Conclusions and Recommendations .................................................................................... 64
  5.1 Analysis of Results ............................................................................................................................ 64
    5.1.1 Effects of Distance Between CG and IMU ............................................................................... 64
    5.1.2 Algorithm Accuracies and Performance .................................................................................... 65
    5.1.3 Complexity and Computational Cost .......................................................................................... 66
5.2 Recommendations for Further Research .......................................................... 66

References .................................................................................................................. 68

APPENDIX A: Optimized Kalman Filter Code ....................................................... 71
APPENDIX B: Optimized TIMU Algorithm Code .................................................... 73
APPENDIX C: Quaternion Multiplication and other Operations ............................. 75
APPENDIX D: Arduino Code Used in Experiment ..................................................... 76
APPENDIX E: MATLAB Code Used in Experiment .................................................. 87
LIST OF FIGURES

Figure 1.1: Waveshare 10-DOF (B) IMU sensor board [6] ................................. 3
Figure 3.1: Diagram of rotating rigid body and accompanying IMUs. .................. 12
Figure 3.2: Free-body diagram of IMU #0 as drawn in Figure 3.1 ....................... 13
Figure 3.3: Photograph of experimental apparatus used in this research. ............. 24
Figure 3.4: The encoder and locking pin on the experimetal apparatus. ............... 25
Figure 3.5: Photograph of the Waveshare 10 DOF IMU sensor. Two are mounted to the experimental apparatus. This is IMU #1 ................................. 25
Figure 3.6: The Arduino Mega 2560 used to power and interface with IMUs ........ 26
Figure 3.7: Multiplexer which allowed using two identical IMUs at the same time 26
Figure 3.8: Excerpt from the recorded data. ................................................... 27
Figure 3.9: The left plot shows the convergence of a “P” matrix value without pre-convergence. The right plot shows the same value with pre-convergence. Note the scale of the dependant axes. ......................................................... 30
Figure 3.10: The change in the values of “E” for the Mahony algorithm during calculation of pitch estimate. ................................................................. 31
Figure 4.1: Ground truth pitch from the recorded data ...................................... 32
Figure 4.2: Ground truth pitch rate from the recorded data .............................. 33
Figure 4.3: All of the pitch estimates from IMU #0 data and ground truth plotted together with labeled regions of interest ........................................... 34
Figure 4.4: Zoomed in view of region A from Figure 4.3 ............................... 35
Figure 4.5: Zoomed in view of region A from Figure 4.3 ............................... 36
Figure 4.6: Zoomed in view of region A from Figure 4.3 ............................... 37
Figure 4.7: Zoomed in view of region B from Figure 4.3 ............................... 38
Figure 4.8: Zoomed in view of region B from Figure 4.3 ............................... 39
Figure 4.9: Zoomed in view of region B from Figure 4.3 ............................... 40
Figure 4.10: Zoomed in view of region C from Figure 4.3 ............................. 41
Figure 4.11: Zoomed in view of region C from Figure 4.3 ............................. 42
Figure 4.12: Zoomed in view of region D from Figure 4.3 ............................. 43
Figure 4.13: Zoomed in view of region E from Figure 4.3 ............................. 44
Figure 4.14: Zoomed in view of region E from Figure 4.3 ............................. 45
Figure 4.15: All of the pitch estimates from IMU #1 data and ground truth plotted together with labeled regions of interest ........................................... 46
Figure 4.16: Zoomed in view of region B from Figure 4.15 .......................... 47
Figure 4.17: Zoomed in view of region E from Figure 4.15 .......................................................... 48
Figure 4.18: All of the pitch rate estimates from IMU #0 data and ground truth plotted together. .......................................................................................................................... 49
Figure 4.19: All of the pitch rate estimates from IMU #0 data and ground truth except for algorithm #2, plotted together .......................................................... 50
Figure 4.20: All of the pitch rate estimates from IMU #0 data and ground truth except for algorithm #2 and #7, plotted together with labeled regions of interest ........................................ 51
Figure 4.21: Zoomed in view of region A from Figure 4.20 .......................................................... 52
Figure 4.22: Zoomed in view of region A from Figure 4.20 .......................................................... 53
Figure 4.23: Zoomed in view of region A from Figure 4.20 .......................................................... 54
Figure 4.24: Zoomed in view of region B from Figure 4.20 .......................................................... 55
Figure 4.25: Zoomed in view of region C from Figure 4.20 .......................................................... 56
Figure 4.26: Zoomed in view of region C from Figure 4.20 .......................................................... 57
Figure 4.27: All of the pitch rate estimates from IMU #1 data and ground truth plotted together. .......................................................................................................................... 58
Figure 4.28: Bar graph showing the sum of normalized RMS pitch and RMS pitch rate errors for IMU #0 .......................................................................................................................... 60
Figure 4.29: Bar graph showing the sum of normalized RMS pitch and RMS pitch rate errors for IMU #1 .......................................................................................................................... 61
Figure 4.30: Bar graph showing the mean of summed normalized RMS pitch and RMS pitch rate errors from IMU #0 and IMU #1 .......................................................................................................................... 62
Figure 5.1: Measured accelerations in x-direction from IMU #0 and #1 plotted together .......... 64
LIST OF TABLES

Table 3.1: The upper/lower bounds and starting values used in the optimizations of tuning parameters for the algorithms. .................................................................................................................. 29

Table 3.2: The tuning values and associated errors found using the optimizer for the different algorithms. ........................................................................................................................................ 29

Table 4.1: Characteristics of the sample times from the recorded data. .......................................................... 32

Table 4.2: The RMS pitch errors and RMS pitch rate errors from each algorithm for IMU #0 and IMU #1 measurements. ........................................................................................................ 59

Table 4.3: The sum of normalized RMS and RMS pitch rate errors for both IMU #0 and #1 and their mean, the error index .................................................................................................. 60

Table 4.4: Average run times of algorithm functions recorded by MATLAB Profiler. .................. 63
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>Acceleration</td>
<td>$[G]$</td>
</tr>
<tr>
<td>$\Delta t$</td>
<td>Sample Time</td>
<td>$[sec]$</td>
</tr>
<tr>
<td>$\theta$</td>
<td>Pitch Angle</td>
<td>$[deg]$</td>
</tr>
<tr>
<td>$\sigma^2$</td>
<td>Covariance</td>
<td></td>
</tr>
<tr>
<td>$\omega$</td>
<td>Angular Rate</td>
<td>$[dps]$</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>AHRS</td>
<td>Attitude and Heading Reference Systems</td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>Center of Gravity</td>
<td></td>
</tr>
<tr>
<td>DIY</td>
<td>Do It Yourself</td>
<td></td>
</tr>
<tr>
<td>EKF</td>
<td>Extended Kalman Filter</td>
<td></td>
</tr>
<tr>
<td>GPS</td>
<td>Global Positioning System</td>
<td></td>
</tr>
<tr>
<td>IMU</td>
<td>Inertial Measurement Unit</td>
<td></td>
</tr>
<tr>
<td>MEMS</td>
<td>Microelectromechanical systems</td>
<td></td>
</tr>
<tr>
<td>RMS</td>
<td>Root Mean Square</td>
<td></td>
</tr>
<tr>
<td>TIMU</td>
<td>Twin IMU</td>
<td></td>
</tr>
<tr>
<td>UAS</td>
<td>Unmanned Aerial Systems</td>
<td></td>
</tr>
<tr>
<td>sUAS</td>
<td>Small Unmanned Aerial Systems</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION AND BACKGROUND

1.1 Introduction

Small Unmanned Aerial Systems (sUAS) generally require onboard attitude determination sensors often called an inertial measurement unit (IMU). The first examples of IMUs were used for azimuth control of early rockets but their use became more prevalent for inertial navigation on large aircraft. At that time, IMUs were relatively large devices. They could be almost two feet in diameter and weigh 100 pounds or more [1]. The modern electronic IMUs investigated in this study consists of small microchips that weigh fractions of a gram so that they are suitable for installation on small devices such as cell phones and sUAS. The accuracy of these tiny IMU devices is generally poor. They suffer from significant noise and drift. Even if sUAS are not completely controlled autonomously, they often have compensation systems to stabilize the aircraft for the human pilot [2]. The accuracy of the IMU is extremely important to ensure the correct operation of any compensation systems or automatic controls and is investigated herein.

IMUs used for sUAS have evolved much over the past several decades. The most modern systems incorporate Global Positioning System (GPS) and magnetometer information to provide very accurate attitude information, including UAS’s position, attitude, and heading. These systems are called attitude and heading reference systems (AHRS) and incorporate the IMU to measure the linear and angular acceleration data along with a system to process the data into useful attitude and heading information. This type of system has the disadvantage that it relies on a clear view of the sky in order to receive GPS data and is susceptible to magnetic interference [2]. Usually, this is not a problem for aircraft because they are generally used in outdoor settings. However, there are many emerging applications for sUAS in indoor settings. The challenge in most indoor settings is that they are “GPS denied” environments and may have areas of significant magnetic interference. This prohibits GPS information and can skew the information from the magnetometer. In this type of environment, the attitude determination system may be severely crippled rendering the sUAS without the ability to maintain a safe attitude or navigate. There are attitude determination schemes that do not require GPS or magnetometers, but they have limitations in accuracy that make them less than adequate for precision autonomous sUAS applications. The main goal of this investigation is to compare different attitude determination schemes that do not require GPS or magnetometers and to propose a novel approach to this problem.

Another consideration which lead to this research is the notion that there are emerging applications for sUAS that will require the center of gravity (CG) of the sUAS to significantly change location during flight. Examples may include a delivery drone that must offload cargo in an asymmetric way or a sUAS with a robotic manipulator. The movement of the robotic manipulator will change the CG of the carrier sUAS very rapidly and in an infinite number of locations within a certain 3-dimensional envelope. The reason this is such a concern from the perspective of attitude determination is because the attitude sensor will experience a different level of linear acceleration versus angular accelerations as the position of the sensor relative to the CG changes. For example, if the sensor is very close to the CG, then the linear accelerations exerted on the sensor during a pure rotational change in attitude of the sUAS, will be small. Conversely, the farther away from the CG the sensor is, the larger the linear accelerations. Hypothetically, this will cause a decrease in attitude estimation accuracy.
Consider that the sUAS will always rotate around its CG and when the sensor is not located at the CG, the sensor has a translational component of motion when the sUAS changes attitude. This causes a linear acceleration to be exerted on the sensor. Most attitude measurement schemes use the acceleration of gravity as a reference. If there is another linear acceleration present, this skews the perception of the direction of gravity. Of course, sUAS will accelerate linearly as they move, making this an important issue for on-board attitude estimation.

Gyroscopes, or gyros for short, have been integral parts of IMU systems since the earliest examples of IMU. At that time, they consisted of spinning masses that could be used to measure angular rate based on the fact that they resist angular motion once spun at high speed [1]. In fact, the spinning mass gyro is still in use today in many aerospace application ranging from aircraft instruments to spacecraft navigation [3]. The reason why gyros are important to this research is because once attached to a rigid body, they provide angular motion information about the rigid body regardless of location on said body. This fact is leveraged to mitigate some of the measurement inaccuracies caused by the location of the IMU relative to the CG mentioned in the preceding paragraphs. The gyros that are often referred to in this document are not spinning mass gyros. Instead, they consist of microscopic vibrating masses that are displaced by Coriolis forces when the gyro experiences rotation. These gyros, and their accelerometer counterparts, are called Microelectromechanical systems sensors (MEMS) [4].

1.2 MEMS Sensors

MEMS sensors consist of microscopic mechanical systems that take advantage of principles of electricity to provide sensing capabilities for a wide range of applications. The most common type of MEMS accelerometers use capacitive transduction, where tiny plates inside the accelerometer move closer or farther from one another depending on accelerations exerted on the sensor [4]. The relative displacement of the plates causes a change in the capacitance through the plates which is then translated into a measure of the accelerations. MEMS gyroscopes, mentioned earlier, are similar in nature to the accelerometers. There are different types available, but the most common types operate in a similar fashion to the accelerometers. The gyroscopes use an oscillating mass or a vibrating disc in close proximity to fixed plates. The Coriolis Effect of the rotating sensor causes the distance between the mass and the fixed plates to change which is then interpreted as an angular rate.

MEMS offer a very cheap, and compact way of measuring angular rates and accelerations, making them attractive for installation on sUAS. To be specific, the MEMS microchips used in this research measure 3x3x0.9mm, weigh 0.1g and cost about $7 per chip [5].
In this small IMU chip, pictured in Figure 1.1 and labeled with an arrow, there are tri-axis accelerometers, tri-axis gyroscopes, tri-axis magnetometers, and a thermometer. Clearly, this kind of capability in such a tiny footprint is attractive for sUAS as well as countless other applications. “MEMS sensors” in this document is synonymous with the inexpensive consumer grade sensors, though higher cost, tactical grade MEMS sensors are available [7]. The inexpensive, consumer grade MEMS sensors are significantly less accurate than the larger types of IMU sensors such as tactical grade MEMS, ring laser, fiber optic, and spinning mass gyroscopes. High drift and noise associated with MEMS sensors are significant impediments to the usefulness of the sensors for control of sUAS. This has proliferated research into ways of achieving accurate attitude information from MEMS sensors. Many different schemes have been proposed. These schemes are generally called sensor fusion because they involve taking the information from the different specific sensors that make-up IMU sensors, and “fusing” the information to create a unified estimate with improved accuracy.

1.3 Sensor Fusion

The most common methods of sensor fusion involve different forms of Kalman filtering or complementary filters, both of which are considered in this research. Kalman filtering is the use of a recursive optimal estimator, based on a linear model to predict the behavior of the system’s states. The filter accounts for the dynamics and statistical properties of the system [8]. In other words, the Kalman filter produces a best possible guess of the dynamics of a linear system even if the inputs to the system are noisy. It has disadvantages that will be described later which mostly revolve around that fact that in reality, systems do not behave in a perfect linear fashion. A complementary filter for the application in question, involves fusing the gyro measurements filtered through a high pass filter and the accelerometer measurements filtered through a low pass filter. Since the MEMS gyros are very prone to drift but not noise, and the accelerometers are very noisy but not as susceptible to drift, complementary filters combine the low frequency information from accelerometers and the high frequency gyro data into a composite prediction. This is much more accurate than using either the gyros or accelerometers by themselves.
There is another very common sensor fusion scheme proposed by Magdwick which is often referred to in literature for comparisons. It uses quaternions to represent attitude information and is said to provide better accuracy with less complexity and computational cost as the Kalman filter [9]. The Mahony algorithm is another commonly referenced algorithm that was likely the basis for the Madgwick algorithm and operates on similar principles [10]. This study will compare the performance of the Kalman filter, complementary filter, Madgwick algorithm, Mahony algorithm and a few novel approaches.

All of the algorithms that are evaluated against the novel solutions in this research are widely used for hobbyist grade experimentation with sUAS. In fact many outlets that promote Do-It-Yourself (DIY) UAS projects promote the use of some of the algorithms, especially the complementary filter [11], [12], [13], [14]. The software that was distributed by the manufacturer of the IMUs used in this study employed the Mahony algorithm, which is evaluated herein. An important goal of this research is to propose a novel algorithm that is on par with the others in terms of complexity. There are two reasons for this. The first is that significantly increasing complexity might incur enough computational cost that the algorithm cannot be practically employed on a microcontroller in addition to other processes required for the UAS such as receiving and interpreting control inputs and stabilization compensation. The second is that very complicated algorithms may be inaccessible to hobbyists who may have basic programming resources and wish to use the UAS in a non-academic setting. Such an audience likely prefers an easy to understand and easily adoptable solution. An illustration of this is the fact that DIY drone hobbyists who program their own sensor fusion, still commonly use complementary filters despite years of academic research on other more advanced methods [15]. Improvements exits over the original that boast great performance, even compared to non-linear Kalman filtering, but online blogs are filled with descriptions of single axis first order complementary filters and single axis linear Kalman filters like those evaluated here.

Performance of the algorithms evaluated herein is measured by the root mean square (RMS) error of a pitch estimate and the RMS error of an approximate derivative of the pitch estimate. First, a RMS error is calculated between the pitch estimate and a ground truth pitch value. Then an approximate derivative of the pitch is calculated using a first-order backwards finite difference equation shown below.

\[
\hat{\theta}_k = \frac{\theta_k - \theta_{k-1}}{\Delta t} \tag{1.1}
\]

Equation 1.1 shows how the approximate derivative of pitch, “\(\hat{\theta}_k\)”, is calculated using the difference between the pitch estimate from the current time step, “\(\theta_k\)” and the estimate from last time step, “\(\theta_{k-1}\)”, divided by the duration of the time step, “\(\Delta t\)”. The “\(k\)” subscripts indicate whether the variable is taken from the current time step or otherwise, following common convention.

In common aerospace convention, “pitch rate” is the term used to describe the angular velocity of an aircraft about its lateral axis. This is not the same as the derivative of pitch, “\(\hat{\theta}\)” unless the aircraft’s lateral axis is parallel with the plane created by the North and East axes of the world or navigation frame. In other words, when the roll angle is zero. Since the experiment
described herein is one dimensional and the roll and yaw angels are kept at zero, “derivate of pitch” and “pitch rate” are used interchangeably in this document for brevity.

Much of the literature reviewed for this research only examined the pitch RMS error, and some presented the standard deviation as well. The inclusion of the derivative as a performance metric is important because the derivative is often used in control systems, and therefore will adversely affect the performance of the system if it is inaccurate. Pitch rate is often used in feedback controllers and autopilots that control pitch or go further to control altitude in the case of fixed wing aircraft. Through the course of this investigation it became apparent that only considering the RMS error of the pitch estimate did not give a clear picture of how the derivative tracked with the ground truth derivative. Even comparing the standard deviations and variances of the pitch estimates did not portray how the derivatives performed as concisely as simply calculating them and comparing them directly. Therefore, they cannot adequately represent performance of the derivative.

1.4 Research Goals

The goals of this investigation are:

1. Evaluate common sensor fusion algorithms that can estimate attitude using only MEMS accelerometer and gyro sensors.
2. Test the hypothesis that the IMU performance is inversely proportional to the distance from the center of gravity.
3. Propose a scheme that can offer better accuracy using two separate MEMS IMUs, and is comparable in terms of computational effort as well as complexity so that it can be easily implemented by the layman for DIY projects.
4. Investigate RMS derivative estimate error in addition to RMS estimate error.

The attempt to design a novel estimator that uses two separate IMUs instead of one, was born from the decision to install and simultaneously operate two separate IMUs on the experimental apparatus while collecting data. The original reason for the two IMUs operating in conjunction was to investigate the hypothesis that distance from the CG effects accuracy. The experimental apparatus is described in detail later, but a brief description is that the two IMUs are attached to a rigid structure that is changing attitude in one dimension around a center of rotation. One of the IMUs is closer to the center than the other. Since there are well established methods of attitude estimation using one IMU, it was decided to take advantage of the dual IMU data and try to design a method requiring two.

1.5 Document Structure

The next section in this document is a literature review of the sources that informed this research. The section following the literature review describes the details of the experiment and how the research was conducted. It begins with a detailed illustration that explains concepts from the literature review that are needed to understand the sensor fusion algorithms. The algorithms used for this research are then presented in a practical and concise way. This is followed by a section presenting the numerical results of the experiment which encompasses the relative performance of the different algorithms. The last section offers a concluding review of the results.
and recommendations based on the research. Several appendices at the end of the document are referred to where relevant.
CHAPTER 2: LITERATURE REVIEW

The experiment performed for this research is similar to many in the literature where a set of sensor data is processed using different filters or sensor fusion algorithms and the results from each compared. A good example of this is the report by McCarron where he compares the performance of using unfiltered gyro measurements, unfiltered accelerometer measurements, sensor fusion with a Kalman filter, and sensor fusion with a complementary filter [16]. The main focus of his report was to show that the Kalman filter produced a more accurate attitude estimate and could be coded in a way that was efficient, without excessive computational cost. There are some mistakes in the code provided in the appendix with the report. Those errors where corrected and a similar method used to code a Kalman filter for this research, which is presented in Appendix A.

A well know sensor fusion algorithm that is not a type of Kalman filter but instead is based on the complementary filter and principle of gradient decent, is the Mahony filter. As mentioned before, the IMU sensors that were procured for this research came with accompanying software that employed the Mahony algorithm for sensor fusion. Mahony authored multiple publications that presented this algorithm [10], [15]. One of these publications compared the position estimate using his algorithm with an estimate from a GPS and extended Kalman filter (EKF) based estimator, showing impressive performance of the Mahony algorithm [15]. In this 2008 paper, he states that most UAS use a form of EKF with magnetometers for yaw estimation, and that the UAS often have problems with inherent magnetic interference because they use electric motors and other equipment that creates the interference. He argues that this makes the EKF estimators less reliable. The Mahony filter has 2 tuning variables which is fewer than most Kalman filter sensor fusion schemes, which may makes the Mahony filter easier to tune.

Madgwick compares the performance of his algorithm with a proprietary Kalman filter that was included with the sensor he used for his experiments [9]. He presents results indicating that his algorithm provides a more accurate attitude estimate than the Kalman-based sensor fusion. The fact that the Madgwick algorithm only has one tuning variable is highlighted to bolster the argument that the Madgwick algorithm is much easier to tune than a Kalman filter, or the Mahony filter.

In the literature, using multiple IMUs is generally referred to as a “sensor array” or “redundant IMUs”. Most of the reviewed publications for this research were dated beginning in 2009 or later. This is likely related to the trends in availability and popularity of MEMS sensors. Yadav and Bleakley state in their 2011 publication that to their knowledge, they are the first to employ, “spatially separated dual IMUs on a single rigid body for position estimation” [17]. Reviewing more sources indicates that there were previous experiments using spatially separated IMUs. An example of this is the 2004 paper from Colomina, et al., documenting the collection of actual flight test data of two IMUs rigidly mounted close to one another (about 10 inches) [18]. They present the residual error between the two IMUs after transforming the coordinate frames into one such that they have two distinct sets of measurements about the same frame. Most of their investigation is focused on ways of using the redundant data, for either noise reduction, accuracy improvement or fault detection. They used large (3.5x3.5 inch) fiber-optic IMUs, which generally have outstanding accuracy compared to MEMS sensors, but are very expensive.
Conversely, Yadez and Bleakley were investigating microchip-sized sensors. Their goal was to test and compare their solution, which they describe as a two-stage EKF, with other solutions for position estimation. They compare their solution with estimates from using one sensor, averaging two sensors, and averaging two sensors before using a Kalman filter. According to their publication, simulation-only results indicate that their solution improves accuracy by 30% over the solution that simply uses one sensor and no filter. The first stage of their solution uses a Kalman filter to estimate the orientation for both IMUs separately. The second stage uses the EKF to combine the two estimates into a position estimate. It appears that they do not consider fusing the data from the two IMUs during the orientation estimation step.

Published in 2016, Nilsson and Skog assembled an impressive literature review on the subject of MEMS sensor arrays that became a valuable resource for this investigation [19]. They give a very brief explanation of the paradigms common to sensor array research and reference over 300 publications. Their references are organized into categories and areas of focus. One of the most apparent deductions one can make by reviewing their list of documents is how prolifically variations of the Kalman filter are used in this area. Another large commonality in the publications is the usage of “no-gyro” systems, where an array of accelerometers is used to determine angular motion in lieu of gyros. They present a simple model that relates the accelerations measured at an arbitrary number of accelerometers as well as the gyro measurements at those locations, to the angular motion and linear accelerations of the entire system, plus measurement errors. It is by accounting for the linear accelerations, centripetal accelerations, and tangential accelerations, one can relate the common angular motion of all the sensors in the rigid array as long as the relative distances are known. This is the cornerstone of the novel estimator presented herein and will be further addressed later.

They briefly discuss several other important issues, and make important claims. One of these is that the minimum number of accelerometers required to deduce three-dimensional angular motion is six, or two triad accelerometers, which is intuitive. They also give sources that state that the gravity vector can be separated which is relevant when considering the very dynamic motion of aircraft and how it effects the perception of gravity such as during coordinated turns.

One of the papers cited by Nilsson and Skog, published in 2012 by He and Cardou, chronicles the usage of centripetal and tangential acceleration in a model designed to determine angular motion without gyros [20]. This no-gyro solution uses a weighting scheme to combine the time integration and quadratic form of the angular velocity components. This paper demonstrates common problems shared by many of the reviewed resources. The solution presented here seems fairly straightforward at first, but further investigation shows that it is likely too complicated for those of basic programming abilities and will incur great computational cost that may be too much for small microcontrollers. Their solution is approximately ten steps long, one of which is an algorithm all on its own, and the rest are non-trivial equations.

One of the later publications reviewed, published by Rasoulzdeh and Shahri in 2017, states that the vast majority of research in the area of MEMS sensor arrays is Kalman filter-based [21]. They then present an algorithm that is two stage. First, minimum-mean-square-error criterion are used to combine angular rate measurements from 4 MEMS gyros. Second, an iterative finite
impulse response filter is used in lieu of a Kalman filter to reduce the standard deviation of the fused signal from the first stage. They provide data collected using a 3-axis turntable indicating a five-fold increase in accuracy versus a single gyro. In their introduction, they cite research that achieved accuracy on par with fiber optic IMUs by using arrays of 100 MEMS sensors. They also cite Nilsson and Skog regarding the averaging of MEMS sensor measurements. Their research shows promise in the area of gyro accuracy improvement but does not address accelerometer data, though they claim that their method can be adapted to accelerometers.

Nilsson, Skog and Handel, published a description about a sensor array that they constructed, where they also discuss advantages to sensor arrays [22]. For example, they mention redundancy for fault tolerance and averaging out noise/bias errors. They again present the fact that non-collocated IMUs can be used for angular motion determination, which is relevant to this research. However, they do not investigate this. They present results of simply averaging measurements to boost accuracy, as well as a method of increasing communication speeds with the IMUs to make large arrays feasible.

Averaging the measurements and properly understanding the noise error does have an impact on performance and is important to consider. Martin, et al., provide a look at this subject and produce extensive tables with empirical error data and characteristics of MEMS sensor [23]. They present several methods aimed at reducing the overall error of arrays which go beyond simply averaging, but do not consider application specific solutions. None of their solutions are readily applicable to the general orientation estimation problem, but averaging will be used to a small extent in the proposed novel estimator of this investigation.

As mentioned earlier, many AHRS solutions involve GPS and magnetometers. Much of the sensor array research is aimed at maintaining position accuracy in the event of temporary GPS outage. An example of this kind of situation is the unreliability of GPS coverage while driving through narrow urban streets between tall buildings. Bancroft addresses this scenario, and presents a Kalman filter-based fusion scheme for 2 to 5 IMUs [24]. This scheme consists of a 21-state filter for each individual IMU that become blocks which form a single large filter. This filter incorporates GPS data and zero-velocity updates to keep the states accurate until there is a GPS outage. Bancroft artificially created GPS outages while collecting data with his sensor array mounted to the roof of a vehicle. His method did not consider the relative location between the IMUs and was mostly focused on position accuracy, though attitude information was included in the states he considered. In a later publication, Bancroft along with Lachapelle, document a similar study focused on pedestrian position tracking as opposed to a vehicle mounted array [25]. They again examine a large block filter scheme, and record data using both large tactical grade IMUs as well as MEMS IMUs. One disadvantage of their method is highlighted when they mention that the 5 IMU array can have around 120 tuning variables. This kind of complexity is not in line with the goals of this research. That is, to provide a practical solution with very small computational footprint, as well as having a greater focus on attitude estimation for airborne systems.

There are publications that show similar research to Bancroft but with an aerospace theme, such as the 2010 paper from Becker, et al [26]. They perform experiments similar to Bancroft but instead of a road vehicle or pedestrian platform for their array, they use a manned aircraft, and artificially simulate GPS outage. They also performed an extensive error and sensitivity analysis.
on their IMUs covering factors such as temperature, acceleration, turn rate, axes non-linearity, cross-coupling, and higher-order G-dependencies for the gyros. In this publication, they do not consider the relative placement of the IMU unlike, Colomina, et al.

Revisiting the road vehicle navigation perspective, Clausen, et al. present an experiment almost identical to Bancroft’s vehicle based investigation [27]. They do an excellent job of clearly presenting the method they used to improve accuracy over a single IMU system. Their method is to simply transform the frame of the IMUs into one, and average measurements from 4 IMUs. This is fairly simplistic but they show that it does significantly improve estimates and is in line with research previously mentioned above that discussed benefits of averaging.

Another road vehicle themed publication consulted for this research was written by Luo, et al. [28]. They concisely provide the background behind the model mentioned earlier that relates relative position of accelerometers to angular motion, based on the magnitudes of total perceived accelerations. They describe how pitch angle can be determined by taking the arctangent of two components of acceleration, if the object is only experiencing the acceleration of gravity, and how roll angle can be determined in a similar fashion. They use a Kalman filter on the IMU signals individually before processing them using the model to provide attitude and angular rate information. The simplicity behind their method is promising for this research. The novel method presented later herein is similar, but the raw IMU data is processed through the model and then filtering occurs. This decreases complexity by not employing two separate filters on each IMU.

Another publication, by Tsai, et al., presents something similar, a non-gyro method intended for motion tracking of wearable objects [29]. They give a fairly detailed description of how they model the related nature of redundant IMUs. Instead of filtering each IMU signal prior to fusion, as presented by Luo, et al., they use complex criterion to screen the individual signals prior to including them in the model and calculating attitude estimation. They also use averaging to reduce noise. Using a set of screening criterion seems non-generalizable and non-scalable, because the criteria would have to be application and maybe even sensor specific.

The most recent paper reviewed for this research is a 2018 publication by Wahlstrom, Skog and Handel [30]. Wahlstrom, et al. focus on processing the signals with a measurement model and minimization algorithm to mitigate noise, similar to Kalman filtering. They use a motion model where the signals are combined with a linear least square estimator to determine angular motion. Again, this is more complex than desired in terms of the goals of this research. It does however reinforce the notion that a linear motion model can be effectively used to fuse IMU data.

The 2014 paper by Al-Rawashdeh, et al. was reviewed in reference to the topic of variable CG location on UAS mentioned earlier [31]. They provide a no-gyro method using three banks of 12 accelerometers mounted far from one another on an aircraft. Their simulation results seem to indicate success and they refer to other similar efforts. They use the motion model described in the other papers above. They use an EKF scheme to filter the measurements and recommend exploring different measurement models as well, similar to Wahlstrom, et al. This further substantiates the motion method referred to in many of the reviewed papers.
Charel, et al. describe in their 2016 paper, an effort to apply a dual-IMU array to attitude measurement for an inverted pendulum on a wheel, a small unicycle [32]. They describe their main motivation of using a dual-IMU system as an effort to increase the sample rate of IMU measurements. The interesting thing about what they present, is that they use a complementary filter to fuse the measurements from the two sensors into a single estimate. The two signals are staggered so that when the signal from one sensor arrives at the microcontroller, the signal from the other sensor is halfway through the measurement cycle, this essentially doubles the measurement sample rate. The composite signal, that is half of one sensor and half of the other, is smoothed together using a complementary filter. Prior to being combined, the individual signals are filtered using a Kalman filter, similar to what Luo, et al. and others above described.

After reviewing the literature on redundant IMUs, it seems that the motion model provided in many of the above papers is an important paradigm to adopt for this research. It is after all, the basic rigid body dynamic description that relates linear accelerations and angular motion. The use of Kalman filters to remove noise errors is another very common theme, but scaling becomes problematic because of the complexity of the filters which incur increasing computational costs and difficulty in tuning. Therefore, Kalman filtering is not considered for the novel solution presented here. Since the one-dimensional linear Kalman filter is commonly referenced for use in DIY single IMU applications, it will be evaluated along with other single IMU algorithms. Averaging of IMU measurements is another recurrent theme in the literature that is used to a small extent in the proposed solution.
CHAPTER 3: EXPERIMENTAL DESIGN AND APPARATUS

3.1 The Experimental Model of Angular Motion

The motion model mentioned in the literature review and used throughout this research is more thoroughly explained here for the benefit of the reader in a way that is consistent with the experimental apparatus. An idealized diagram of the experimental apparatus is given below.

The diagram in Figure 3.1 above represents a rigid body that is rotating around its CG with angular velocity “ω” and angular acceleration “α”. The linear accelerations and angular rates are measured by the two IMUs represented by squares and marked “IMU #0” and “IMU #1”. The acceleration vectors representing the tangential acceleration at IMU #0 is labeled “A₀,tan” and the centripetal acceleration is labeled “A₀,cen”. The accelerations at IMU #1 are labeled in the same manner but with the “₁” subscript. “Aₙ” represents the acceleration of gravity that is acting on the rigid body. The dashed lines with “R” labels represent the distances of the IMUs from the CG or center of rotation. “R₀” represents the radial distance of IMU #0 from the CG. “R₀,x” and “R₀,z” are the distances along the x and z-axes respectively, and the same is true with IMU #1 with the “₁” subscript.

Figure 3.1: Diagram of rotating rigid body and accompanying IMUs.
subscripts. Focusing on one of the IMUs will better illustrate the components of the linear accelerations that are relevant to this research.

Figure 3.2: Free-body diagram of IMU #0 as drawn in Figure 3.1.

Figure 3.2 shows the components of the tangential and centripetal accelerations acting on IMU #0 divided into x and z-components, distinguished by the addition of “x” or “z” subscripts. The sensing axes of the IMU are in the x and z-directions, so the summation of the components along the x and z-axes is the acceleration measured by the IMU. The tangential and centripetal accelerations are dependent on the radial distance of the IMU from the CG.

\[ A_{0,cen} = R_0 \omega^2 \] 3.1

\[ A_{0,tan} = R_0 \alpha \] 3.2

Equations 3.1 and 3.2, show the expressions for centripetal and tangential accelerations at IMU #0. Below, these are broken into x and z-components to calculate the total acceleration in the x and z-directions.

\[ A_{0,x} = A_{0,cen,x} + A_{0,tan,x} = R_{0,x} \omega^2 + R_{0,z} \alpha \] 3.3

\[ A_{0,z} = A_{0,cen,z} + A_{0,tan,z} + A_g = R_{0,z} \omega^2 + R_{0,x} \alpha + A_g \] 3.4

Equations 3.3 and 3.4 show how the accelerations measured by the IMU along the x and z-axes are related to the angular velocity, angular acceleration and the acceleration due to gravity.
The angular velocity and acceleration for both IMUs are the same, so applying Equations 3.1 and 3.2 to IMU #1 with the associated distances between IMU #1 and the CG, will yield smaller values for tangential and centripetal acceleration. This is because the radial distance between IMU #1 and the CG is smaller. This is exploited to build a model that relates the differences in measured accelerations at the two IMUs to the angular motion if the components of the radial distances are known.

3.2 The Algorithms Used for Comparison:

There are nine different algorithms that are considered for comparison. A few of them are closely related. Three of them are novel approaches to attitude described here. These three require measurements from two separate IMUs. For this reason, they are referred to herein as Twin IMU (TIMU) algorithms or simply Novel algorithms. To simplify this experiment, only one degree of freedom was considered. To make the algorithm explanations and analysis more consistent with aerospace applications, the single axis is assumed to be the pitch axis, and the pitch attitude is represented by “θ” with units of degrees. The angular rate, is represented by “ω” to make the equations more general and familiar to a wider audience. Therefore, pitch rate which is normally represented by “q” in aerospace conventions, is represented by “ω_y”, with units of degrees per second. “q” is used herein to represent quaternions, which is explained where relevant. As mentioned in the first chapter, pitch rate and derivative of pitch, “θ̇”, are used interchangeably in the document for brevity since the experiment is restricted to one degree or freedom.

To be clear, only two of the algorithms described below, the Madgwick (#5) and Mahony (#6) are complete three dimensional attitude estimation algorithms. The others are simplified to fit the one dimensional experiment for this research. Therefore, the equations and explanations given below for all but algorithms #5 and #6, need further generalization to account for rotation transformations in three dimensions if they are to be used as three dimensional estimation schemes. Since this experiment is restrained to one dimension, this does not affect the results except for the computation time of the algorithms which is addressed later. Despite this, there are sources that use the one axis algorithms as they are below to determine attitude for multi-copter UAS [11][12]. Two single axis estimators, one for pitch and one for roll, are implemented in tandem to provide a theoretically incorrect but still practically usable attitude determination scheme.

3.2.1 Algorithm #1: Integration of Angular Rate

The first algorithm simply integrates the angular rate measured by the gyro to provide attitude. An initial attitude is usually set to zero, and the change in attitude is equal to the angular rate sampled at a given time step multiplied by the length of the time step. The integrator used is the first order Euler forward method.

θ = ∫ ω_y dt

θ_k = θ_{k-1} + ω_{y,k}Δt

Equation 3.5 simply shows the continuous representation of the relationship between pitch, “θ”, and angular rate around the y-axis, “ω_y”. Equation 3.6 is the Euler forward method used to
process the angular rate measurements from the gyro to calculate the pitch at each time step. The angular rate measured during the specific cycle of the algorithm, “$\omega_{y,k}$”, in degrees per second is multiplied by the sample time in seconds, “$\Delta t$” and summed with the value of pitch from the previous time step, “$\theta_{k-1}$”, to get the estimated pitch for the current time step, “$\theta_k$”. While this is an extremely simple algorithm, a major disadvantage is that the MEMS gyros have significant non-linear drift that can affect the accuracy after only a few seconds.

### 3.2.2 Algorithm #2: Accelerometers Only

The second algorithm involves using the accelerometer data and the assumption that gravity is the only acceleration [16]. Accelerometers in two axis can then be used to calculate an angle for the direction of gravity in one plane. The attitude of the sUAS is described as an angle relative to the gravity vector. This is done for both pitch and roll separately.

$$\theta_k = \text{atan2}(a_{x,k}, -a_{z,k})$$  \hspace{1cm} 3.7

Equation 3.7 states that the 2-argument arctangent of the measured acceleration along the x-axis and the negative acceleration along the z-axis, “$a_x$” and “$-a_z$”, respectively, equals the angle between the gravity vector and the z-axis along the xz-plane. This angle is equal to the pitch angle. There are two major problems with this algorithm. The first is that it assumes gravity is the only linear acceleration acting on the IMU. Naturally, for a sUAS in flight this is not true most of the time. Even if the sUAS was stationary in one location and changing attitude by pure rotation, unless the IMU was located at the CG, the accelerometers would experience linear accelerations in addition to gravity. The non-gravity accelerations cause the estimation from this algorithm to be inaccurate. Lastly, this algorithm’s performance suffers because of the extremely noisy measurements coming from the MEMS accelerometers.

### 3.2.3 Algorithm #3: Complementary Filter

The third algorithm is the complementary filter which is a sensor fusion scheme that consists of a low pass filter applied to the accelerometer measurements and a high pass filter applied to the gyro measurements [33]. This takes advantage of the fact that the accelerometers are very noisy but are accurate if considered from a low frequency. Conversely, the gyros are accurate if considering small time intervals but are less accurate over larger time intervals because of their significant drift. Like the previous algorithms, this is a fairly simple method when used in discrete time as illustrated below.

$$\theta_k = K_c(\theta_{k-1} + \omega_{y,k}\Delta t) + (1 - K_c)\text{atan2}(a_{x,k}, -a_{z,k})$$  \hspace{1cm} 3.8

The first term on the right-hand side of Equation 3.8 above is the same as Equation 3.6 in Algorithm #1 multiplied by the tuning factor or gain, “$K_c$”. “$\omega_{y,k}$” is the angular rate measurement about the y-axis from the gyro for the current time step. The second term is the same as Equation 3.7 in Algorithm #2 multiplied by the reciprocal of the tuning factor. This algorithm still suffers from the disadvantages of the two previous algorithms but to a much lesser extent. Another disadvantage is the apparent lag between the actual and the estimated signals. This is characteristic of discrete time first order filters, or weighted moving average filters.
3.2.4 Algorithm #4: Kalman Filter

The forth algorithm is a Kalman filter [34] [16]. The Kalman filter takes a statistical approach to estimate the state(s) based on inputs and past observations. This is a fairly complicated but powerful algorithm.

Because of its complexity, this algorithm is explained by giving step-by-step instructions for each of the equations in their order for one iteration of the algorithm. This starts with calculating the “a priori” state vector estimate denoted by the “-” superscript. This just means that it is based on the state vector calculated during the previous time step. The “a posteriori” state vector will be calculated afterwards during the same time step, or iteration, based on this initial estimate and other “a priori” information. Another way to describe this is that the “a priori” is a predicted estimate based on a model of the dynamic system and the “a posteriori” is a corrected estimate which considers the predictions, the statistical properties or noise of the measurements as well as alternative measurements.

\[ x_{k^-} = Ax_{k-1} + B\omega_{y,k} \] \hspace{2cm} 3.9

In Equation 3.9, “\( x_{k^-} \)” represents the “a priori” state vector estimate. “\( x_{k-1} \)” is the “a posteriori” state vector estimate from the previous iteration, or the initial state estimate during the first iteration. The “A” and “B” represent the state and input matrices respectively. The matrices used to model the IMU, including the state vector, are given below.

\( x = \begin{bmatrix} \theta \\ b_{gyro} \end{bmatrix} \) \hspace{2cm} 3.10

\( A = \begin{bmatrix} 1 & -\Delta t \\ 0 & 1 \end{bmatrix} \) \hspace{2cm} 3.11

\( B = \begin{bmatrix} \Delta t \\ 0 \end{bmatrix} \) \hspace{2cm} 3.12

Note that the value of pitch, represented by “\( \theta \)”, refers to the “a priori” or “a posteriori” pitch estimate depending on the notation applied to “\( x \)”. “\( b_{gyro} \)” is the gyro bias that is estimated by the filter in order to mitigate gyro bias error.

The next step is to calculate the “a priori” or predicted estimate covariance matrix.

\[ P_{k^-} = AP_{k-1}A^T + Q \] \hspace{2cm} 3.13

Equations 3.13 shows that the predicted covariance matrix, “\( P_{k^-} \)”, is calculated using the corrected covariance matrix form the previous time step and the covariance for process noise, “\( Q \)”, which is a characteristic of the accelerometer.
Equation 3.14 shows that "$Q$" is a diagonal matrix with the process noise covariance for the accelerometer, "$\sigma_{\text{accel}}^2$", populating the diagonal.

Once the predicted state and covariance estimates are calculated, they are used to determine the Kalman gain matrix. The Kalman gain matrix is used to correct the predicted state and covariance matrices.

$$K_k = P_k^- H^T((HP_k^- H^T + R)^{-1})$$  \hspace{1cm} (3.15)

$$H = [1 \ 0]$$  \hspace{1cm} (3.16)

$$R = [\sigma_{\text{gyro}}^2]$$  \hspace{1cm} (3.17)

Equation 3.15 requires the measurement matrix, "$H$", and the matrix for measurement noise covariance, "$R$", shown in Equations 3.16 and 3.17 respectively. Note that the measurement matrix is model specific similar to the state and input matrices. The equation below shows how it is used to relate the state equation to the accelerometer measurements.

$$x_k = x_k^- + K_k (\theta_{k,\text{accel}} - H x_k^-)$$  \hspace{1cm} (3.18)

The corrected state vector estimate, "$x_k$", is calculated using Equation 3.18 above where the difference between the predicted state vector and the pitch measurement from the accelerometer is multiplied by the Kalman gain, then added to the predicted state vector. The pitch measurement from the accelerometers, "$\theta_{k,\text{accel}}$", is found using Equation 3.7. Next, the corrected estimate covariance is calculated, the last step in the iteration.

$$P_k = (I - K_k H)P_k^-$$  \hspace{1cm} (3.19)

$$P_0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$  \hspace{1cm} (3.20)

Equation 3.20 shows the initial covariance matrix used in the MATLAB scripts for this research, but it is important to understand that the "$P$" matrix will change during the first time step and will continue to change until it converges to values in accordance with the sensor noise covariance values in the "$Q$" and "$R$" matrices as well as the Kalman gain matrix and the model specific, "$A$" and "$H$" matrices.

The Kalman filter is widely used and considered very accurate and reliable, but it does have significant disadvantages. Perhaps the most significant disadvantage for this particular problem is how difficult the algorithm is to tune. The tuning parameters are the noise covariance
values of the “Q” and “R” matrices. If the sensor’s noise variation is constant then this would not create such a difficulty, however, this simplistic characterization of the sensor noise is not adequate for the dynamic noise properties of the actual sensor. The noise characteristics will change depending on the accelerations being exerted on the sensor or the vibrations of the device on which it is mounted, causing the Kalman filter to become less effective. In fact, the noise characteristics of one sensor compared with another of the same make and model will be different enough to make using one set of tuning values for a filter, less than optimal for both sensors.

The other serious disadvantage of the Kalman filter is the computational cost associated with its complexity. However, the IMU problem is simple enough that the Kalman filter’s computational cost is not unreasonable. Also, the equations above can be simplified such that instead of using matrix math, the equations can be coded as a set of general arithmetic expressions, reducing the number of operations and eliminating the added cost of interpreting matrix math. A method of doing this is presented by McCarron, but the code included in an appendix with his report is flawed [16]. A corrected code is included in Appendix A.

3.2.5 Algorithm #5: Madgwick Filter

The fifth is an algorithm that is often referenced in attitude estimation literature. It is known as the Madgwick algorithm, named after Sebastian O. H. Madgwick, the person who proposed it [9]. It takes an approach similar to the Kalman filter but has the advantage of being easier to tune, only having one tunable parameter and shows an improvement in accuracy over the Simple Kalman filter according to literature.

The form of the Madgwick algorithm used in this study uses quaternions to describe the attitude information. A quaternion is a row vector with four elements as shown below.

\[ q = [q_1 \quad q_2 \quad q_3 \quad q_4] \tag{3.21} \]

Quaternions have their own set of arithmetic concepts, some of which are described in Appendix C. The following explanation and equations are arranged in order, to most concisely describe the elements required to make this algorithm function. The publications of Madgwick should be consulted for a more complete explanation and derivations. The equation which yields the attitude estimate when solved for every time step is given below, followed by the supporting equations.

\[ \hat{q}_k = \hat{q}_{k-1} + \left( \frac{1}{2} \hat{q}_{k-1} \otimes \hat{\omega}_k - \beta \frac{\nabla f}{\|\nabla f\|} \right) \Delta t \tag{3.22} \]

\[ \hat{q}_k = \frac{\hat{q}_k}{\|\hat{q}_k\|} \tag{3.23} \]

“\( \hat{q}_k \)” in Equation 3.22 above is the estimated attitude quaternion calculated for each time step. Note that it is immediately normalized before use as an attitude estimate, illustrated in Equation 3.23. An example of how to convert the quaternion to Euler angles is given in Appendix C. The tuning parameter is represented by “\( \beta \)”. The other terms are expanded in the equations.
below for greater transparency. The “\(\otimes\)” symbol is the quaternion multiplication operator. This is explained more completely in Appendix C, but the expanded equations below provide the needed form to reproduce the algorithm.

\[
\frac{1}{2} \hat{q}_{k-1} \otimes \hat{\omega}_k = \frac{1}{2} \begin{bmatrix} q1 & q2 & q3 & q4 \end{bmatrix} \otimes \begin{bmatrix} \omega_{x,k} & \omega_{y,k} & \omega_{z,k} \end{bmatrix} \]
\]

Equations 3.24 through 3.25 give a better explanation of the “\(\frac{1}{2} \hat{q}_{k-1} \otimes \hat{\omega}_k\)” term in Equation 3.22, including the quaternion multiplication. The remaining term is the normalized objective function gradient, “\(\nabla f\)”, and is expanded below.

\[
\nabla f = J^T f = \begin{bmatrix} -2q_3 & 2q_2 & 0 & 2(q_2q_4 - q_1q_3) - a_{x,k} \\ 2q_4 & 2q_1 & -4q_2 & 2(q_1q_2 + q_3q_4) - a_{y,k} \\ -2q_1 & 2q_4 & -4q_3 & 2 \left( \frac{1}{2} - q_2^2 - q_3^2 \right) - a_{z,k} \end{bmatrix} \]

As shown in Equation 3.26, the gradient of the objective function is the transpose of the Jacobian multiplied by the objective function.

It is also important to note that if the initial attitude is such that all three Euler angles equal zero, the quaternion is as given in the equation below. Otherwise, see Appendix C on how to convert from Euler angles to a quaternion.

\[
\hat{q}_0 = [q1 \quad q2 \quad q3 \quad q4] = [1 \quad 0 \quad 0 \quad 0] \]

### 3.2.6 Algorithm #6: Mahony Filter

The Mahony algorithm is very similar to, and predates the Madgwick algorithm [15]. It is included for completeness of this review but is shown in the literature to be less accurate than the Madgwick filter. It has two tunable parameters. This is the algorithm that was included in the source code provided with the IMU chips used in this research.
This algorithm is explained below in a step by step fashion. The first step is to normalize the acceleration vector, illustrated by the equations below.

\[
\tilde{a}_k = \begin{bmatrix} a_{x,k} & a_{y,k} & a_{z,k} \end{bmatrix}
\]

\[
\tilde{a}_k = \frac{\tilde{a}_k}{\|\tilde{a}_k\|}
\]  \hspace{1cm} 3.28 \hspace{1cm} 3.29

Then the estimated direction of the gravity field, “\(v\)”, is calculated with the elements of the quaternion from the previous time step.

\[
\hat{q}_{k-1} = \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix}
\]

\[
v = \begin{bmatrix} 2(q_2 q_4 - q_1 q_3) \\ 2(q_1 q_2 + q_3 q_4) \\ q_1^2 - q_2^2 - q_3^2 + q_4^2 \end{bmatrix}
\]  \hspace{1cm} 3.30 \hspace{1cm} 3.31

The error, “\(e\)”, between the estimated and measured directions of the gravity field is equal to the cross product of the acceleration vector and the estimated direction of the gravity field.

\[
e = \tilde{a}_k \times v
\]  \hspace{1cm} 3.32

\[
E_k = E_{k-1} + e \Delta t
\]  \hspace{1cm} 3.33

Equation 3.32 shows the cross product to calculate “\(e\)” and Equation 3.33 shows how it is integrated to determine “\(E_k\)”, which is then used to adjust the angular rate vector.

\[
\tilde{\omega}_k = \tilde{\omega}_{k-1} + K_p e + K_i E_k
\]  \hspace{1cm} 3.34

Equation 3.34 shows the calculation of the corrected angular rate vector, “\(\tilde{\omega}_k\)”, where “\(K_p\)” is the proportional gain and “\(K_i\)” is the integral gain. These are the algorithms two tunable parameters. The corrected angular rate vector is then used to calculate the rate of change of the attitude quaternion in a way similar to Equation 3.24.

\[
\dot{q} = \frac{1}{2} \hat{q}_{k-1} \otimes \tilde{\omega}_k = \frac{1}{2} \begin{bmatrix} q_1 & q_2 & q_3 & q_4 \end{bmatrix} \otimes \begin{bmatrix} 0 & \omega_{x,k} & \omega_{y,k} & \omega_{z,k} \end{bmatrix}
\]  \hspace{1cm} 3.35

See Equations 3.24 and 3.25 in the Madgwick algorithm section for the expanded form of Equation 3.35. The rate of change of the quaternion, “\(\dot{q}\)”, is then integrated using a familiar form to yield the estimated attitude quaternion for the current time step, “\(\hat{q}_k\)”, which is then immediately normalized, illustrated in the two equations below.

\[
\hat{q}_k = \hat{q}_{k-1} + \dot{q} \Delta t
\]  \hspace{1cm} 3.36
3.2.7 Algorithms #7, #8, and #9: Novel/TIMU Filters

The novel approaches introduced here are different than the previously mentioned algorithms because they require two separate IMU sensors. The basic premise is to use a linear model of the accelerations exerted on each sensor to fuse the measurements from them. An advantageous effect of this is that the noise of the accelerometers cancel out one another to a certain extent. The processed acceleration values are then used in the same manner as Algorithm #2 to calculate a pitch estimate. That description summarizes the first TIMU method, Algorithm #7, introduced here.

The resulting estimate from Algorithm #7 has very low RMS error when compared to the previously explained algorithms, but still suffers from noise similar to Algorithm #2. This will be explained in more detail in the results, but it is mentioned here to provide the motivation behind a modification to this method which then becomes the second novel method, Algorithm #8. The modification is that the mean of the two gyro measurements is used as the derivative of attitude instead of estimating the derivative of the attitude using the first-order backwards difference method. This takes advantage of the model’s superior performance in attitude estimation, illustrated by low RMS error and the gyroscopes superior performance in estimating the attitude derivative (angular rate). However, there is still an issue with gyro drift which will cause greater inaccuracies as the time since calibration increases.

Building on the first TIMU approach, Algorithm #9 simply takes the pitch estimate of Algorithm #7 and fuses it with the mean of the two gyro outputs through a complementary filter, similar to Algorithm #3. The advantage of this is that the pitch estimate is smoothed and therefore the derivative is a much more accurate estimate compared with Algorithm #7 and very close to Algorithm #8. Unfortunately, this results in a small increase in pitch estimate RMS error, but even so, this is the best overall performing method of pitch estimation.

Again, the main distinction between the three TIMU methods and the others described previously is the use of two sensors and a linear model to process the measurements from them. The basis of this model is the equation below that describes the total acceleration on a point of a rigid body as equal to the sum of the linear, tangential, and centripetal accelerations.

\[
a_{total} = a_{linear} + a_{tangential} + a_{centripetal} = a_{linear} + r\alpha + r\omega^2
\]  

Equation 3.38 also relates the tangential and centripetal accelerations to the distance of the point away from the center of rotation, “r”, and the angular acceleration and angular velocity, “\(\alpha\)” and “\(\omega\)” respectively. For a free rigid body, the center of rotation is located at the center of gravity, the CG. The model applies Equation 3.38 to both sensors along the x and z-axes.

\[
a_{total,x0} = r_{x0}\alpha + r_{x0}\omega^2 + a_{lin,x0}
\]  

\[
a_{total,z0} = r_{z0}\alpha + r_{z0}\omega^2 + a_{lin,z0}
\]

\[
\hat{q}_k = \frac{\hat{q}_k}{\|\hat{q}_k\|} 
\]  

3.37
In Equations 3.39 through 3.42 above, the distance between sensor #0 and the CG in the x-axis is annotated by \( r_{x0} \) and in the z-axis is \( r_{z0} \). Naturally, for sensor #1 the distances are annotated by, \( r_{x1} \) and \( r_{z1} \). The convention of subscripts is the same for the linear acceleration terms as well. The angular rate and acceleration are constant along the rigid body and are therefore, the same in each of the 4 equations. Similarly, the linear accelerations not caused by rotation along the rigid body with respect to an external navigation frame are equal at all points along the rigid body. This leads to the next two equations.

\[
\begin{align*}
0 &= a_{\text{lin},x0} - a_{\text{lin},x1} \\
0 &= a_{\text{lin},z0} - a_{\text{lin},z1}
\end{align*}
\]

Equations 3.43 and 3.44 in addition to Equations 3.39 through 3.42 provide enough equations to assemble a linear system of a form shown below.

\[
[b] = [A][x]
\]

This matrix equation can be solved at every time step to calculate the “x” matrix. Solving the equation is not trivial, especially when program complexity and computational cost are considered. This is addressed again later in this document and a method to solve the equation is clearly presented. The entire equation with all terms for each matrix is given below.

\[
\begin{bmatrix}
a_{\text{total},x0} \\
a_{\text{total},z0} \\
a_{\text{total},x1} \\
a_{\text{total},z1} \\
0 \\
0
\end{bmatrix} =
\begin{bmatrix}
r_{z0} & -r_{x0} & 1 & 0 & 0 & 0 \\
r_{x0} & r_{z0} & 0 & 1 & 0 & 0 \\
r_{z1} & -r_{x1} & 0 & 0 & 1 & 0 \\
r_{x1} & r_{z1} & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & -1 & 0 \\
0 & 0 & 0 & 1 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
\alpha_y \\
\omega_y^2 \\
a_{\text{lin},x0} \\
a_{\text{lin},z0} \\
a_{\text{lin},x1} \\
a_{\text{lin},z1}
\end{bmatrix}
\]

Once the equation is solved, the linear accelerations are used to calculate a pitch estimate using the same form as Equation 3.7 from Algorithm #2. Note that since the linear accelerations are equal at all points on the rigid body as per the model, it does not matter whether the terms representing the accelerations at sensor #0 or #1 are used, they are equal. The equation below illustrates this fact.

\[
\theta_k = atan2(a_{\text{lin},x0,k}, -a_{\text{lin},z0,k}) = atan2(a_{\text{lin},x1,k}, -a_{\text{lin},z1,k})
\]

As mentioned before, the resulting pitch estimate has a very low RMS error but is noisy, leading to a high RMS error when considering the derivative of the estimate. This is an important
consideration because it indicates that the derivative of the estimate, which is usually required by the controller, is not accurate. The gyro measurements from the sensors are fairly accurate when compared with the derivative of the ground truth measurements. This leads to Algorithm #8, where the gyro measurements are averaged and substituted as the derivative of the pitch estimate. This can easily be accomplished in practice by sending the averaged angular rate measurements to the controller as opposed to using finite differences to calculate it from the pitch estimate, as is commonly done.

This modification seems to elevate the problem with the derivative, however, the pitch estimate is still noisy, which is a major drawback. For example, if Algorithms #7 or #8 are used in conjunction with a PID controller, the proportional term of the controller is exposed to a rapidly changing value of the process variable whose average over time is accurate but is very inaccurate when considering each individual time step. Algorithm #9 addresses this issue by taking advantage of the complementary filter from Algorithm #3. The linear accelerations calculated with Equation 3.46 are used in an equation very similar to Equations 3.8 where the mean of the two gyro measurements is substituted for the appropriate term as illustrated below. Algorithm #9 also decreases the negative effects of gyro drift.

\[
\theta_k = K_c(\theta_{k-1} + \omega_{y,k}\Delta t) + (1 - K_c)\text{atan2}(a_{lin,x0,k}, -a_{lin,z0,k})
\]  

\[
\omega_{y,k} = \frac{\omega_{y0,k} + \omega_{y1,k}}{2}
\]

Solving Equation 3.46 is not trivial, but there are ways to simplify the process to make it easier to code for the purposes of attitude estimation. The MATLAB symbolic toolbox is used to symbolically solve it, reducing the terms and operations in a fashion similar to the Kalman filter simplification presented by McCarron [16]. The simplification for the TIMU algorithm is given in Appendix B. This makes the algorithm much easier to code and more efficient in terms of computational cost.

### 3.3 Experimental Apparatus and Data Processing

In order to compare the selected attitude reference algorithms as directly as possible, one set of IMU data is collected and then the algorithms are individually used to process the same data. Two IMU chips are positioned on a seesaw-like test stand consisting of an arm which rotates around a central axis.
The two IMUs can be mounted at different locations and heights along the seesaw. Changing the mounting locations creates a difference in the linear speeds and accelerations exerted on the two IMUs while the angular speeds and accelerations for both always stay the same. The location of the fulcrum is analogous to the location of the center of gravity of an aircraft. The notion behind this test stand is to simulate putting the IMUs at different distances from the CG of an aircraft. Only one distance from the center of rotation for each IMU is examined by this investigation, but this test stand can be used to examine more IMU positions in future research.

The fulcrum of the seesaw is assembled with small ball bearings to make the motion of the seesaw as smooth as possible. A CUI Inc. AMT112Q incremental encoder is attached to the fulcrum shaft which provides a ground truth angle measurement for the arm.
The IMUs are both InvenSense MPU9255 which are installed on a small circuit board with a pressure sensor and sold as the Waveshare brand “10 DOF IMU Sensor (B)”. An Arduino Mega 2560 prototyping board is used to power the IMUs and as an interface through which the data can be collected, first over I²C protocol from the IMUs and then over serial protocol to a PC where the data is stored.
Software is written for the Arduino to properly configure the IMUs for the data collection and then to take the data and send it to the PC as raw numerical values. Setting up the IMUs for data collection is accomplished by taking advantage of the “Wire” library which is included with the Arduino software. The functions needed are written based on online information regarding the subject, where forum users posted their experiences in doing similar things. An I2C multiplexer is used to alternate interrogating between the IMUs for data. This is required because both IMUs use the same register address, so without the multiplexer, a command from the Arduino would be received by both IMUs simultaneously and indiscriminately.

The Boost C++ library is used to create a PC terminal program that is similar to the Arduino IDE Serial Monitor which records the data coming from the serial port and saves it to a file. The raw data leaves the Arduino in the form of lines containing 23 numerical values. These values included the accelerometer, gyro, and magnetometer readings in all three axes as well as the temperature for both IMUs. It also includes the encoder position for ground truth, the change in time between each reading of the data in microseconds and the total time since recording data begins. The recorded data is in the form of a text file where each line contains one complete sample of the 23 floating point data values so that a time series for each of the values can be extracted through post-processing. An excerpt of the recorded data is given below. It is the first five lines of logged data after recording began.
Before recording of the data starts, the Arduino collects data for four seconds and then averages the data to determine offset values to compensate for accelerometer and gyro bias. This is written in the Arduino software as a calibration function. These offset values are recorded and later applied to the raw data during processing to compensate for bias.

The data is recorded after the IMUs were given a chance to warm up, which requires about 10 minutes. This is to mitigate the effects of thermal drift on the data. When the program that records the IMU data is started, the seesaw is held motionless in a horizontal position with a locking pin until the offset values are calculated. Then the locking pin is removed and the seesaw is manipulated by hand to simulate a series of rapid and slow angular motions between approximately ±60 degrees.

The recorded data is processed using MATLAB scripts in which all of the attitude algorithms are coded. MATLAB is chosen for this purpose because it simplifies the comparison in many ways. If these tests were done using processing on-board the Arduino for all algorithms simultaneously, it is possible that the extra computational effort would make processing and logging at a reasonable sample rate impossible. The sample rate of 100 Hz was chosen because it is commonly used for control of aerial vehicles and it is likely that any of the algorithms by themselves could be used in the Arduino software for on-board processing at this sample rate. Using MATLAB also has the added benefit of easier coding as well as built-in tools for evaluating results and the computational cost of each algorithm.

The raw data from the data logging text files is saved as a MATLAB MAT-file (.mat extension) which includes a matrix of all the data and offset values calculated during the calibration period before data recording begins. A single MATLAB script processes the data and displays results by first applying offsets, next applying coefficients to make it dimensional, and finally converting the axes to agree with the coordinate systems shown in Figure 3.3. The result is a set of matrices that include 3-D accelerometer measurements in G’s, 3-D gyro measurements in degrees per second, and a vector of temperature in degrees Celsius for both IMUs. It also produces a vector of the sample times in seconds and the accumulated sample times or the time since the beginning of recording at each time step in seconds. Lastly, a vector of the position of the seesaw measured by the encoder in degrees, which is the ground truth pitch angle.

After the initial steps mentioned above, the script generates the pitch estimates calculated with each of the evaluated algorithms by calling functions containing each algorithm. Separate functions calculate the RMS errors for each estimate and lastly, the errors are organized and plotted in order to facilitate evaluation.

The evaluation is done by first constructing four vectors. The first contains the RMS pitch estimate error values for each algorithm after processing the data from IMU #0. The second contains the RMS pitch errors from IMU #1. The third contains the RMS pitch rate estimate errors.
from IMU #0. Lastly, the forth is the RMS pitch rate errors for IMU #1. Comparing the first and second vectors provides insight into how the distance between the IMUs and the CG affects pitch error for the different algorithms. The error for a given algorithm is compared to the error for the same algorithm using the other IMU. Similarly, comparing the third and fourth vectors shows the differences in pitch rate errors.

Comparing the errors from a given algorithm with that of another algorithm and incorporating the effects of the IMU locations, requires further processing. Each of the vectors are normalized by dividing each element by the sum of the vector elements.

$$V_i = \frac{V_i^-}{\sum_{i=1}^{n} V_i^-}$$  

“$$V_i^-$$” in Equation 3.50 above represents the i’th element in the un-normalized “$$V^-$$” vector. The i’th element of the normalized vector is represented by “$$V_i$$”. “$$n$$” is the number of elements in the vector. The normalized vectors for IMU #0, one being RMS pitch error, and the other RMS pitch rate error, are summed to form a composite error vector. The same is done for IMU #1. Then, the elements of one of the composite vectors are averaged with the corresponding elements of the other, to form a single vector that compares the errors of each algorithm considering both IMU locations, pitch, and pitch rate errors. This method provides an error index with which to compare the algorithms that considers their pitch and pitch rate errors equally, and essentially averages the difference in errors from the two IMUs to make the comparison as direct as possible. At each step described above, the resulting vectors are displayed in bar graphs and a table for evaluation throughout the process of arriving at the final error index, as well as for better understanding of the errors contributing to the final index.

An important consideration of this research is the fact that five of the evaluated algorithms have tuning variables that alter performance. The respective literature for the algorithms give descriptions of the tuning variables, how they are derived and suggested starting values based on certain factors. This makes it difficult to directly compare the algorithms since they are tuned differently and based on different things. A consistent method was followed to tune each algorithm in order to make the results as directly comparable as possible. This method entails using the MATLAB Optimization App (MATLAB Version: R2018a) to find the tuning variables that minimize the errors for each algorithm so that the error indices represent the best possible performance from each algorithm for comparison. This was accomplished by going through each algorithm and first finding the tuning variable(s) that yield the least RMS pitch error and recording that error value. Then the tuning variable(s) for the best RMS pitch rate error are determined and the error value recoded. The cost function is altered so that its value is the sum of the pitch and pitch rate errors after dividing each by their respective best values determined previously. Therefore, the cost value that the optimizer is minimizing is the sum of the pitch and pitch rate errors as percentages of their respective best possible values for the given algorithm. The intent of this scheme is to minimize the pitch and pitch rate errors with equal consideration, so that the composite error index described earlier is the lowest possible for each algorithm and the overall best possible performance of each algorithm is compared. The minimization solver is set to “fmincon” constrained non-linear optimizer and the algorithm to “interior point”. All of the
stopping criteria are left at the default values. The bounds and starting values used for the optimization are shown in the table below.

Table 3.1: The upper/lower bounds and starting values used in the optimizations of tuning parameters for the algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Starting Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complementary</td>
<td>Kc =0.0001</td>
<td>Kc =1</td>
<td>Kc =0.99</td>
</tr>
<tr>
<td>Kalman</td>
<td>Q4 =1e-6</td>
<td>Q4 =1</td>
<td>Q4 =1e5</td>
</tr>
<tr>
<td></td>
<td>R4 =1</td>
<td>R4 =1e12</td>
<td>R4 =10</td>
</tr>
<tr>
<td>Madgwick</td>
<td>Beta =1e12</td>
<td>Beta =0.0605</td>
<td>Beta =500</td>
</tr>
<tr>
<td>Mahony</td>
<td>Kp =1e12</td>
<td>Kp =100</td>
<td>Kp =0.5</td>
</tr>
<tr>
<td></td>
<td>Ki =1e12</td>
<td>Ki =100</td>
<td>Ki =1</td>
</tr>
<tr>
<td>Novel, #8</td>
<td>Kc =0.0001</td>
<td>Kc =1</td>
<td>Kc =0.95</td>
</tr>
</tbody>
</table>

The optimum error values and the associated tuning variables are shown in the table below.

Table 3.2: The tuning values and associated errors found using the optimizer for the different algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Tuning Variable(s) for Min. Pitch Error</th>
<th>Min. Pitch Error (deg.)</th>
<th>Tuning Variable(s) for Min. Pitch Rate Error</th>
<th>Min. Pitch Rate Error (dps)</th>
<th>Final Tuning Variable(s) for Min. Combined Error</th>
<th>Final Pitch Error (deg.)</th>
<th>Final Pitch Rate Error (dps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complementary</td>
<td>Kc =0.986097290647375</td>
<td>3.1095</td>
<td>Kc =0.999999999859923</td>
<td>9.8652</td>
<td>Kc =0.99994155873211</td>
<td>3.2584</td>
<td>9.8662</td>
</tr>
<tr>
<td>Kalman</td>
<td>Q4 =1.033970159894913E-6</td>
<td>3.1122</td>
<td>Q4 =8.04508716418936E-6</td>
<td>8.7774</td>
<td>Q4 =6.654010708978304</td>
<td>3.1123</td>
<td>8.8151</td>
</tr>
<tr>
<td></td>
<td>R4 =38682.3331966906</td>
<td></td>
<td>R4 =1.8707915976844532E10</td>
<td></td>
<td>R4 =1.73243766716539E10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Madgwick</td>
<td>Beta =0.092447994175758</td>
<td>2.8813</td>
<td>Beta =1.27716709596870E-7</td>
<td>9.8823</td>
<td>Beta =4.517795369958336E-7</td>
<td>3.2656</td>
<td>9.8823</td>
</tr>
<tr>
<td></td>
<td>Ki =2.4689521467601926E-8</td>
<td></td>
<td>Ki =1.2689020644184956E-6</td>
<td></td>
<td>Ki =6.39510754736633E-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Novel, #9</td>
<td>Kc =0.5087849803036555</td>
<td>0.3178</td>
<td>Kc =0.9347119332144092</td>
<td>8.2012</td>
<td>Kc =0.8218959581737024</td>
<td>0.9041</td>
<td>12.9201</td>
</tr>
</tbody>
</table>

The tuning variables listed in Table 3.2 above as the “Final Tuning Variables for Minimum Combined Error” are used in the data processing MATLAB script to determine the results for comparison.

Two of the algorithms, the Kalman filter and the Mahony filter, have variables that carry over from one time step to the next, not including the variables containing the attitude information. For example, at any time step except for the first, the Kalman filter requires the pitch from the previous time step and the elements of the covariance matrix or “P” matrix shown in Equation 3.20. Conversely, all of the other algorithms except for the Mahony filter, only carry the attitude information from one time step to the next, or do not carry any information at all. As mentioned earlier in the section introducing the Kalman filter, the “P” matrix starts at initial values and then converges to values based on the sensor error covariance matrices and the Kalman gain matrix. It is thought that the filter is not outputting its best estimate while the “P” matrix values are converging, which decreases its performance compared to the other algorithms and makes the comparisons less direct. In order to mitigate this, the “P” matrix is initially set to the values on which it converges. This is accomplished by including a loop in the Kalman filter processing function preceding the loop that determines the attitude estimate that allows the “P” matrix to
converge. Then the converged values are used as the initial values for the pitch estimation loop. The plots below illustrate the convergence.

![Figure 3.9: The left plot shows the convergence of a “P” matrix value without pre-convergence. The right plot shows the same value with pre-convergence. Note the scale of the dependant axes.](image)

The plot on the left in Figure 3.9 above shows the value of the first element in the “P” matrix as it converges during a loop that contained ten times the number of time steps as the data set. This ensures that the values of the “P” matrix are well converged by the time the estimation loop happens, as illustrated by the plot on the right in Figure 3.9 showing the very small variation in the value during the calculation of the pitch estimate. This is representative of the other elements of the “P” matrix. Finding a converged covariance matrix in this way is only possible because feedback from the estimate or measurements is not part of the covariance matrix calculation. If they were, there would need to be enough time series measurement data for the values to converge. This would likely require a time series ten times that used in the experiment based on the results of the experiment.

The Mahony filter carries a vector of three accumulated errors from one time step to the next represented by the “E_k” from Equation 3.33. This is essentially the error between the measured and the estimated direction of the gravity field, integrated over time. Since these integral values are calculated based on the actual measurements compared to the estimates, they cannot be pre-converged like the covariance matrix in the Kalman filter. In other words, feedback from the measurements is part of the integral of errors vector calculation. The plot below shows the values of the vector.
Figure 3.10: The change in the values of “E” for the Mahony algorithm during calculation of pitch estimate.

If the data time series was longer, it may be possible to see the integral values stabilize more than what is shown in Figure 3.10 above. Ideally, if there was more data, they will stop trending up or down as the measurements and estimates start to agree more consistently.

Since the computational cost of the different algorithms is a part of the comparison, the MATLAB Profiler is used to determine how long each algorithm takes to compute an estimate in order to qualitatively compare the computational cost of each algorithm. As mentioned before, each algorithm is coded into a function or nested functions that are called by a single MATLAB script. The function for each algorithm is called twice since an estimate for each IMU is calculated. The Novel algorithms were also called twice to make the timing comparison easier to interpret. Since the functions are coded with MATLAB, the relative computational cost of the algorithms may vary once coded into whichever language is used for a practical attitude estimation system. But, when the algorithms were coded or modified from other code for this research, they were coded in a way that was more like the C language, and some of the MATLAB built-in features and vectorization were avoided. This is not strictly true of the code, but the algorithms are generally the same in terms of coding style and relative number of operations.
CHAPTER 4: RESULTS

In order to evaluate how well the Arduino code programmatically provided a sample rate of 100Hz, the mean sample time from the data set is calculated along with the minimum and maximum sample times. They are given in the chart below.

Table 4.1: Characteristic of the sample times from the recorded data.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min. Sample Time (sec)</td>
<td>0.0104</td>
</tr>
<tr>
<td>Max. Sample Time (sec)</td>
<td>0.0106</td>
</tr>
<tr>
<td>Mean Sample Time (sec)</td>
<td>0.0105</td>
</tr>
<tr>
<td>Mean Sample Rate (Hz)</td>
<td>95.4733</td>
</tr>
</tbody>
</table>

Table 4.1 shows that the sample rate was close to but not exactly equal to the target sample rate of 100Hz. This may be due to inaccuracies in the micro-controller, or delay in the way the timing is programmed. Either way, the coefficient used in the Arduino code to tune the sample time can be adjusted by trial and error to make the sample rate closer to desired.

The seesaw was manipulated by hand following the short calibration period. The ground truth measurements from the encoder are shown below.

Figure 4.1: Ground truth pitch from the recorded data.
The plot in Figure 4.1 shows the results of the entire time series of encoder measurements from the experiment data set which lasted for almost 30 seconds. The measurements begin immediately following the calibration period and show the removal of the locking pin, labeled on the plot above as well as the re-installation of the pin. This ensured that the seesaw was held stationary before and after the test to determine how well the algorithms followed the ground truth from and back to near zero pitch. The motions start with a pitch down movement to an angle close to -55 degrees and a pause at that position for about one second. This is followed by a motion to an angle close to 65 degrees with a pause of approximately 0.6 seconds. Then there are six rapid pitching motions in succession that cover almost the entire range of motion. These are followed by another six rapid motions that are within the ±20 degrees envelope. Then the locking pin is reinserted and the data is recoded for approximately 5 seconds without motion. The pitch rate is plotted in a similar fashion below.

![Figure 4.2: Ground truth pitch rate from the recorded data.](image)

The pitch rate of the ground truth shown above in Figure 4.2 was calculated using the first-order backwards finite difference equation, given as Equation 1.1 earlier. It shows that during the large rapid motions, the pitch rate was as high as 290 degrees per second and fluctuated during periods where the motion was supposed to be still, but the locking pin was not installed.

### 4.1 Results: Pitch Estimates

Plotting all of the estimates together with the ground truth makes the differences between the individual estimates and the ground truth difficult to observe. For this reason regions of the
combined plot are assigned and zoomed-in views of the different regions are provided to make the differences between the estimates visible.

The reader is reminded that there are two distinct estimates provided by each of the compared algorithms except for the three novel algorithms, because there is a separate estimate calculated for both of the IMUs. The IMUs are labeled #0 and #1 because of the Arduino/C++ programming convention and the following figures are labeled as such. IMU #0 is the IMU farther from the fulcrum as labeled in Figure 3.3. In the plots to follow, there is not a curve for the Novel Modified algorithm, #8, because its pitch estimate is the same as the first Novel algorithm, #7, and its pitch rate measurements are the same as the Unfiltered Gyro algorithm, #1. The plot below shows all of the estimates calculated with measurements from IMU #0, plotted together with the ground truth.

![Graph showing pitch estimates and ground truth](image)

**Figure 4.3**: All of the pitch estimates from IMU #0 data and ground truth plotted together with labeled regions of interest.

Figure 4.3 above shows all of the estimates based on IMU #0 plotted together. Since the three novel algorithm estimates are based on both IMUs, their estimates are identical, but are still presented on all plots along with the other estimates. The regions that are shown zoomed-in below are lettered A through E and those locations are labeled in Figure 4.3 above. Region A shows the
estimates shortly after the data recording starts and before the locking pin is removed from the seesaw.

Figure 4.4: Zoomed in view of region A from Figure 4.3.

Figure 4.4 shows the zoomed-in view of region A for the IMU #0 data. The Unfiltered Accelerometer algorithm, #2, is noticeably the noisiest, followed by the #7 Novel algorithm. The #9 algorithm is less noisy than #2 or #7, but does appear to be noisier than the remaining algorithms that are tracking in strait lines with the ground truth. The Kalman filter is standing out amongst them by tracking above the rest and the ground truth.
Figure 4.5: Zoomed in view of region A from Figure 4.3.

Figure 4.5 zooms in further to provide a better view of the separation between the ground truth and the estimates.
Figure 4.6: Zoomed in view of region A from Figure 4.3.

Figure 4.6 zooms in even farther on the tight grouping that includes the Unfiltered Gyro, Complementary, Madgwick, and Mahony algorithms. Zoomed views of region B are given below.
The high noise of the #2 algorithm compared to the rest is very apparent in Figure 4.7. At the peak of the pitching motion, where the pitch is held constant for about half of a second, the #7 and #9 algorithms track very closely with the ground truth, much closer than the other algorithms which are indicating a pitch angle approximately eight degrees lower than ground truth. The #9 seems to not only be accurate, but is relatively smooth. A zoomed in view of the #1, #3, #4, #5 and #6 estimates is shown below.
Figure 4.8: Zoomed in view of region B from Figure 4.3.

Zooming in on the #1, #3, #4, #5 and #6 in Figure 4.8 shows that the #4 is above and slightly closer to the ground truth than the rest.
Figure 4.9 shows that the #1 and #3 algorithms are tracking together very closely, as are the #5 and #6.
Moving on to region C, Figure 4.10 shows the same basic trends as region B except that the #4 algorithms seems to indicate a pitch attitude slightly further from ground truth compared to the closely grouped #1, #3, #5 and #6.
Figure 4.11: Zoomed in view of region C from Figure 4.3.

Figure 4.11 shows a growing gap between #1 and the group of #3, #5, and #6. This gap continues to grow in the following plots.
Region D shows how closely the #7 and #9 follow along with the ground truth. The group of #1, #3, #5 and #6 are not far from the ground truth either while #4 is noticeably indicating a higher pitch than the rest. The #2 is, of course, varying wildly around the ground truth but its mean value is tracking accurately.
The zoomed-in view of region E from Figure 4.13 shows that after re-installation of the locking pin, the encoder is indicating the pitch approximately 0.4 degrees higher than the beginning of the data set. The #2, #7, #9 as well as the #4 algorithms seem to have followed along well with the ground truth to conclude the measurements. The grouping of the #1, #3, #5 and #6 are indicating a pitch close to 0.5 degrees lower than ground truth.
Figure 4.14 shows that toward the end of the data set, the #1, #3, #5, and #6 algorithms are showing a little more separation. The #1 is farthest from ground truth, with the tightly grouped #5 and #6 close by. The #3 is the closest to the ground truth of the group. Next, the results of the estimates from the IMU #1 data is presented.
Figure 4.15: All of the pitch estimates from IMU #1 data and ground truth plotted together with labeled regions of interest.

Figure 4.15 shows the results of plotting the estimates calculated using the data from IMU #1 along with the ground truth and Novel algorithms. Observing Figure 4.15 shows a significant reduction in the noisiness of the #2 estimate. Zoomed in views of some of the regions are presented to gain further insights.
Figure 4.16: Zoomed in view of region B from Figure 4.15.

Figure 4.16 showing region B, is very similar to Figure 4.7 except that noise from algorithm #2 is vastly less pronounced. The gap between #4 and the group of #1, #3, #5 and #6 is slightly more apparent but otherwise everything is about the same as the IMU #0 estimates. Investigating the C and D regions shows the same trends as this when compared to the IMU #0 estimates so plots of those regions are not presented. But, region E is shown below.
4.2 Results: Pitch Rate Estimates

Next, the pitch rate estimates are examined for their agreement with the ground truth pitch rate calculated using Equation 1.1.
Figure 4.18: All of the pitch rate estimates from IMU #0 data and ground truth plotted together.

When comparing Figure 4.18 above with Figure 4.2 that only shows the ground truth, it is clear that the noise of the #2 estimate overwhelms the plot estimating angular rates of up to 8000 degrees per second. This makes the plot ineffective for examining the rest of the estimates so the #2 estimate is removed.
Figure 4.19: All of the pitch rate estimates from IMU #0 data and ground truth except for algorithm #2, plotted together.

With the #2 estimate removed, Figure 4.19 shows a much more defined plot of the estimates compared to Figure 4.18, but it is overwhelmed by the #7 estimate. Therefore, the #7 estimate is also removed for better examination of the others.
Figure 4.20 shows that with the #2 and #7 estimates removed, the plot of the pitch rate estimates is comparable to Figure 4.2. The #9 estimate is noticeably noisier than the rest indicated by the peaks and jittery regions along the curve. The three regions labeled A, B and C are investigated with zoomed-in views below.
Figure 4.21 shows the zoomed-in view of region A. The mean of the #9 algorithm seems to follow the ground truth very well, but the noise is very apparent. Jitters in the ground truth curve are noticeable while the #1, #3, #4, #5, and #6 estimates are very smooth with a significant bias. The #4 stand out from the rest while the #1, #3, #5 and #6 are in a tight group similar to what was shown in the pitch estimate plots.
Zooming in further, Figure 4.22 shows a small delay between the #4 estimate and the others, as if the estimate is shifted slightly. At the sample rate of approximately 100Hz it seems that this shift is equal to one cycle.
Figure 4.23: Zoomed in view of region A from Figure 4.20.

Figure 4.23 zooms-in even more, to show the close relation between the #1, #3, #5 and #6 estimates. The #1 and #3 are close to one another but distinguishable in the plot while the #5 and #6 are so close that they are indistinguishable.
Figure 4.24 shows region B. The same trends form region A persist. Except for the noisiness of the #9 algorithm, all of the estimates track the ground truth very well through this section of rapid oscillations.
Figure 4.25: Zoomed in view of region C from Figure 4.20.

Figure 4.25 provides an illustration of the noise magnitude and frequency of estimate #9. The rest of the estimates are tracking the ground truth along zero much more smoothly.
Figure 4.26 shows the noise of the other estimates along the ground truth curve that is horizontal along zero. The shift in the #4 compared to the other group is still apparent.

The pitch rate estimates calculated using the data from IMU #1 are investigated next.
Figure 4.27: All of the pitch rate estimates from IMU #1 data and ground truth plotted together.

The pitch rate estimates from the IMU #1 data, plotted together in Figure 4.27 above, are similar to Figure 4.18 in that the #2 estimate overwhelms the plot. But, it does so to a lesser extent. The peak angular rate in Figure 4.27 is -2000 degrees per second compared to the 8000 shown in Figure 4.18. In fact, the peaks shown in Figure 4.27 are not much different than those shown in Figure 4.19 where the #7 estimate is shown to overwhelm the plot after the #2 is removed. Since the #7 and #9 estimate curves are the same for both sets of IMU data, it will be redundant to show plots where only the #2 is removed or the #7 removed similar to Figure 4.19 and Figure 4.20 respectively. Investigations into the plots of the estimates for IMU #1 data, give no more insights than the those of IMU #0 shown in Figure 4.18 through Figure 4.26, so no additional pitch rate plots are presented.

4.3 Results: Numerical Pitch and Pitch Rate Errors

The errors between the estimate curves and the ground truth curves are calculated in order to quantify algorithm performance. Results are shown below from calculating the RMS pitch and RMS pitch rate errors for all of the estimates.
Table 4.2: The RMS pitch errors and RMS pitch rate errors from each algorithm for IMU #0 and IMU #1 measurements.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>RMS Pitch Error, IMU #0 (deg)</th>
<th>RMS Pitch Error, IMU #1 (deg)</th>
<th>RMS Pitch Rate Error, IMU #0 (dps)</th>
<th>RMS Pitch Rate Error, IMU #1 (dps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Gyro Only</td>
<td>3.1937</td>
<td>3.3238</td>
<td>9.7534</td>
<td>9.9769</td>
</tr>
<tr>
<td>2 Accel Only</td>
<td>6.4701</td>
<td>1.7900</td>
<td>325.4153</td>
<td>98.7302</td>
</tr>
<tr>
<td>3 Complementary</td>
<td>3.1931</td>
<td>3.3237</td>
<td>9.7551</td>
<td>9.9774</td>
</tr>
<tr>
<td>6 Mahony IMU</td>
<td>3.1949</td>
<td>3.3363</td>
<td>9.7594</td>
<td>10.0053</td>
</tr>
<tr>
<td>7 Novel</td>
<td>0.8290</td>
<td>0.8290</td>
<td>88.3125</td>
<td>88.3125</td>
</tr>
<tr>
<td>8 Novel Modified</td>
<td>0.8290</td>
<td>0.8290</td>
<td>8.7681</td>
<td>8.7681</td>
</tr>
<tr>
<td>9 Novel with Comp. Filter</td>
<td>0.9041</td>
<td>0.9041</td>
<td>12.9201</td>
<td>12.9201</td>
</tr>
</tbody>
</table>

Table 4.2 above lists all of the RMS pitch and pitch rate errors when the estimates are compared to the ground truth. The RMS pitch error column for IMU #0 shows that the #1, #3, #4, #5 and #6 algorithms have very close to the same reported errors with the #4 showing the lowest error of those by a small margin. The #2 algorithm provides significantly larger error than the others. The three novel algorithms provide by far, the lowest errors. Algorithm #9, the Novel with Complementary Filter, has slightly higher error than #7 and #8. #7 and #8 are equal to one another because as stated in the algorithm descriptions, algorithm #7 and #8 are the same in terms of pitch estimate, but #8 uses the mean pitch rate estimate between the two IMUs from the #1 algorithm.

The RMS pitch error column for IMU #1 in Table 4.2 is similar to the column for IMU #0 except for the fact that the #2 algorithm yields less error than all but the Novel algorithms as opposed to showing the worst performance in the previous column. The Novel algorithm errors are identical to the previous column, because as mentioned before, there is only one estimate provided from these with two IMUs and the errors are reproduced in both columns for easier comparison of results.

The RMS pitch rate error column in Table 4.2 for IMU #0 shows that again the #1, #3, #4, #5, and #6 algorithms are very close in performance, with the #4 filter performing slightly better than the rest. In fact, for IMU #0, the #4 algorithm provided the lowest pitch rate error of all the methods including the novel algorithms. It was followed most closely by #8 that takes the mean of the two gyro measurements. The #2 estimate gave error many times greater than most. The #7 algorithm also gave extremely high error, but not as high as the #2.

The column for RMS pitch rate error from IMU #1 is similar except for the drastic decrease in error of the #2 algorithm. Even so, that error is still far greater than the rest, save for #7 that is still less erroneous but by a small amount. In contrast to the previous column, the #8 error is the lowest followed closely by the #4.

As mentioned earlier, further processing of the error vectors presented above is accomplished through normalizing them by dividing their elements by the sum of their elements. Then the normalized pitch error vectors are added to the normalized pitch rate error vectors in
order to better compare the algorithms, giving equal consideration to both types of errors. The results of these operations are presented below.

Table 4.3: The sum of normalized RMS and RMS pitch rate errors for both IMU #0 and #1 and their mean, the error index.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Sum of Normalized RMS Errors, IMU #0</th>
<th>Sum of Normalized RMS Errors, IMU #1</th>
<th>Mean Sum of Normalized RMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Gyro Only</td>
<td>0.1488</td>
<td>0.1979</td>
<td>0.1734</td>
</tr>
<tr>
<td>2 Accel Only</td>
<td>0.9342</td>
<td>0.4689</td>
<td>0.7015</td>
</tr>
<tr>
<td>3 Complementary</td>
<td>0.1488</td>
<td>0.1980</td>
<td>0.1734</td>
</tr>
<tr>
<td>4 Kalman</td>
<td>0.1396</td>
<td>0.1883</td>
<td>0.1640</td>
</tr>
<tr>
<td>5 Madgwick IMU</td>
<td>0.1489</td>
<td>0.1987</td>
<td>0.1738</td>
</tr>
<tr>
<td>6 Mahony IMU</td>
<td>0.1489</td>
<td>0.1987</td>
<td>0.1738</td>
</tr>
<tr>
<td>7 Novel</td>
<td>0.2162</td>
<td>0.3824</td>
<td>0.2993</td>
</tr>
<tr>
<td>8 Novel Modified</td>
<td>0.0515</td>
<td>0.0737</td>
<td>0.0626</td>
</tr>
<tr>
<td>9 Novel with Comp. Filter</td>
<td>0.0632</td>
<td>0.0935</td>
<td>0.0783</td>
</tr>
</tbody>
</table>

The first column of Table 4.3 lists the errors resulting from adding the normalized pitch errors and the normalized pitch rate errors from the IMU #0 measurements together. The value for the #8 algorithm is the lowest followed closely by the #9. These values are presented below in a bar graph.

![Figure 4.28: Bar graph showing the sum of normalized RMS pitch and RMS pitch rate errors for IMU #0.](image-url)
Figure 4.28 clearly illustrates that the #8 and #9 provide the lowest error index considering both pitch and pitch rate errors. The #2 algorithm is by far the most erroneous, and the #7 which is basically the #2 algorithm using the measurements from both IMUs, is the second most erroneous but is not much worse than the majority of the others.

The next column in Table 4.3 shows the values resulting from the IMU #1 measurements. These are very similar to the previous column except that the #2 algorithm values are much lower than those in the IMU #0 column. Note that the actual values for the rest of the algorithms are greater than the previous set but this is a result of the normalization step and not a reflection of the actual dimensional errors given in Table 4.2. The bar graph below illustrates the normalized and summed errors.

![Bar graph showing the sum of normalized RMS pitch and RMS pitch rate errors for IMU #1.](image)

Figure 4.29 shows the same basic trends as Figure 4.28 except that the #2 algorithm errors are much closer to the rest compared to the IMU #0, which mirrors the results given in Table 4.2 where the IMU #1 errors for the #2 algorithm are much lower than those form IMU #0. The bar graph also provides a good visual representation of how the #8 and #9 algorithms are much less erroneous compared to the others with the #8 being slightly better.

The last column in Table 4.3 gives the mean between the IMU #0 and IMU #1 normalized then errors. Since the trends from the first two columns are similar, the mean is predictably a clear average between the two as illustrated in the bar graph below.
The purpose behind Figure 4.30 and its corresponding column in Table 4.3 is to average the results from the two IMUs so that the compared algorithms that only use data from one IMU at a time are compared more directly with the Novel algorithms which require both IMUs. This is used as the overall error index for this research. It shows that the best performing algorithm is the #8 followed closely by the #9. The #1, #3, #4, #5 and #6 are all very close to one another with the #4 being the best of that group. The #7 algorithm has an error index approximately 83% more than the #4, and the #2 has an index approximately 134% greater than the #7.

4.4 Results: Algorithm Run Times

As mentioned before, the MATLAB profiler is used to provide a qualitative comparison of the algorithms in terms of computational cost. The averaged run time for each algorithms’ function is presented in the table below.
Table 4.4: Average run times of algorithm functions recorded by MATLAB Profiler.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Average Run Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Gyro Only</td>
<td>5</td>
</tr>
<tr>
<td>2 Accel Only</td>
<td>2</td>
</tr>
<tr>
<td>3 Complementary</td>
<td>3</td>
</tr>
<tr>
<td>4 Kalman</td>
<td>8</td>
</tr>
<tr>
<td>5 Madgwick IMU</td>
<td>144</td>
</tr>
<tr>
<td>6 Mahony IMU</td>
<td>236</td>
</tr>
<tr>
<td>7 Novel</td>
<td>6</td>
</tr>
<tr>
<td>8 Novel Modified</td>
<td>6</td>
</tr>
<tr>
<td>9 Novel with Comp. Filter</td>
<td>8</td>
</tr>
</tbody>
</table>

The units in Table 4.4 above are milliseconds, and these totals represent the time taken to process the entire set of time series data into a pitch estimate. Again, these values are meant to give a relative comparison, not quantitative speed of the algorithms. The table shows that most of the algorithms take 8 ms or less to run except for the Madgwick and Mahony algorithms that take 144 ms and 236 ms respectively. The Novel algorithms do take slightly longer to run than algorithms #1 through #3, but the difference is small. They take the same amount of time as the Kalman filter.
CHAPTER 5: CONCLUSIONS AND RECOMMENDATIONS

5.1 Analysis of Results

5.1.1 Effects of Distance Between CG and IMU

The results presented in the previous section, do not entirely support the hypothesis that an IMU placed farther from the center of rotation, is less accurate than one that is positioned closer. During the experiments IMU #0 was placed at a radial distance of approximately 244 mm from the axis of rotation while IMU #1 was placed approximately 61 mm away (Refer to Figure 3.3). The difference in the error from IMU #0 to #1 is significant when considering the results from the Unfiltered Accelerometer algorithm, algorithm #2, in the first two columns of Table 4.2. The pitch error for algorithm #2 from the IMU #1 data has an error lower than most, but the error from the IMU #0 data is much worse than all others. This is caused by the extra accelerations exerted on IMU #0 because of its greater radial distance from the center of rotation as illustrated in Figure 3.1 and the accompanying equations. The plot below compares the accelerations measured by the two IMUs along the x-axis.

![Figure 5.1: Measured accelerations in x-direction from IMU #0 and #1 plotted together.](image)

It is clear in Figure 5.1 that the measured accelerations from IMU #0 have much higher peaks. Since most of the compared algorithms make the assumption that gravity is the only linear acceleration, it was hypothesized that the added centripetal and tangential accelerations decrease the performance of the algorithms. This effect is exacerbated by placing the IMU farther from the CG or center of rotation.
It was expected that algorithms #3 through #6 would also be significantly affected. However, examining the first two columns of Table 4.2 shows that the errors are actually slightly higher for the IMU #1 estimates compared to the IMU #0 estimates. Further processing of the data showed that this is due to the different gyro measurements, using either the gyro from IMU #0 or from IMU #1. If all algorithms use the same gyro measurements, the differences in the pitch estimate errors are negligible. Therefore, this experiment indicates that the difference in the radial distance between IMU and CG does not affect algorithms #3 through #6 under certain conditions. The tuning parameters for those algorithms were set to rely more on the gyro data than the accelerometer data. If tuning adjustments caused the accelerometer measurements to have more of an influence, it is very likely that the increased radial distance would affect performance.

5.1.2 Algorithm Accuracies and Performance

The error indices for the different algorithms given in Table 4.3 and Figure 4.30 show that algorithm #8 performed the best followed closely by algorithm #9. Generally, the Novel/TIMU algorithms had a definite advantage over the others, especially #8 and #9 that performed significantly better than the rest. Of all the compared algorithms, #9 is recommended for further investigation and use because it had a very low error index, almost the lowest. Although algorithm #8 has the lowest error index, the pitch error estimate of algorithm #9 is much smoother than #8 making it the preferred algorithm. This is well illustrated in Figure 4.13 where the amplitude of #8 noise is as much as 0.25 degrees while #9 is much smaller.

Observing the plots of the estimates shows that #1, #3, #5, and #6 were very close to one another. Before completing the tuning procedure where the tuning parameters for each algorithm were determined, they were not as tightly grouped. It appears that the method used for tuning exposed the fact that those four algorithms rely heavily on the gyro data. Algorithms #3, #5, and #6 can be tuned such that the balance between the influence of the accelerometers and the gyros are changed. The tuning method dictated tuning parameters that make the algorithms rely more heavily on the gyro measurements. This was caused by the inclusion of the pitch rate estimate in the optimization criteria. If only the pitch estimate were considered, the estimates would have lower values of RMS pitch error, trend closer to the ground truth pitch curve and have high noise or jittery pitch estimates that would make the RMS pitch rate estimate very inaccurate.

Ultimately, the tuning method showed that if the algorithms are tuned in a highly consistent way, they produce similar results. Since most of them rely on the same input information and have similar structure, they can only produce similar results. In order to make a significant change in performance, there must be a significant change in available input information. The addition of measurements from a second IMU is what makes the increase in performance of the Novel algorithms possible.

The apparent shift between the Kalman filter and the group of algorithms #1, #3, #5 and #6, cannot be definitely explained. Checking the code for a possible index shift of data was inconclusive. The way in which the Kalman filter calculates a prediction and then checks the prediction is likely why it appears to estimate one cycle ahead of the others which are all basically backwards difference schemes.
5.1.3 Complexity and Computational Cost

Regarding the comparative computational costs addressed with Table 4.4, it is important to understand that the #5 Madgwick and #6 Mahony algorithms are quaternion formulations. Therefore, the estimates that they produce contain three-dimensional attitude information, not just the pitch attitude. This means that if roll attitude information is required in addition to pitch, the runtime of the other algorithms must double, while the Madgwick and Mahony do not change. Even if the run times of the others double, the Madgwick and Mahony take much more time to run which is likely because of the quaternion arithmetic required to convert quaternions to Euler angles. This step may or may not be required for an actual sUAS flight control system. If single degree of freedom formulations of these algorithms were used instead, it is likely that they would be similar to the Novel algorithms, judging by the complexity of the code.

It was important for this research that the Novel algorithms are comparable to the others in computational cost. The run times presented in Table 4.4 indicate that the Novel algorithms did take close to the same amount of time as algorithms #1 through #4. This indicates that the Novel algorithms do not incur significantly more computational cost than #1 through #4 and are comparable.

The complexity of the code for the Novel algorithms was also comparable to the Kalman Madgwick and Mahony algorithms. An average coder can certainly reproduce the Novel algorithms fairly quickly. This, along with the aforementioned conclusions, show that the goals of this research were met. Specifically, algorithms that use only accelerometer and gyroscope sensors were evaluated and compared as directly as possible. It is shown that investigating the derivative of the estimate is important to understand the overall performance of the estimator. Next, the hypothesis that performance of the algorithms is affected by relative distance between the IMU and the CG was investigated. It was shown that the performance can be affected by the relative distance under certain conditions, but the results presented here do not show an affect for most of the algorithms. Lastly, a novel attitude estimation algorithm which uses two IMUs was proposed that offers better performance than the evaluated algorithms with comparable computational cost and is simple to understand, explain, and reproduce for laymen or DIY projects.

5.2 Recommendations for Further Research

Only one relative position between the IMUs and the CG was investigated in this research. It is recommended for further assessment of the TIMU’s performance that a sensitivity analysis be performed on the positions of the IMUs from the CG. In other words, the algorithm’s performance is determined for multiple cases where the IMUs are moved closer and further away from the CG and each other. The TIMU algorithm relies on the motion model where there is separation between the IMU and CG. Determining the minimum separation for acceptable accuracy is an important next step in evaluation of the TIMU algorithm.

The comparison of the algorithms presented here was mostly academic because it focused on post-processing of measurements and performance criteria that only considered the algorithms themselves. For future research, it is recommended that the algorithms be used to provide attitude information to an automatically controlled system and performance be measured based on the performance of the system as a whole. Possible examples of this are using the algorithms in a
system meant to control attitude of a partially constrained UAS, or an inverted pendulum. A practical comparison of this kind will expose the algorithms to performance effecting factors that were not included in the research presented here. For example, the vibrations of an electric motor that is part of a sUAS may cause changes in the IMU measurements that effect some algorithms differently. This insight is not possible with the experimental apparatus presented here, though it is possible to modify it for that purpose. A more practical experiment may also bring to light the advantages of the different tuning mechanisms for the algorithms which, if they exist, were mostly nullified by the tuning method used in this research. It is also likely that the tuning of the controller or compensator can rectify any issues caused by jitter or noise coming from the attitude estimation, which makes the performance metrics of this research less relevant. The importance of the pitch rate estimate was championed herein, but a practical test may show that its importance is overstated.

With a subject as vast as attitude estimation, the number of things that one can try in order to improve the estimates is almost endless. For example, it may be useful to use higher order integration methods instead of the first order Euler method used for the algorithms presented here, such as Runge-Kutta or Adams-Bashforth. The added complexity will have to be balanced with added performance, if any. It would also be beneficial to employ an EKF estimator and see how it compares to the no-GPS/no-magnetometer methods presented here musing the same gyro and accelerometer data. This comparison would add more context to the quality of the estimates compared to what the vast majority of UAS are using, the EKF.

Adding additional IMUs will likely improve the potential accuracy of the Novel algorithms. Incorporating the added information into the equations is trivial, but the increase in computational cost may be an impediment, since simply adding one IMU will add several terms to the computations. If the TIMU is to be used for attitude estimation on a free body that rotates in 3 dimensions, the model will need to be expanded to account for the coupled accelerations. This expansion may excessively increase complexity and further investigation is needed. Algorithms #1 through #4 are sometimes used to estimate pitch and roll simultaneously by applying the one dimensional algorithm to each dimension individually, ignoring the coupled accelerations [12], [11]. Applying the TIMU algorithm in a similar fashion may provide adequate results.

The Novel algorithms depend on prior knowledge of the relative distances between the CG and the IMUs, so their use on a UAS that has a highly variable CG location will likely fail. However, the same principles described here for the Novel algorithms can be used as the basis for a system which can estimate the CG location and provide attitude information similar to what is described by Al-Rawashdeh, et al. [31]. This is certainly an important consideration with the possibility of delivery drones and drones with robotic manipulators becoming more than just concepts in the future.
REFERENCES


APPENDIX A: Optimized Kalman Filter Code

This appendix gives a summary of an optimized way of coding the Linear Kalman filter.

The Kalman filter was presented in Section 3.2.4 as a set of matrix equations. Writing code using matrix arithmetic requires either a library or special functions to interpret the syntax and perform the needed calculations. This can add computational overhead and complexity. In order to avoid this, the method of coding the Kalman filter that is used in this research, is given here.

This method consists of a set of equations that calculates the minimum number of coefficients to produce the results of the equations given in Section 3.2.4. The a priori coefficients begin with the equation below.

\[ X_1 = X_1 + dT(\omega_y - X_2) \]  \hspace{1cm} A.1

Equation A.1 yields the a priori value of the pitch angle represented by “\( X_1 \)”. “\( dT \)” is the sample time from the previous cycle. “\( X_2 \)” is the gyro bias estimate and is updated below, but can be initialized with a zero. “\( \omega_y \)” is the gyroscope measurement from the current time step.

\[ P_{00} = P_{00} + dT[-2P_{01} + dT(P_{11})] + Q_4 \]  \hspace{1cm} A.2

\[ P_{01} = P_{01} - dT(P_{11}) \]  \hspace{1cm} A.3

\[ P_{11} = P_{11} + Q_4 \]  \hspace{1cm} A.4

Equations A.2 through A.4 represent the a priori elements of the “\( P \)” matrix, where “\( Q_4 \)” is the process noise covariance for the accelerometer

\[ Kk0 = P_{00}/(P_{00} + R_4) \]  \hspace{1cm} A.5

\[ Kk1 = P_{01}/(P_{00} + R_4) \]  \hspace{1cm} A.6

Equations A.5 and A.6 calculate the elements of the Kalman gain matrix where “\( R_4 \)” represents the measurement noise covariance.

\[ T_1 = atan2( a_x, -a_z ) \]  \hspace{1cm} A.7

\[ X_2 = X_2 + Kk1(T_1 - X_1) \]  \hspace{1cm} A.8

\[ X_1 = X_1 + Kk0(T_1 - X_1) \]  \hspace{1cm} A.9
“TI” in Equation A.7 above, is the same as the unfiltered estimate based on only accelerometer measurements, represented by “\(a_x\)” and “\(a_z\)”. It is used in Equations A.8 and A.9 to calculate the a posteriori values of “\(X_1\)” and “\(X_2\)”.

\[
P_{11} = P_{11} - K_{k1}(P_{01}) \tag{A.10}
\]

\[
P_{00} = P_{00}(1 - K_{k0}) \tag{A.11}
\]

\[
P_{01} = P_{01}(1 - K_{k0}) \tag{A.12}
\]

Equations A.10 through A.12 are used to calculate the a posteriori values of “\(P_{00}\)”, “\(P_{01}\)”, and “\(P_{11}\)”. This concludes one full cycle of the algorithm.
APPENDIX B: Optimized TIMU Algorithm Code

This appendix gives a summary of an optimized way of coding the TIMU algorithm.

The TIMU algorithm requires the solution to a matrix equation for every cycle or set of measurements. The equation is shown below.

\[ [b] = [A][x] \] \hspace{1cm} B.1

In the above equation, “x” is a vector of the unknown variables for which the equation is being solved. Separating the vector of unknowns and writing the equation to solve for “x” is trivial, shown in the equation below.

\[ [x] = [A]^{-1}[b] \] \hspace{1cm} B.2

While writing the equation above is trivial, determining the inverse of the “A” matrix is not. Finding the inverse of a matrix can be done in a number of ways, such as pseudo inverse algorithms or factorization methods. A popular way of solving matrix equations of the form in Equation B.2 above is the use of the “\" operator in MATLAB. The “\" operator in MATLAB invokes a function called “mldivide” that chooses the best method of solving the equation based on the composition of the “A” matrix. However, this function is likely not available to those programming attitude estimators on small micro-controllers or those who want the software to be as lightweight as possible. For that reason, an optimized way of coding the TIMU algorithm that is suitable for C or C++ code, is presented here.

The “A” matrix used in the TIMU algorithm is size 6x6. The symbolic toolbox in MATLAB is used to solve for the inverse of that matrix, producing a matrix of the 36 terms of the original 6x6 matrix. Then the number of terms is reduced based on symmetries and like terms. This reduction shows that there is a minimum of 5 terms required to reproduce the entire inverse matrix. These terms are presented below.

\[ AI00 = (rz0 - rz1)/(rx0^2 + 2*rz0 * rz1 + rz1^2 + rz0^2) - 2 * rz0 * rz1 + rz1^2) \] \hspace{1cm} B.3

\[ AI01 = (rx0 + rz1)/(rx0^2 + 2*rz0 * rz1 + rz1^2 + rz0^2) - 2 * rz0 * rz1 + rz1^2) \] \hspace{1cm} B.4

\[ AI20 = (rx1^2 + rx0 * rz1 + rz1^2 - rz0 * rz1)/(rx0^2 + 2 * rz0 * rz1 + rz1^2 + rz0^2 - 2 * rz0 * rz1 + rz1^2); \] \hspace{1cm} B.5
\[ AI21 = \frac{-(rx0 \cdot rz1 + rx1 \cdot rz0)}{(rx0^2 + 2 \cdot rx0 \cdot rx1 + rx1^2 + rz0^2 - 2 \cdot rz0 \cdot rz1 + rz1^2)} \]  

\[ AI22 = \frac{(rx0^2 + rx1 \cdot rx0 + rz0^2 - rz1 \cdot rz0)}{(rx0^2 + 2 \cdot rx0 \cdot rx1 + rz1^2)} \]  

The five terms shown above can be calculated once the relative distances of the IMUs from the CG are known. These are represented by “rx0” and “rz0” for the distances in the x and y-directions respectively, for IMU #0. “rx1” and “rz1” represent the distances for IMU #1. Calculating these terms needs to occur only once at the start of the program, and then they can be stored for use during every cycle of the estimator.

The last two terms of the “b” vector are zeros, which further simplifies the expressions. Using the terms shown above to fill in the inverse “A” matrix, multiplying it by the “b” vector produces the expression for the “x” vector shown below.

\[
\begin{bmatrix}
\alpha_y \\
\omega_y^2 \\
a_{lin,x0} \\
a_{lin,z0} \\
a_{lin,x1} \\
a_{lin,z1}
\end{bmatrix} = \begin{bmatrix}
AI00 \cdot b1 - AI00 \cdot b3 + AI01 \cdot b2 - AI01 \cdot b4 \\
AI00 \cdot b2 - AI01 \cdot b1 - AI00 \cdot b4 + AI01 \cdot b3 \\
AI20 \cdot b1 + AI21 \cdot b2 - AI21 \cdot b4 + AI22 \cdot b3 \\
AI20 \cdot b2 - AI21 \cdot b1 + AI21 \cdot b3 + AI22 \cdot b4 \\
AI20 \cdot b1 + AI21 \cdot b2 - AI21 \cdot b4 + AI22 \cdot b3 \\
AI20 \cdot b2 - AI21 \cdot b1 + AI21 \cdot b3 + AI22 \cdot b4
\end{bmatrix}
\]

where:

\[
\begin{bmatrix}
b1 \\
b2 \\
b3 \\
b4
\end{bmatrix} = \begin{bmatrix}
a_{total,x0} \\
a_{total,z0} \\
a_{total,x1} \\
a_{total,z1}
\end{bmatrix}
\]

Refer back to Section 3.2.7 for explanation of the symbols above. Note that the third and fourth terms in the “x” vector above are always equal to the fifth and sixth terms respectively. This is another place where the operations can be reduced in code, but all the terms are shown here for simplicity and readability. In addition, calculating the first and second terms can be omitted all together since those terms are not needed for the TIMU algorithm.

In summary, the matrix equation for the TIMU algorithm can be solved every cycle by first calculating the five terms of the inverse “A” equation, given in Equations B.3 through B.7 above, once when the program starts and then storing them. Then, they are used at each cycle in the expression for “x”, along with the accelerometer measurements, to determine the unknown variables.

For further simplification, only values for the “AI20”, “AI21”, and “AI22” terms are required to calculate the third and fourth terms of the “x”, which are the only two of the six terms needed for the TIMU algorithm. This process greatly reduces the computational effort required to facilitate the TIMU algorithm compared to other methods of solving the matrix equation.
APPENDIX C: Quaternion Multiplication and other Operations

Quaternions have their own associated set of arithmetic operations. Quaternion multiplication is presented in this appendix to provide explanation beyond what is given in sections 3.2.5 and 3.2.6, regarding the Madgwick and Mahony algorithms respectively. The multiplication of quaternion “a” and “b” is shown below.

\[
a \otimes b = \begin{bmatrix}
a_1(b_1) - a_2(b_2) - a_3(b_3) - a_4(b_4) \\
a_1(b_2) + a_2(b_1) + a_3(b_4) - a_4(b_3) \\
a_1(b_3) - a_2(b_4) + a_3(b_1) + a_4(b_2) \\
a_1(b_4) + a_2(b_3) - a_3(b_2) + a_4(b_1)
\end{bmatrix}
\]  

C.1

where:

\[
a = [a_1 \ a_2 \ a_3 \ a_4]
\]  

C.2

and:

\[
b = [b_1 \ b_2 \ b_3 \ b_4]
\]  

C.3

Converting from quaternion to Euler angles is also used for this research.

\[
r_{11} = 2q_1^2 - 1 + 2q_2^2 
\]  

C.4

\[
r_{21} = 2(q_2q_3 - q_1q_4) 
\]  

C.5

\[
r_{31} = 2(q_2q_4 + q_1q_3) 
\]  

C.6

\[
r_{32} = 2(q_1q_4 - q_1q_2) 
\]  

C.7

\[
r_{33} = 2q_1^2 - 1 + 2q_4^2 
\]  

C.8

\[
\varphi = \text{atan2}(r_{32}, r_{33})
\]  

C.9

\[
\theta = -\text{atan}\left(\frac{r_{31}}{\sqrt{1 - r_{31}^2}}\right) 
\]  

C.10

\[
\psi = \text{atan2}(r_{21}, r_{11})
\]  

C.11

Equations C.4 through C.11 provide the expressions needed to convert the orientation quaternion, “q” into Euler angles. All of the equations from this appendix are taken from code written by Sebastian Madgwick [35].
APPENDIX D: Arduino Code Used in Experiment

The code written in the Arduino language (C++) for the recoding of measurements from the IMUs is given in this appendix.

File Name: MPU9255_experiment.ino

// Joshua Milam, 2017
#include <Wire.h>
#include <Encoder.h>
#include "MPU9255.hpp"

// ---- Initializing Variables for Data ------:
// ---- Encoder ---:
Encoder encoder(18,19);
float encoderRatio = 4096.0/180.0;
float encoderAngle;

// ---- IMU 0 ---:
MPU9255 IMU_0;
int16_t accelVect_0[3];
int16_t gyroVect_0[3];
int16_t magVect_0[3];
int16_t tempData_0;

// ---- IMU 1 ---:
MPU9255 IMU_1;
int16_t accelVect_1[3];
int16_t gyroVect_1[3];
int16_t magVect_1[3];
int16_t tempData_1;

// Timer Variables:
float loopTime = 0; // (sec)
unsigned long loopTime_desired = 10000; // (micro-seconds) adjust for desired cycle rate - 10000 for Hz, 92000 for 95Hz
float totalTime = 0; // (sec)
unsigned long timerStart = 0; // (sec)

void setup() // Setup Loop
{
    Serial.begin(115200); //Start the serial connection.
    Serial.println(); Serial.print(" -- Begin Setup Loop -- ");
    Serial.println();
    Serial.print("encoderRatio = "); Serial.println(encoderRatio);

    Wire.begin(); // Start the I2C Communication.
    Wire.setClock(500000);
    tcaSelect(0); // Initialize IMU #0
    IMU_0.initMPU9255();
}
tcaSelect(1); // Initialize IMU #1
IMU_1.initMPU9255();

tcaSelect(0); // Call calibration function for IMU #0
IMU_0.calibrateAHRS();

tcaSelect(1); // Call calibration function for IMU #1
IMU_1.calibrateAHRS();

encoder.write(0); // Setting the initial encoder postion to 0 for 0 degrees pitch.
} // end setup()

void loop() // Main Loop
{
    timerStart = micros(); // Start of loop in microseconds

    // Read Raw IMU Data:
    // IMU_0
    tcaSelect(0);
    IMU_0.readAccelData(accelVect_0);
    IMU_0.readGyroData(gyroVect_0);
    IMU_0.readMagData(magVect_0);
    IMU_0.readTempData(tempData_0);

    // IMU_1
    tcaSelect(1);
    IMU_1.readAccelData(accelVect_1);
    IMU_1.readGyroData(gyroVect_1);
    IMU_1.readMagData(magVect_1);
    IMU_1.readTempData(tempData_1);

    encoderAngle = encoder.read()/encoderRatio; //Read the encoder position.

    // -----------Output Raw Measurements-----------:
    // IMU_0
    // Accelerometer Data:
    Serial.print(accelVect_0[0]); Serial.print("t"); // X
    Serial.print(accelVect_0[1]); Serial.print("t"); // Y
    Serial.print(accelVect_0[2]); Serial.print("t"); // Z
    // Gyroscope Data:
    Serial.print(gyroVect_0[0]); Serial.print("t"); // X
    Serial.print(gyroVect_0[1]); Serial.print("t"); // Y
    Serial.print(gyroVect_0[2]); Serial.print("t"); // Z
    // Magnetometer Data:
    Serial.print(magVect_0[0]); Serial.print("t"); // X
    Serial.print(magVect_0[1]); Serial.print("t"); // Y
    Serial.print(magVect_0[2]); Serial.print("t"); // Z
    // Temperature Data:
    Serial.print(tempData_0); Serial.print("t");

    // IMU_1:
    // Accelerometer Data:
    Serial.print(accelVect_1[0]); Serial.print("t"); // X
    Serial.print(accelVect_1[1]); Serial.print("t"); // Y
    Serial.print(accelVect_1[2]); Serial.print("t"); // Z
// Gyroscope Data:
Serial.print(gyroVect_1[0]); Serial.print("\t"); // X
Serial.print(gyroVect_1[1]); Serial.print("\t"); // Y
Serial.print(gyroVect_1[2]); Serial.print("\t"); // Z

// Magnetometer Data:
Serial.print(magVect_1[0]); Serial.print("\t"); // X
Serial.print(magVect_1[1]); Serial.print("\t"); // Y
Serial.print(magVect_1[2]); Serial.print("\t"); // Z

// Temperature Data:
Serial.print(tempData_1); Serial.print("\t");

// Encoder Output:
Serial.print(encoderAngle,4); Serial.print("\t");

//------- Delays for Sample Rate Control and Time Keeping ---------
while((loopTime_desired-(micros()-timerStart)) < 0){} // this is to protect
against delaying the loop too long
delayMicroseconds(loopTime_desired-(micros()-timerStart)); // Making the
loop run at the desired cycle rate.

loopTime = (micros()-timerStart)*0.000001; // (sec) Calculating loop time
and converting to seconds.
totalTime += loopTime;

Serial.print(loopTime,6); Serial.print("\t"); // Recording the sample time
for current loop
Serial.print(totalTime); Serial.println(); // Recording the total time
since beginning of recording

} // end loop()

// end program

File Name: TCA9548.ino

// Joshua Milam, 2018
#define TCAADDR 0x70

void tcaSelect(uint8_t i)
{
if (i > 7) return;

Wire.beginTransmission(TCAADDR);
Wire.write(i << i);
Wire.endTransmission();
}

//You can then call tcaselect(0) thru tcaselect(7) to set up the multiplexer.

File Name: MPU9255.hpp

// Joshua Milam, 2017
// This code is based on partially copied and modified
// code from various Arduino forum threads and online examples as well as
class MPU9255
{

protected:

// Register names according to the datasheet.
// According to the InvenSense document
// "MPU-9255 Register Map and Descriptions Revision 1.0",

uint8_t MPU9255_I2C_ADDRESS = 0x68;

uint8_t MPU9255_SELF_TEST_X_GYRO = 0x00; // R/W
uint8_t MPU9255_SELF_TEST_Y_GYRO = 0x01; // R/W
uint8_t MPU9255_SELF_TEST_Z_GYRO = 0x02; // R/W
uint8_t MPU9255_SELF_TEST_X_ACCEL = 0x0D; // R/W
uint8_t MPU9255_SELF_TEST_Y_ACCEL = 0x0E; // R/W
uint8_t MPU9255_SELF_TEST_Z_ACCEL = 0x0F; // R/W
uint8_t MPU9255_XG_OFFSET_H = 0x13; // R/W
uint8_t MPU9255_XG_OFFSET_L = 0x14; // R/W
uint8_t MPU9255_YG_OFFSET_H = 0x15; // R/W
uint8_t MPU9255_YG_OFFSET_L = 0x16; // R/W
uint8_t MPU9255_ZG_OFFSET_H = 0x17; // R/W
uint8_t MPU9255_ZG_OFFSET_L = 0x18; // R/W
uint8_t MPU9255_SMPLRT_DIV = 0x19; // R/W
uint8_t MPU9255_CONFIG = 0x1A; // R/W
uint8_t MPU9255_GYRO_CONFIG = 0x1B; // R/W
uint8_t MPU9255_ACCEL_CONFIG = 0x1C; // R/W
uint8_t MPU9255_ACCEL_CONFIG_2 = 0x1D; // R/W
uint8_t MPU9255_LP_ACCEL_ODR = 0x1E; // R/W
uint8_t MPU9255_WOM_THR = 0x1F; // R/W
uint8_t MPU9255_FIFO_EN = 0x23; // R/W
uint8_t MPU9255_I2C_MST_CTRL = 0x24; // R/W
uint8_t MPU9255_I2C_SLV0_ADDR = 0x25; // R/W
uint8_t MPU9255_I2C_SLV0_REG = 0x26; // R/W
uint8_t MPU9255_I2C_SLV0_CTRL = 0x27; // R/W
uint8_t MPU9255_I2C_SLV1_ADDR = 0x28; // R/W
uint8_t MPU9255_I2C_SLV1_REG = 0x29; // R/W
uint8_t MPU9255_I2C_SLV1_CTRL = 0x2A; // R/W
uint8_t MPU9255_I2C_SLV2_ADDR = 0x2B; // R/W
uint8_t MPU9255_I2C_SLV2_REG = 0x2C; // R/W
uint8_t MPU9255_I2C_SLV2_CTRL = 0x2D; // R/W
uint8_t MPU9255_I2C_SLV3_ADDR = 0x2E; // R/W
uint8_t MPU9255_I2C_SLV3_REG = 0x2F; // R/W
uint8_t MPU9255_I2C_SLV3_CTRL = 0x30; // R/W
uint8_t MPU9255_I2C_SLV4_ADDR = 0x31; // R/W
uint8_t MPU9255_I2C_SLV4_REG = 0x32; // R/W
uint8_t MPU9255_I2C_SLV4_DO = 0x33; // R/W
uint8_t MPU9255_I2C_SLV4_CTRL = 0x34; // R/W
uint8_t MPU9255_I2C_SLV4_DI = 0x35; // R
uint8_t MPU9255_I2C_MST_STATUS = 0x36; // R
uint8_t MPU9255_INT_PIN_CFG = 0x37; // R/W
uint8_t MPU9255_INT_ENABLE = 0x38; // R/W
uint8_t MPU9255_INT_STATUS = 0x3A; // R
```c
uint8_t MPU9255_ACCEL_XOUT_H = 0x3B; // R
uint8_t MPU9255_ACCEL_XOUT_L = 0x3C; // R
uint8_t MPU9255_ACCEL_YOUT_H = 0x3D; // R
uint8_t MPU9255_ACCEL_YOUT_L = 0x3E; // R
uint8_t MPU9255_ACCEL_ZOUT_H = 0x3F; // R
uint8_t MPU9255_ACCEL_ZOUT_L = 0x40; // R
uint8_t MPU9255_TEMP_OUT_H = 0x41; // R
uint8_t MPU9255_TEMP_OUT_L = 0x42; // R
uint8_t MPU9255_GYRO_XOUT_H = 0x43; // R
uint8_t MPU9255_GYRO_XOUT_L = 0x44; // R
uint8_t MPU9255_GYRO_YOUT_H = 0x45; // R
uint8_t MPU9255_GYRO_YOUT_L = 0x46; // R
uint8_t MPU9255_GYRO_ZOUT_H = 0x47; // R
uint8_t MPU9255_GYRO_ZOUT_L = 0x48; // R
uint8_t MPU9255_EXT_SENS_DATA_00 = 0x49; // R
uint8_t MPU9255_EXT_SENS_DATA_01 = 0x4A; // R
uint8_t MPU9255_EXT_SENS_DATA_02 = 0x4B; // R
uint8_t MPU9255_EXT_SENS_DATA_03 = 0x4C; // R
uint8_t MPU9255_EXT_SENS_DATA_04 = 0x4D; // R
uint8_t MPU9255_EXT_SENS_DATA_05 = 0x4E; // R
uint8_t MPU9255_EXT_SENS_DATA_06 = 0x4F; // R
uint8_t MPU9255_EXT_SENS_DATA_07 = 0x50; // R
uint8_t MPU9255_EXT_SENS_DATA_08 = 0x51; // R
uint8_t MPU9255_EXT_SENS_DATA_09 = 0x52; // R
uint8_t MPU9255_EXT_SENS_DATA_10 = 0x53; // R
uint8_t MPU9255_EXT_SENS_DATA_11 = 0x54; // R
uint8_t MPU9255_EXT_SENS_DATA_12 = 0x55; // R
uint8_t MPU9255_EXT_SENS_DATA_13 = 0x56; // R
uint8_t MPU9255_EXT_SENS_DATA_14 = 0x57; // R
uint8_t MPU9255_EXT_SENS_DATA_15 = 0x58; // R
uint8_t MPU9255_EXT_SENS_DATA_16 = 0x59; // R
uint8_t MPU9255_EXT_SENS_DATA_17 = 0x5A; // R
uint8_t MPU9255_EXT_SENS_DATA_18 = 0x5B; // R
uint8_t MPU9255_EXT_SENS_DATA_19 = 0x5C; // R
uint8_t MPU9255_EXT_SENS_DATA_20 = 0x5D; // R
uint8_t MPU9255_EXT_SENS_DATA_21 = 0x5E; // R
uint8_t MPU9255_EXT_SENS_DATA_22 = 0x5F; // R
uint8_t MPU9255_EXT_SENS_DATA_23 = 0x60; // R
uint8_t MPU9255_I2C_SLV0_DO = 0x63; // R/W
uint8_t MPU9255_I2C_SLV1_DO = 0x64; // R/W
uint8_t MPU9255_I2C_SLV2_DO = 0x65; // R/W
uint8_t MPU9255_I2C_SLV3_DO = 0x66; // R/W
uint8_t MPU9255_I2C_MST_DELAY_CTRL = 0x67; // R/W
uint8_t MPU9255_SIGNAL_PATH_RESET = 0x68; // R/W
uint8_t MPU9255_MOT_DETECT_CTRL = 0x69; // R/W
uint8_t MPU9255_USER_CTRL = 0x6A; // R/W
uint8_t MPU9255_PWR_MGMT_1 = 0x6B; // R/W
uint8_t MPU9255_PWR_MGMT_2 = 0x6C; // R/W
uint8_t MPU9255_FIFO_COUNTH = 0x72; // R/W
uint8_t MPU9255_FIFO_COUNTL = 0x73; // R/W
uint8_t MPU9255_FIFO_R_W = 0x74; // R
uint8_t MPU9255_WHO_AM_I = 0x75; // R/W
uint8_t MPU9255_XA_OFFSET_H = 0x77; // R/W
uint8_t MPU9255_XA_OFFSET_L = 0x78; // R/W
uint8_t MPU9255_YA_OFFSET_H = 0x7A; // R/W
uint8_t MPU9255_YA_OFFSET_L = 0x7B; // R/W
uint8_t MPU9255_ZA_OFFSET_H = 0x7D; // R/W
```
```c
uint8_t MPU9255_ZA_OFFSET_L = 0x7E; // R/W

// Registers For Accelerometer:
uint8_t AK8963_I2C_ADDRESS = 0x0C;
uint8_t AK8963_WIA = 0x00; // R
uint8_t AK8963_INFO = 0x01; // R
uint8_t AK8963_ST1 = 0x02; // R
uint8_t AK8963_HXL = 0x03; // R
uint8_t AK8963_HXH = 0x04; // R
uint8_t AK8963_HYL = 0x05; // R
uint8_t AK8963_HYH = 0x06; // R
uint8_t AK8963_HZL = 0x07; // R
uint8_t AK8963_HZH = 0x08; // R
uint8_t AK8963_ST2 = 0x09; // R
uint8_t AK8963_CNTL1 = 0x0A; // R/W
uint8_t AK8963_CNTL2 = 0x0B; // R/W
uint8_t AK8963_ASTC = 0x0C; // R/W
uint8_t AK8963_TS1 = 0x0D; // R/W // DO NOT USE
uint8_t AK8963_TS2 = 0x0E; // R/W // DO NOT USE
uint8_t AK8963_I2CDIS = 0x0F; // R/W // Disables I2C interface
uint8_t AK8963_ASAX = 0x10; // R
uint8_t AK8963_ASAY = 0x11; // R
uint8_t AK8963_ASAZ = 0x12; // R

int16_t accelVect[3]; // < ---- consider making these two private
int16_t gyroVect[3]; // < ---- consider making these two private

public:

int16_t accelVect_offset[3];
int16_t gyroVect_offset[3];

uint8_t magSensAdjVect[3]; // Vector of X/Y/Z Magnetic sensitivity adjustment values from magnetometer fuse ROM.
float magAdjCoefVect[3];

void initMPU9255();
void initAK8963();
void calibrateAHRS();

void writeByte(uint8_t address, uint8_t subAddress, uint8_t data);
uint8_t readByte(uint8_t address, uint8_t subAddress);
void readBytes(uint8_t address, uint8_t subAddress, uint8_t count, uint8_t * dest);
void readAccelData(int16_t * destination);
```
void readGyroData(int16_t * destination);
void readMagData(int16_t * destination);
void readTempData(int16_t & destination);

}; //end of class

File Name: MPU9255.cpp

// Joshua Milam, 2017
// This code is based on partially copied and modified
// code from various Arduino forum threads and online examples as well as
// some open source libraries. Some of this code can be found here:
// https://playground.arduino.cc/Main/MPU-6050#short

#include <Wire.h>
#include <arduino.h>
#include "MPU9255.hpp"

void MPU9255::initMPU9255()
{
    //writeByte(MPU9255_I2C_ADDRESS, PWR_MGMT_1, 0x00); //(0x68, 0x6B, 0x00) -
    //Sets the clock source to the Internal 20MHz oscillator. I don't think this is
    //needed.
    //delay(100);
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_PWR_MGMT_1, 0x01); //(0x68, 0x6B,
    //0x01) - This is reset for PWR_MGMT_1, auto selects the best available clock
    //source
    delay(200);
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_CONFIG, 0x03); // (0x68,
    //0x1A, 0x03) - some sort of filter configuration. See chart on page 13
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_SMPLRT_DIV, 0x04); // (0x68, 0x19,
    //0x04) - Sets the update rate of sensor registers
    uint8_t c = readByte(MPU9255_I2C_ADDRESS, MPU9255_GYRO_CONFIG); // (0x68,
    //0x1B)
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_GYRO_CONFIG, c & ~0x02);
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_GYRO_CONFIG, c & ~0x18);
    int Gscale = 0;
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_GYRO_CONFIG, c | Gscale << 3);
    //c = readByte(MPU9255_I2C_ADDRESS, MPU9255_GYRO_CONFIG); // (0x68, 0xB)
    Verifying that the Gyro Is set to config 0 so that it is 131 LSB/dps
    Serial.print("c2 ");Serial.println(c); /*
    c = readByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG); // (0x68,
    //0x1C) */
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG, c & ~0x18);
    int Ascale = 0;
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG, c | Ascale << 3);
    c = readByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG_2); // (0x68,
    //0x1D)
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG_2, c & ~0x0F);
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_CONFIG_2, c | 0x03);
    writeByte(MPU9255_I2C_ADDRESS, MPU9255_INT_PIN_CFG, 0x02); // - Needed to
    //enable communication with the magnetometer.
//writeByte(MPU9255_I2C_ADDRESS, MPU9255_INT_ENABLE, 0x00);   // (0x68, 0x38, 0x01) - Can probably disable this without effect (write 0x00) delay(100);

initAK8963(); // Initialize the magnetometer.

Serial.print("- End of initMPU9255() - "); Serial.println("\n");} // end MPU9255::initMPU9255()

void MPU9255::initAK8963()
{
  // First extract the factory calibration for each magnetometer axis
  uint8_t rawData[3]; // x/y/z gyro calibration data stored here

  writeByte(AK8963_I2C_ADDRESS, AK8963_CNTL1, 0x00); // 0x0A // Power down magnetometer
delay(10);

  writeByte(AK8963_I2C_ADDRESS, AK8963_CNTL1, 0x0F); // 0x0A // Enter Fuse ROM access mode
  delay(10);

  readBytes(AK8963_I2C_ADDRESS, AK8963_ASAX, 3, &rawData[0]); // 0x10 // Return x-axis sensitivity adjustment values, etc.

  magAdjCoefVect[0] = (((magSensAdjVect[0]-128)*0.5)/128)+1;
magAdjCoefVect[1] = (((magSensAdjVect[1]-128)*0.5)/128)+1;
magAdjCoefVect[2] = (((magSensAdjVect[2]-128)*0.5)/128)+1;

  //readBytes(AK8963_I2C_ADDRESS, AK8963_ASAX, 3, &rawData[0]);

  Serial.print("Mag Adj Coef Values"); Serial.print("\t");
  Serial.print(magAdjCoefVect[0], 9); Serial.print("\t");
  Serial.print(magAdjCoefVect[1], 9); Serial.print("\t");
  Serial.print(magAdjCoefVect[2], 9); Serial.print("\n");

  writeByte(AK8963_I2C_ADDRESS, AK8963_CNTL1, 0x00); // 0x0A // Power down magnetometer
  delay(10);

  // Configure the magnetometer for continuous read and highest resolution
  // set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution in_CNTL register,
  // and enable continuous mode data acquisition Mmode (bits [3:0]), 0010 for 8 Hz and 0110 for 100 Hz sample rates
  // Replace the hex below with variables ---------------

  writeByte(AK8963_I2C_ADDRESS, AK8963_CNTL1, 0x01 << 4 | 0x06); // Set magnetometer data resolution and sample ODR
  delay(10);
} // end MPU9255::initAK8963()

void MPU9255::calibrateAHRS()
{
  unsigned long calibrationStart = millis();
  int i = 0;

  int32_t accelVect_calib[3] = {0, 0, 0};
int32_t gyroVect_calib[3] = {0, 0, 0};

while((millis() - calibrationStart) < 4000){ // runs for 4 seconds
    readAccelData(accelVect);
    readGyroData(gyroVect);

    accelVect_calib[0] += accelVect[0];
    accelVect_calib[1] += accelVect[1];

    gyroVect_calib[0] += gyroVect[0];
    gyroVect_calib[1] += gyroVect[1];

    i++;
}

accelVect_offset[0] = accelVect_calib[0] / i;
accelVect_offset[1] = (accelVect_calib[1] / i); // +16384; // The "+16384" is to prevent calibrating out the effect of gravity (use this when on bread board)
accelVect_offset[2] = (accelVect_calib[2] / i) - 16384; // The "-16384" is to prevent calibrating out the effect of gravity (use this when on seesaw)

gyroVect_offset[0] = gyroVect_calib[0] / i;

Serial.print("accelVect_offset: "); Serial.print('t');
Serial.print(accelVect_offset[0]); Serial.print('t');
Serial.print(accelVect_offset[1]); Serial.print('t');
Serial.print(accelVect_offset[2]); Serial.println();

Serial.print("gyroVect_offset: "); Serial.print('t');
Serial.print(gyroVect_offset[0]); Serial.print('t');
Serial.print(gyroVect_offset[1]); Serial.print('t');
Serial.print(gyroVect_offset[2]); Serial.println();

Serial.println("End of Calibrate AHRS");
//delay(50000);
}

void MPU9255::writeByte(uint8_t address, uint8_t subAddress, uint8_t data)
{
    Wire.beginTransmission(address);
    Wire.write(subAddress);
    Wire.write(data);
    Wire.endTransmission();
} // end MPU9255::writeByte()

uint8_t MPU9255::readByte(uint8_t address, uint8_t subAddress)
{
    uint8_t data;
    Wire.beginTransmission(address);
    Wire.write(subAddress);
    Wire.endTransmission();

```cpp
void MPU9255::readBytes(uint8_t address, uint8_t subAddress, uint8_t count, uint8_t * dest)
{
    Wire.beginTransmission(address);
    Wire.write(subAddress);
    Wire.endTransmission(false);
    uint8_t i = 0;
    Wire.requestFrom(address, count);
    while (Wire.available())
    {
        dest[i++] = Wire.read();
    }
    Wire.endTransmission();  //<------- Test this!!!
} // end MPU9255::readBytes()

void MPU9255::readAccelData(int16_t * destination)
{
    uint8_t rawData[6];
    readBytes(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_XOUT_H, 6, &rawData[0]);
    readBytes(MPU9255_I2C_ADDRESS, MPU9255_ACCEL_XOUT_H, 6, &rawData[0]);
    destination[0] = ((int16_t)rawData[0] << 8) | rawData[1];
} // end MPU9255::readAccelData()

void MPU9255::readGyroData(int16_t * destination)
{
    uint8_t rawData[6];
    readBytes(MPU9255_I2C_ADDRESS, MPU9255_GYRO_XOUT_H, 6, &rawData[0]);
    destination[0] = ((int16_t)rawData[0] << 8) | rawData[1];
} // end MPU9255::readGyroData()

void MPU9255::readMagData(int16_t * destination)
{
    uint8_t rawData[7]; // x/y/z gyro register data, ST2 register stored here,
                         // must read ST2 at end of data acquisition
    if (readByte(AK8963_I2C_ADDRESS, AK8963_ST1) & 0x01) { // 0x02 // wait
        for magnetometer data ready bit to be set
            readBytes(AK8963_I2C_ADDRESS, AK8963_HXL, 7, &rawData[0]); // 0x03 //
        Read the six raw data and ST2 registers sequentially into data array
        uint8_t c = rawData[6]; // End data read by reading ST2 register
        if (!(c & 0x08)) // Check if magnetic sensor overflow set, if not then
                         // report data
        {
```
destination[0] = ((int16_t)rawData[1] << 8) | rawData[0]; // Turn the MSB and LSB into a signed 16-bit value
}
} // end MPU9255::readMagData()

void MPU9255::readTempData(int16_t & destination)
{
    uint8_t rawData[2];
    readBytes(MPU9255_I2C_ADDRESS, MPU9255_TEMP_OUT_H, 2, &rawData[0]);
    destination = ((int16_t)rawData[0] << 8) | rawData[1];
} // end MPU9255::readTempData()

// end of MPU9255.cpp
APPENDIX E: MATLAB Code Used in Experiment

The code written in the MATLAB for processing and analyzing data from the IMUs is given in this appendix.

File Name: IMU_Processing.m

% Joshua Milam, 2018

% Note: the biases are not applied in raw data
load('DevData3_seesaw.mat'); % Loading the recorded data.

%% Applying the Offsets/Biases, and Making Data Dimensional:

% -- IMU_0 --
% accelerometer data for X, Y, and Z axes
accelVect_0(:,1) = (datamat(:,1) - IMU_0_accelVect_offset(1)) ./ 16384.0; % (g)
accelVect_0(:,2) = (datamat(:,2) - IMU_0_accelVect_offset(2)) ./ 16384.0;
accelVect_0(:,3) = (datamat(:,3) - IMU_0_accelVect_offset(3)) ./ 16384.0;
% gyroscope data for X, Y, and Z axes
gyroVect_0(:,1) = (datamat(:,4) - IMU_0_gyroVect_offset(1)) ./ 131.0; % (dps)
gyroVect_0(:,2) = (datamat(:,5) - IMU_0_gyroVect_offset(2)) ./ 131.0;
gyroVect_0(:,3) = (datamat(:,6) - IMU_0_gyroVect_offset(3)) ./ 131.0;
% magnetometer data for X, Y, and Z axes
magVect_0(:,1) = (datamat(:,7) .* IMU_0_magAdjCoefVect(1)) .* 0.15; % (microT)
magVect_0(:,2) = (datamat(:,8) .* IMU_0_magAdjCoefVect(2)) .* 0.15;
magVect_0(:,3) = (datamat(:,9) .* IMU_0_magAdjCoefVect(3)) .* 0.15;
% temperature data
tempData_0 = datamat(:,10)/333.87 + 21.0; % (degrees C)

% -- IMU_1 --
% accelerometer data for X, Y, and Z axes
accelVect_1(:,1) = (datamat(:,11) - IMU_1_accelVect_offset(1)) ./ 16384.0; % (g)
accelVect_1(:,2) = (datamat(:,12) - IMU_1_accelVect_offset(2)) ./ 16384.0;
accelVect_1(:,3) = (datamat(:,13) - IMU_1_accelVect_offset(3)) ./ 16384.0;
% gyroscope data for X, Y, and Z axes
gyroVect_1(:,1) = (datamat(:,14) - IMU_1_gyroVect_offset(1)) ./ 131.0; % (dps)
gyroVect_1(:,2) = (datamat(:,15) - IMU_1_gyroVect_offset(2)) ./ 131.0;
gyroVect_1(:,3) = (datamat(:,16) - IMU_1_gyroVect_offset(3)) ./ 131.0;
% magnetometer data for X, Y, and Z axes
magVect_1(:,1) = (datamat(:,17) .* IMU_1_magAdjCoefVect(1)) .* 0.15; % (microT)
magVect_1(:,2) = (datamat(:,18) .* IMU_1_magAdjCoefVect(2)) .* 0.15;
magVect_1(:,3) = (datamat(:,19) .* IMU_1_magAdjCoefVect(3)) .* 0.15;
% temperature data
tempData_1 = datamat(:,20)/333.87 + 21.0; % (degrees C)
pitchTruth = datamat(:,21); % (deg) The ground truth from encoder
loopTime = datamat(:,22); % (sec) the loop times for each cycle
totalTime = datamat(:,23); % (sec)

num_samples = length(totalTime); % number of total samples
clear('datamat'); % deleting the loaded time series data, no longer needed

% Using backwards finite difference equation to estimate derivative of % ground truth pitch data.
deriv_pitchTruth = zeros(1, num_samples);
for i = 2:num_samples
deriv_pitchTruth(i) = (pitchTruth(i) - pitchTruth(i-1)) / loopTime(i-1);
end

%% Correcting for non-Aerospace Coordinate System:
% This is needed because the raw data from the IMUs are not in the % conventional aerospace coordinate system which is used throughout the % data processing and analysis:

accelVect_0_tmp = accelVect_0;
gyroVect_0_tmp = gyroVect_0;
magVect_0_tmp = magVect_0;

accelVect_1_tmp = accelVect_1;
gyroVect_1_tmp = gyroVect_1;
magVect_1_tmp = magVect_1;

accelVect_0(:,1) = -accelVect_0_tmp(:,2); % X
accelVect_0(:,2) = -accelVect_0_tmp(:,1); % Y
accelVect_0(:,3) = -accelVect_0_tmp(:,3); % Z
gyroVect_0(:,1) = -gyroVect_0_tmp(:,2); % X
gyroVect_0(:,2) = -gyroVect_0_tmp(:,1); % Y
gyroVect_0(:,3) = -gyroVect_0_tmp(:,3); % Z
magVect_0(:,1) = -magVect_0_tmp(:,2); % X
magVect_0(:,2) = -magVect_0_tmp(:,1); % Y
magVect_0(:,3) = -magVect_0_tmp(:,3); % Z

accelVect_1(:,1) = -accelVect_1_tmp(:,2); % X
accelVect_1(:,2) = -accelVect_1_tmp(:,1); % Y
accelVect_1(:,3) = -accelVect_1_tmp(:,3); % Z
gyroVect_1(:,1) = -gyroVect_1_tmp(:,2); % X
gyroVect_1(:,2) = -gyroVect_1_tmp(:,1); % Y
gyroVect_1(:,3) = -gyroVect_1_tmp(:,3); % Z
magVect_1(:,1) = -magVect_1_tmp(:,2); % X
magVect_1(:,2) = -magVect_1_tmp(:,1); % Y
magVect_1(:,3) = -magVect_1_tmp(:,3); % Z

%% Sample Time Stats:
loopTime_mean = mean(loopTime);
loopTime_std = std(loopTime);
loopTime_min = min(loopTime);
loopTime_max = max(loopTime);
sampleRate_avg = 1/loopTime_mean;
%% Method #1: No filter, only gyro data:

% Calculating the Pitch angle in degrees using only the Gyro Data.
[pitch_m1_0_deg] = gyroOnly_IMU_calc(gyroVect_0, loopTime);
[pitch_m1_1_deg] = gyroOnly_IMU_calc(gyroVect_1, loopTime);

[errorStats_m1_0, deriv_pitch_m1_0] = errorProcessing_calc(pitch_m1_0_deg, pitchTruth, loopTime);
[errorStats_m1_1, deriv_pitch_m1_1] = errorProcessing_calc(pitch_m1_1_deg, pitchTruth, loopTime);

%% Method #2: No filter, only accel data:

% Calculating the Pitch angle in degrees using only the Accel Data:
[pitch_m2_0_deg] = accelOnly_IMU_calc(accelVect_0);
[pitch_m2_1_deg] = accelOnly_IMU_calc(accelVect_1);

[errorStats_m2_0, deriv_pitch_m2_0] = errorProcessing_calc(pitch_m2_0_deg, pitchTruth, loopTime);
[errorStats_m2_1, deriv_pitch_m2_1] = errorProcessing_calc(pitch_m2_1_deg, pitchTruth, loopTime);

%% Method #3: Complementary Filter:

Kc = 0.9999941558732112;

[pitch_m3_0_deg] = complementaryFilter_IMU_calc(accelVect_0, gyroVect_0, loopTime, Kc);
[pitch_m3_1_deg] = complementaryFilter_IMU_calc(accelVect_1, gyroVect_1, loopTime, Kc);

[errorStats_m3_0, deriv_pitch_m3_0] = errorProcessing_calc(pitch_m3_0_deg, pitchTruth, loopTime);
[errorStats_m3_1, deriv_pitch_m3_1] = errorProcessing_calc(pitch_m3_1_deg, pitchTruth, loopTime);

%% Method #4: Kalman Filter:

Q4 = 0.6654010708978304; % Process Noise Covariance
R4 = 1.73243766716539E10; % Measurement Noise Covariance

x2 = 0;
P00 = 0.01;
P01 = 0.01;
P11 = 0.01;

[pitch_m4_0_deg] = simpleKalman_IMU_calc(accelVect_0, gyroVect_0, loopTime, Q4, R4, x2, P00, P01, P11);
pitch_m4_1_deg] = simpleKalman_IMU_calc(accelVect_1, gyroVect_1, loopTime, Q4, R4, x2, P00, P01, P11);

[errorStats_m4_0, deriv_pitch_m4_0] = errorProcessing_calc(pitch_m4_0_deg, pitchTruth, loopTime);
[ errorStats_m4_1, deriv_pitch_m4_1 ] = errorProcessing_calc(pitch_m4_1_deg, pitchTruth, loopTime);

% Method #5: Madgwick IMU Filter:

beta = 4.5177955369958336E-7;

[ pitch_m5_0_deg ] = MadgwickFilter_IMU_calc( accelVect_0, gyroVect_0, loopTime, beta);
[ pitch_m5_1_deg ] = MadgwickFilter_IMU_calc( accelVect_1, gyroVect_1, loopTime, beta);

[ errorStats_m5_0, deriv_pitch_m5_0 ] = errorProcessing_calc(pitch_m5_0_deg, pitchTruth, loopTime);
[ errorStats_m5_1, deriv_pitch_m5_1 ] = errorProcessing_calc(pitch_m5_1_deg, pitchTruth, loopTime);

% Method #6: Mahony IMU Filter

Kp = 9.28649512050167E-6;
Ki = 6.395107547366333E-7;

[ pitch_m6_0_deg ] = MahonyFilter_IMU_calc( accelVect_0, gyroVect_0, loopTime, Kp, Ki);
[ pitch_m6_1_deg ] = MahonyFilter_IMU_calc( accelVect_1, gyroVect_1, loopTime, Kp, Ki);

[ errorStats_m6_0, deriv_pitch_m6_0 ] = errorProcessing_calc(pitch_m6_0_deg, pitchTruth, loopTime);
[ errorStats_m6_1, deriv_pitch_m6_1 ] = errorProcessing_calc(pitch_m6_1_deg, pitchTruth, loopTime);

% Method Novel #7/8: Milam Novel Method 1:

[ pitch_N1_deg ] = MilamIMU_1_calc(accelVect_0, accelVect_1, loopTime);
%[ pitch_N1_deg ] = MilamIMU_1_calc(accelVect_0, accelVect_1, loopTime); % Called twice for timing test

[ errorStats_N1, deriv_pitch_N1 ] = errorProcessing_calc(pitch_N1_deg, pitchTruth, loopTime);

% Calculate the error between the ground truth pitch rate and the average of % the two gyros.
[ deriv_errGyro ] = errRMS_calc( (gyroVect_0(:,2)+gyroVect_1(:,2))./2, deriv_pitchTruth' );

% Method Novel #9: Milam Novel Method 2:

Kc = 0.8218959581737024;

[ pitch_N2_deg ] = MilamIMU_2_calc(accelVect_0, accelVect_1, gyroVect_0, gyroVect_1, loopTime, Kc);
% [ pitch_N2_deg ] = MilamIMU_2_calc(accelVect_0, accelVect_1, gyroVect_0, gyroVect_1, loopTime, Kc); % called twice for timing test

[ errorStats_N2, deriv_pitch_N2] = errorProcessing_calc(pitch_N2_deg, pitchTruth, loopTime);

%% Error Processing:

% Putting the pitch errors from each algorithms into a vector:
i = 1; % i=1 is to select the RMS pitch errors
errRMS_Vect_0 = [ errorStats_m1_0(i), errorStats_m2_0(i), errorStats_m3_0(i), errorStats_m4_0(i), errorStats_m5_0(i), errorStats_m6_0(i), errorStats_N1(i), errorStats_N2(i)];
errRMS_Vect_1 = [ errorStats_m1_1(i), errorStats_m2_1(i), errorStats_m3_1(i), errorStats_m4_1(i), errorStats_m5_1(i), errorStats_m6_1(i), errorStats_N1(i), errorStats_N2(i)];

% Putting the pitch rate errors from each algorithms into a vector:
i = 5; % i=1 is to select the RMS pitch rate errors
deriv_errRMS_Vect_0 = [ errorStats_m1_0(i), errorStats_m2_0(i), errorStats_m3_0(i), errorStats_m4_0(i), errorStats_m5_0(i), errorStats_m6_0(i), errorStats_N1(i), deriv_errGyro, errorStats_N2(i)];
deriv_errRMS_Vect_1 = [ errorStats_m1_1(i), errorStats_m2_1(i), errorStats_m3_1(i), errorStats_m4_1(i), errorStats_m5_1(i), errorStats_m6_1(i), errorStats_N1(i), deriv_errGyro, errorStats_N2(i)];

% Normalizing the error vectors:
errRMS_Vect_0_norm = errRMS_Vect_0./sum(errRMS_Vect_0);
errRMS_Vect_1_norm = errRMS_Vect_1./sum(errRMS_Vect_1);
deriv_errRMS_Vect_0_norm = deriv_errRMS_Vect_0./sum(deriv_errRMS_Vect_0);
deriv_errRMS_Vect_1_norm = deriv_errRMS_Vect_1./sum(deriv_errRMS_Vect_1);

% Adding the normalized errors together:
errRMS_Vect_0_sum = errRMS_Vect_0_norm + deriv_errRMS_Vect_0_norm;
errRMS_Vect_1_sum = errRMS_Vect_1_norm + deriv_errRMS_Vect_1_norm;

% Finding the mean of the normalized errors between IMU #0 and #1:
errRMS_Vect_avg_sum = mean([errRMS_Vect_0_sum; errRMS_Vect_1_sum]);

% end of Script

File Name: gyroOnly_IMU_calc.m

function [ pitch_deg ] = gyroOnly_IMU_calc(gyroVect, loopTimeVect)
% gyroOnly_IMU_calc
% This function calculated the pitch estimate for the time series data
% based on only the gyrooscope measurements.

num_samples = length(loopTimeVect);
pitch_deg = zeros(1,num_samples);
for i=2:1:num_samples
    pitch_deg(i) = pitch_deg(i-1) + gyroVect(i-1,2) * loopTimeVect(i-1);
end
File Name: accelOnly_IMU_calc.m

-function [pitch_deg] = accelOnly_IMU_calc(accelVect)
    % accelOnly_IMU_calc
    % This function calculated the pitch estimate for the time series data
    % based on only the accelerometer measurements.
    pitch_deg = (atan2(accelVect(:,1), -accelVect(:,3))*180/pi)'; % (deg)
end

File Name: complementaryFilter_IMU_calc.m

-function [pitch_deg] = complementaryFilter_IMU_calc(accelVect, gyroVect, loopTimeVect, Kc)
    %complementaryFilter_IMU_calc
    % This function uses a complementary filter to fuse the gyro and
    % accelerometer time series measurements and calculate a pitch estimate.
    % "Kc" is the tuning variable.
    num_samples = length(loopTimeVect);
    pitch_accel_deg = (atan2(accelVect(:,1), -accelVect(:,3))*180/pi)'; % (deg)
    pitch_deg = zeros(1,num_samples);
    for i = 2:1:num_samples
        pitch_deg(i) = Kc*(pitch_deg(i-1) + gyroVect(i-1,2) * loopTimeVect(i-1)) + (1-Kc) * pitch_accel_deg(i);
    end
end

File Name: simpleKalman_IMU_calc.m

-function [pitch_deg] = simpleKalman_IMU_calc(accelVect, gyroVect, loopTimeVect, Q4, R4, x2, P00, P01, P11)
    %simpleKalman_IMU_calc
    % This function uses a Kalman filter to fuse the gyro and
    % accelerometer time series measurements and calculate a pitch estimate.
    % "Q4" and "R4" are the tuning variables. This code is based on code found
    % in Brandon McCarron's, "Low-Cost IMU Implementation via Sensor Fusion
    % Algorithms in the Arduino Environment" but, corrects some of the flaws
    % in McCarron's code.
    num_samples = length(loopTimeVect);
    pitch_deg = zeros(1,num_samples);
    pitch_accel_deg = (atan2(accelVect(:,1), -accelVect(:,3))*180/pi)'; % (deg)
    pitch_deg(1) = pitch_accel_deg(1);
loopTimeVect_alt = [loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect; loopTimeVect];
% This loop pre-calculates the values of P and K matrices. so that the % initial values do not affect the results:
for i=2:1:length(loopTimeVect_alt)
    P00 = P00+loopTimeVect_alt(i-1)*(-2*P01+loopTimeVect_alt(i-1)*P11)+Q4;
    P01 = P01-loopTimeVect_alt(i-1)*P11;
    P11 = P11+Q4;

    Kk0 = P00/(P00+R4);
    Kk1 = P01/(P00+R4);

    P11 = P11-Kk1*P01;
    P00 = P00*(1-Kk0);
    P01 = P01*(1-Kk0);
end

% This loop calculates the Kalman filter pitch estimate using the initial % covariance values calculated in the previous loop.
for i=2:1:num_samples
    pitch_deg(i) = pitch_deg(i-1)+loopTimeVect(i-1)*(gyroVect(i,2)-x2);

    P00 = P00+loopTimeVect(i-1)*(-2*P01+loopTimeVect(i-1)*P11)+Q4;
    P01 = P01-loopTimeVect(i-1)*P11;
    P11 = P11+Q4;

    Kk0 = P00/(P00+R4);
    Kk1 = P01/(P00+R4);

    x2 = x2+Kk1*(pitch_accel_deg(i)-pitch_deg(i));
    pitch_deg(i) = pitch_deg(i)+Kk0*(pitch_accel_deg(i)-pitch_deg(i));

    P11 = P11-Kk1*P01;
    P00 = P00*(1-Kk0);
    P01 = P01*(1-Kk0);
end
end

File Name: MadgwickFilter_IMU_calc.m

function [ pitch_deg ] = MadgwickFilter_IMU_calc( accelVect, gyroVect, loopTimeVect, beta )
%MadgwickFilter_IMU_calc
% This function uses the Madgwick algorithm to fuse the gyro and
% accelerometer time series measurements and calculate a pitch estimate.
% "beta" is the tuning variable. This code is based on code from
% http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms.

num_samples = length(loopTimeVect);

SE = zeros(1,4);
SE(1) = 1;

pitch_deg = zeros(1,num_samples);

pitch_accel_deg = (atan2(accelVect(1,1),-accelVect(1,3))*180/pi)'; % (deg)
pitch_deg(1) = pitch_accel_deg;

for i=2:1:num_samples
    [ SE ] = MadgwickIMU_filterUpdate( SE, accelVect(i-1,:), gyroVect(i-1,:),
                                      beta, loopTimeVect(i-1) );

    temp = quatern2euler(SE);
    pitch_deg(i) = temp(2)*(180/pi);
end

end

File Name: MadgwickIMU_filterUpdate.m

function [ SE ] = MadgwickIMU_filterUpdate( SE, accel, gyro, beta, deltat )
% MadgwickIMU_filterUpdate
% This function performs the update of the Madgwick algorithm estimate
% for every sample, to fuse the gyro and accelerometer time series
% measurements and calculate a pitch estimate.
% This code is based on code from
% http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms.

SEq_1 = SE(1);
SEq_2 = SE(2);
SEq_3 = SE(3);
SEq_4 = SE(4);

a_x = accel(1);
a_y = -accel(2); % The negative sign appears necessary to put into Aero
               % Coordinate Frame.
a_z = -accel(3); % The negative sign appears necessary to put into Aero
               % Coordinate Frame.

gyro = gyro * pi/180; % (rad/s) Converting to rad/sec
w_x = gyro(1);
w_y = -gyro(2); % The negative sign appears necessary to put into Aero
               % Coordinate Frame.
w_z = -gyro(3); % The negative sign appears necessary to put into Aero
               % Coordinate Frame.

%// Auxiliary variables to avoid repeated calculations
halfSEq_1 = 0.5 * SEq_1;
halfSEq_2 = 0.5 * SEq_2;
halfSeq_3 = 0.5 * SEq_3;
halfSeq_4 = 0.5 * SEq_4;
twoSeq_1 = 2.0 * SEq_1;
twoSeq_2 = 2.0 * SEq_2;
twoSeq_3 = 2.0 * SEq_3;

%// Normalize the accelerometer measurement
normal = sqrt(a_x * a_x + a_y * a_y + a_z * a_z);
a_x = a_x/normal;
a_y = a_y/normal;
a_z = a_z/normal;

%// Compute the objective function and Jacobian
f_1 = twoSeq_2 * SEq_4 - twoSeq_1 * SEq_3 - a_x;
J_11or24 = twoSeq_3;
J_12or23 = 2.0 * SEq_4;
J_13or22 = twoSeq_1;
J_14or21 = twoSeq_2;

J_22 = 2.0 * J_14or21;
J_23 = 2.0 * J_11or24;

J = [J_11or24 J_12or23 J_13or22 J_14or21; J_12or23 J_13or22 J_14or21 J_11or24; J_13or22 J_14or21 J_11or24 J_12or23; J_14or21 J_11or24 J_12or23 J_13or22];
SEqHatDot_1 = J_14or21 * f_2 - J_11or24 * f_1;
SEqHatDot_2 = J_12or23 * f_1 + J_13or22 * f_2 - J_32 * f_3;
SEqHatDot_3 = J_12or23 * f_2 - J_33 * f_3 - J_13or22 * f_1;
SEqHatDot_4 = J_14or21 * f_1 + J_11or24 * f_2;

%// Compute the gradient (matrix multiplication)
SEqHatDot_1 = J_14or21 * f_2 - J_11or24 * f_1;

%// Normalize the gradient
normal = sqrt(SEqHatDot_1 * SEqHatDot_1 + SEqHatDot_2 * SEqHatDot_2 + SEqHatDot_3 * SEqHatDot_3 + SEqHatDot_4 * SEqHatDot_4);

%// Compute the quaternion derivative measured by gyroscopes
SEqDot_omega_1 = -halfSeq_2 * w_x - halfSeq_3 * w_y - halfSeq_4 * w_z;
SEqDot_omega_2 = halfSeq_1 * w_x + halfSeq_3 * w_z - halfSeq_4 * w_y;
SEqDot_omega_3 = halfSeq_1 * w_y - halfSeq_2 * w_z + halfSeq_4 * w_x;
SEqDot_omega_4 = halfSeq_1 * w_z + halfSeq_2 * w_y - halfSeq_3 * w_x;

%// Compute then integrate the estimated quaternion derivative
SEq_1 = SEq_1+(SEqDot_omega_1 - (beta * SEqHatDot_1)) * deltat;
SEq_2 = SEq_2+(SEqDot_omega_2 - (beta * SEqDot_omega_2)) * deltat;
SEq_3 = SEq_3+(SEqDot_omega_3 - (beta * SEqDot_omega_3)) * deltat;
SEq_4 = SEq_4+(SEqDot_omega_4 - (beta * SEqDot_omega_4)) * deltat;

%// Normalize quaternion
normal = sqrt(SEq_1 * SEq_1 + SEq_2 * SEq_2 + SEq_3 * SEq_3 + SEq_4 * SEq_4);
SEq_1 = SEq_1/ normal;
SEq_2 = SEq_2/ normal;
SEq_3 = SEq_3/ normal;
SEq_4 = SEq_4/ normal;
File Name: quaternion2euler.m

function euler = quaternion2euler(q)
%QUATERN2EULER Converts a quaternion orientation to ZYX Euler angles
% q = quaternion2euler(q)
% Converts a quaternion orientation to ZYX Euler angles where phi is a rotation around X, theta around Y and psi around Z.
% For more information see:
% http://www.x-io.co.uk/node/8#quaternions

% Date Author Notes
% 27/09/2011 SOH Madgwick Initial release

R(1,1,:) = 2.*q(:,1).^2-1+2.*q(:,2).*q(:,2);
R(2,1,:) = 2.*(q(:,2).*q(:,3)-q(:,1).*q(:,4));
R(3,1,:) = 2.*(q(:,2).*q(:,4)+q(:,1).*q(:,3));
R(3,2,:) = 2.*(q(:,3).*q(:,4)-q(:,1).*q(:,2));
R(3,3,:) = 2.*q(:,1).^2-1+2.*q(:,4).*q(:,4);

phi = atan2(R(3,2,:), R(3,3,:));
theta = -atan(R(3,1,:) ./ sqrt(1-R(3,1,:).^2) );
psi = atan2(R(2,1,:), R(1,1,:));
euler = [phi(1,:)' theta(1,:)' psi(1,:)'];
end

File Name: MahonyFilter_IMU_calc.m

function [ pitch_deg ] = MahonyFilter_IMU_calc( accelVect, gyroVect, loopTimeVect, Kp, Ki )
%MahonyFilter_IMU_calc
% This function uses the Mahony algorithm to fuse the gyro and accelerometer time series measurements and calculate a pitch estimate.
% "Kp" and "Ki" are the tuning variables. This code is based on code from
% http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms.

num_samples = length(loopTimeVect);

SE = zeros(1,4);
SE(1) = 1;
eInt = [0 0 0];
pitch_deg = zeros(1,num_samples);

pitch_accel_deg = (atan2(accelVect(1,1), -accelVect(1,3))*180/pi)'; % (deg)
pitch_deg(1) = pitch_accel_deg;

for i=2:1:num_samples
    [ SE, eInt ] = MahonyIMU_filterUpdate( SE, accelVect(i-1,:), gyroVect(i-1,:), Kp, Ki, eInt, loopTimeVect(i-1) );
    temp = quatern2euler(SE);
pitch_deg(i) = temp(2)*(180/pi);
end
end

**File Name:** MahonyFilter_IMU_calc.m

```matlab
function [ SE, eInt ] = MahonyIMU_filterUpdate( SE, accel, gyro, Kp, Ki, eInt, deltat )
% MahonyIMU_filterUpdate
% This function performs the update of the Mahony algorithm estimate
% for every sample, to fuse the gyro and accelerometer time series
% measurements and calculate a pitch estimate.
% This code is based on code from
% http://www.x-io.co.uk/node/8#open_source_ahrs_and_imu_algorithms.

q = SE; % short name local variable for readability
Accelerometer(1) = accel(1);
Accelerometer(2) = -accel(2); % This is to correct for non-Aero Coordinate system
Accelerometer(3) = -accel(3);
Gyroscope(1) = gyro(1);
Gyroscope(2) = -gyro(2);
Gyroscope(3) = -gyro(3);
Gyroscope = Gyroscope*(pi/180);

% Normalize accelerometer measurement
if(norm(Accelerometer) == 0), return; end % handle NaN
Accelerometer = Accelerometer / norm(Accelerometer); % normalize magnitude

% Estimated direction of gravity and magnetic flux
v = [2*(q(2)*q(4) - q(1)*q(3))
     2*(q(1)*q(2) + q(3)*q(4))
     q(1)^2 - q(2)^2 - q(3)^2 + q(4)^2];

% Error is sum of cross product between estimated direction and measured direction of field
e = cross(Accelerometer, v);
if(Ki > 0)
eInt = eInt + e * deltat;
else
eInt = [0 0 0];
end
```
% Apply feedback terms
Gyroscope = Gyroscope + Kp * e + Ki * eInt;

% Compute rate of change of quaternion
qDot = 0.5 * quaternProd(q, [0 Gyroscope(1) Gyroscope(2) Gyroscope(3)]);

% Integrate to yield quaternion
q = q + qDot * deltat;
q = q / norm(q); % normalize quaternion

SE(1) = q(1);
SE(2) = q(2);
SE(3) = q(3);
SE(4) = q(4);

File Name: quoternProd.m

function ab = quaternProd(a, b)
%QUATERNPROD Calculates the quaternion product
% ab = quaternProd(a, b)
% Calculates the quaternion product of quaternion a and b.
% For more information see:
% http://www.x-io.co.uk/node/8#quaternions
% 27/09/2011 SOH Madgwick Initial release

   ab(:,1) = a(:,1).*b(:,1)-a(:,2).*b(:,2)-a(:,3).*b(:,3)-a(:,4).*b(:,4);
   ab(:,2) = a(:,1).*b(:,2)+a(:,2).*b(:,1)+a(:,3).*b(:,4)-a(:,4).*b(:,3);
   ab(:,3) = a(:,1).*b(:,3)-a(:,2).*b(:,4)+a(:,3).*b(:,1)+a(:,4).*b(:,2);
   ab(:,4) = a(:,1).*b(:,4)+a(:,2).*b(:,3)-a(:,3).*b(:,2)+a(:,4).*b(:,1);
end

File Name: MilamIMU_1_calc.m

function [ pitch_deg ] = MilamIMU_1_calc(accelVect_0, accelVect_1, loopTimeVect)
%MilamIMU_1_calc
% This function determines the pitch estimate from the modeled combination of the two IMUs.

num_samples = length(loopTimeVect);
rx0 = 0.08;
rz0 = 0.23;
rx1 = 0.05;
rz1 = 0.035;

AI00 = (rz0 - rz1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);
AI01 = (rx0 + rx1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);
AI02 = (rx1^2 + rx0*rx1 + rz1^2 - rz0*rx1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);
AI20 = -(rx0*rz1 + rx1*rz0)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);
AI21 = -(rx0*rz1 + rx1*rz0)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);
AI22 = (rx0^2 + rx1*rx0 + rz0^2 - rz1*rz0)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rx1 + rz1^2);

A = [ rz0, -rx0, 1, 0, 0, 0;
     rx0, rz0, 0, 1, 0, 0;
     rz1, rx1, 0, 0, 1, 0;
     -rx1, rz1, 0, 0, 0, 1;
     0, 0, 1, 0, -1, 0;
     0, 0, 0, 1, 0, -1];

g = 9.81;
x = zeros(6,num_samples);
pitch_deg = zeros(1, num_samples);
for i = 1:num_samples
  b = [accelVect_0(i,1); accelVect_0(i,3); accelVect_1(i,1);
       accelVect_1(i,3); 0; 0];
  b = 9.81.*b;
  x(:,i) = A\b;
  b1 = g*accelVect_0(i,1);
  b2 = g*accelVect_0(i,3);
  b3 = g*accelVect_1(i,1);
  b4 = g*accelVect_1(i,3);
  x(:,i) = [AI00*b1 - AI00*b3 + AI01*b2 - AI01*b4;
            AI00*b2 - AI01*b1 - AI00*b4 + AI01*b3;
            AI20*b1 + AI21*b2 - AI21*b4 + AI22*b3;
            AI20*b2 - AI21*b1 + AI21*b3 + AI22*b4;
            AI20*b1 + AI21*b2 - AI21*b4 + AI22*b3;
            AI20*b2 - AI21*b1 + AI21*b3 + AI22*b4];
  pitch_deg(i) = (atan2(x(3,i), -x(4,i))*180/pi)'; % (deg) Note that
  x(3,:) = x(5,:)
  x(4,:) = x(6,:)
end

end

File Name: MilamIMU_2_calc.m
function [ pitch_deg2 ] = MilamIMU_2_calc(accelVect_0, accelVect_1, gyroVect_0, gyroVect_1, loopTimeVect, Kc)
%MilamIMU_2_calc
%   This function determines the pitch estimate from the modeled
%   combination of the two IMUs. This estimate is then fused with the mean
%   of the gyro measurements from the gyros on both IMUs via the
%   complementary filter.

num_samples = length(loopTimeVect);

% These are the distances between the IMUs and the CG in the x and z
% directions in meters.
rx0 = 0.08;
rz0 = 0.23;
rx1 = 0.05;
rz1 = 0.035;

AI00 = (rz0 - rz1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rz1 + rz1^2);
AI01 = (rx0 + rx1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rz1 + rz1^2);
AI20 = (rx1^2 + rx0*rx1 + rz1^2 - rz0*rz1)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rz1 + rz1^2);
AI21 = -(rx0*rz1 + rx1*rz0)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rz1 + rz1^2);
AI22 = (rx0^2 + rx1*rx0 + rz0^2 - rz1*rz0)/(rx0^2 + 2*rx0*rx1 + rx1^2 + rz0^2 - 2*rz0*rz1 + rz1^2);

g = 9.81;
x = zeros(6,num_samples);
pitch_deg = zeros(1, num_samples);
for i = 1:num_samples
%     b = [accelVect_0(i,1); accelVect_0(i,3); accelVect_1(i,1);
%          accelVect_1(i,3); 0; 0];
%     b = 9.81.*b;
%     x(:,i) = A\b;
%     x(:,i) = A_inv*b;
%     pitch_deg(i) = (atan2(x(3,i), -x(4,i))*180/pi)'; % (deg)
    b1 = g*accelVect_0(i,1);
    b2 = g*accelVect_0(i,3);
    b3 = g*accelVect_1(i,1);
    b4 = g*accelVect_1(i,3);

    x(:,i) = [AI00*b1 - AI00*b3 + AI01*b2 - AI01*b4; %
               AI00*b2 - AI01*b1 - AI00*b4 + AI01*b3; %
               AI20*b1 + AI21*b2 - AI21*b4 + AI22*b3; %
               AI20*b2 - AI21*b1 + AI21*b3 + AI22*b4; %
               AI20*b1 + AI21*b2 - AI21*b4 + AI22*b3; %
               AI20*b2 - AI21*b1 + AI21*b3 + AI22*b4]; %

    pitch_deg(i) = (atan2(x(3,i), -x(4,i))*180/pi)'; % (deg)
end

x3 = AI20*b1 + AI21*b2 - AI21*b4 + AI22*b3;
x4 = AI20*b2 - AI21*b1 + AI21*b3 + AI22*b4;

pitch_deg(i) = (atan2(x3, -x4)*180/pi)';
Finding the mean of the gyro measurements in all three axes

```matlab
gyroVect_avg = zeros(num_samples, 3);
gyroVect_avg(:,1) = mean([gyroVect_0(:,1)'; gyroVect_1(:,1)']);  % X
gyroVect_avg(:,2) = mean([gyroVect_0(:,2)'; gyroVect_1(:,2)']);  % Y
gyroVect_avg(:,3) = mean([gyroVect_0(:,3)'; gyroVect_1(:,3)']);  % Z
```

% Using the Complementary filter to fuse the measurements from the accelerometers and gyros.

```matlab
pitch_deg2 = zeros(1,num_samples);
for i = 2:1:num_samples
    pitch_deg2(i) = Kc*(pitch_deg2(i-1) + (gyroVect_avg(i-1,2)) * loopTimeVect(i-1)) + (1-Kc) * pitch_deg(i);
end
```

**File Name:** errorProcessing_calc.m

```matlab
function [ errorStats, measuredDerivVect ] = errorProcessing_calc(measuredVect, truthVect, timeVect)
%errorProcessing_calc
%   This function takes a pitch estimate and compares it to the ground truth pitch, calculating the RMS error, standard deviation, average absolute deviation, and variance. It then does the same comparison and calculates the same statistics for the pitch rate estimate.
	numSamples = length(truthVect);

if size(measuredVect) == flip(size(truthVect))
    truthVect = truthVect';
end

% Pitch estimate
% Error:
errorVect = measuredVect-truthVect;

% RMS Error:
errorRMS = rms(errorVect);

% Standard Deviation:
errorSTD = std(errorVect);

% Average Absolute Deviation:
errorMAD = mad(errorVect);

% Variance:
errorVAR = var(errorVect);

% Pitch Rate estimate
measuredDerivVect = zeros(1, numSamples);
truthDerivVect    = zeros(1, numSamples);
% Calculating the pitch rate estimate.
for i = 2:numSamples
```
measuredDerivVect(i) = (measuredVect(i) - measuredVect(i-1)) / timeVect(i-1);
truthDerivVect(i) = (truthVect(i) - truthVect(i-1)) / timeVect(i-1);
end

% Derivative Error:
errorDerivVect = measuredDerivVect - truthDerivVect;

% RMS Derivative Error:
errorDerivRMS = rms(errorDerivVect);

% Standard Deviation of Derivative Error;
errorDerivSTD = std(errorDerivVect);

%Average Absolute Deviation of Derivative Error:
errorDerivMAD = mad(errorDerivVect);

% Variance:
errorDerivVAR = var(errorDerivVect);

errorStats = [ errorRMS; errorSTD; errorMAD; errorVAR; errorDerivRMS; errorDerivSTD; errorDerivMAD; errorDerivVAR ];
end

File Name: errRMS_calc.m

function [ errRMS ] = errRMS_calc( vect1, vect2 )
%errRMS_calc
%   A simple function to calculate the RMS error between to vectors.
numSamples = length(vect2);
if size(vect1) == fliplr(size(vect2))
    vect2 = vect2';
end
errRMS = sqrt( (sum( ((vect2 - vect1).^2) )/numSamples) );
end