Fall 2011

Design and development of a debris flow tracking "Smart Rock"

Matthew J. Harding
University of New Hampshire, Durham

DESIGN AND DEVELOPMENT OF A DEBRIS FLOW TRACKING "SMART ROCK"

BY

MATTHEW J. HARDING
B.S., University of New Hampshire, 2010

THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Master of Science
in
Mechanical Engineering

September, 2011

This thesis has been examined and approved.

Thesis Director, Prof. Barry Fussell
Professor of Mechanical Engineering

Prof. Jean Benoît
Professor of Civil Engineering

Prof. May-Win Thein
Associate Prof. of Mechanical Engineering

2 August 2011

Date

DEDICATION

In memory of Professor Pedro de Alba.

"Lo bailado nadie te lo quita"
"The dance you have danced no one can take away from you"

April 2, 1939 - February 20, 2011

ACKNOWLEDGMENTS

There are many people to whom I owe thanks for aiding in the completion of this project. First is to my parents and family for their love and support with everything that came my way over the years. Of course, to my beautiful fiancé and soulmate, Amanda, who has already given me more than I could have ever asked for. As well as to her family for their encouragement and assistance in both of our journeys through academics and through life.

There were several people directly involved in initiating this project and helping it to progress: Prof. Barry Fussell, Prof. Pedro de Alba, Prof. Jean Benoît, Prof. May-Win Thein, and Richard LaHusen. In addition, several other members of the UNH faculty, staff, and student body provided assistance in varying capacities with every aspect. This includes Adam Perkins, Sheldon Parent, Sean Wadsworth, Prof. Radim Bartos, Prof. L.G. Kraft, Graham Cullen, and many others including those who had begun the project.

There are also a few thanks due to others outside of the University community including Mike Dawson of Segway Inc., Mark Looney of Analog Devices, and Cheri Marianetti of Mazzer Plastics.

Without the help of all of these people, as well as many others, this could not have been completed and I could not have made it to where I am today.

Thank you.

Contents

List of Tables

List of Figures

Nomenclature

Inertial Navigation

t	Time $[s]$
$\boldsymbol{\omega}$	Rotation rate in body coordinates $[rad/s]$
$\boldsymbol{\sigma}$	Incremental rotation angle $[rad]$
Σ	Skew symmetric matrix of rotation angles $[rad]$
$\mathbf{q}$	Attitude quaternion [-]
C	Direction cosine attitude representation [-]
A	Matrix transforming body axes at time t_k to body axes at t_{k+1} [-]
$\mathbf{f}^{\mathbf{b}}$	Body force in body coordinates $[m/s^2]$
$\mathbf{f}^{\mathbf{g}}$	Body force in local geographic coordinates $[m/s^2]$
$\mathbf{g}$	Gravity vector in local geographic coordinates $[m/s^2]$
$\mathbf{v}$	Velocity in local geographic coordinates $[m/s]$
L	Latitude $[^o]$
$\boldsymbol{\Omega}_{\boldsymbol{ie}}$	Matrix to correct for the earth's rotation $[rad/s]$
$\boldsymbol{\Omega}_{\boldsymbol{en}}$	Matrix to correct for the Coriolis effect $[rad/s]$

Allan Variance

τ	Averaging time (bin size)
y	Bin averages
$\sigma_A^2, AVAR$	Allan Variance
$\sigma_A, ADEV$	Allan Deviation
ARW	Angle Random Walk
VRW	Velocity Random Walk
N	Angle random walk coefficient
B	Bias instability coefficient
BW	Sensor bandwidth
σ_{white}	Standard deviation of white noise process
σ_{pink}	Standard deviation of pink noise process

Instrumentation Communication/Calibration

SPI	Serial Peripheral Interface
SS, CS	Slave Select,Chip Select
$SCLK$	Clock
$MOSI$	Master Out Slave In
$MISO$	Master In Slave Out
DIN	Digital In
$DOUT$	Digital Out
IRQ	Interrupt Request
$CPOL$	Clock Polarity
$CPHA$	Clock Phase
LSB	Least Significant Bit
$U_{+,-}$	Accelerometer ouput with positive/negative input $[m/s^2]$
K	Accelerometer sensitivity matrix $[m/s^2/LSB]$
B	Accelerometer bias matrix $[m/s^2]$
b_{axis}	Accelerometer axis bias $[m/s^2]$
PCB	Printed Circuit Board

ABSTRACT

DESIGN AND DEVELOPMENT OF OF A DEBRIS FLOW TRACKING "SMART ROCK"

by

Matthew J. Harding

University of New Hampshire, September, 2011

This thesis covers the design and development of a Smart Rock sensor package to be used in U.S. Geological Survey research of debris flow phenomena. This instrumented Rock, containing inertial sensors and pressure sensors, provides information about the movement of the flow as well as the pressures seen within the flow. The goal of the sensor package is to use this information to track the position of a particle in the flow with an accuracy of 1 m over the course of 10 s.

It is found that using an ad hoc filtering method provides the required level of accuracy. Any known positions are incorporated into the filter. Large scale motions over tens of meters can be distinguished as well as small scale motions on the order of centimeters. Thus, the data gathered by the Smart Rock can be used to help verify and refine debris flow models.

Chapter 1

Introduction

1.1 Background

At 7 a.m. on the morning of May 18, 1980, a U.S. Geological Survey volcanologist reported on the measurements he had just taken of Mount St. Helens from his post six miles north: no changes from the trends of the previous month. There was nothing out of the ordinary in the ground temperature readings, sulfur dioxide emissions, or ground movement compared to the previous month.

At 8:32 that morning, a moderate 5.1 magnitude earthquake struck, centered in the southwest of Washington State. Within just 15 seconds, this triggered the largest landslide in recorded history. The volcano's bulge and summit broke into three enormous masses and began to slide north towards Spirit Lake. With the debris moving at speeds of up to 150 miles per hour, 23 square miles were buried by 3.7 billion cubic yards of debris. A 14 mile stretch of the North Fork Toutle River Valley was buried at an average depth of 150 feet. Just below Spirit Lake, the debris was the thickest with a depth of one mile [1] (see Figure 1.1).

Figure 1.1: Mount St. Helens landslide destruction (courtesy of USGS)

The beginning of this slide also triggered the subsequent eruption. This catastrophic event not only changed the landscape of the entire area, but it brought this danger to the attention of the general public and to relevant officials.

Clearly, debris flow phenomena merit a level of scientific research and investigation. A greater understanding of these types of flows can increase awareness and help to mitigate their effects. Theoretical models have been developed for overall movement of this type of fluid mass, leading to some understanding of the underlying processes However, experimental verification of these models proves to be challenging.

In 1991, the U.S. Geological Survey (USGS) constructed a concrete flume in the Willamette National Forest, OR, to recreate debris flow phenomena. In this facility, controlled experiments are conducted to carefully record data on pressure, depth, shear stress, etc. within the flowing mass. This instrumentation only allows for measurements in fixed physical locations at the bed floor or on the surface of the flow.

Observations from these experiments show that the coarser debris tends to move outwards during the flow, forming "levees" which channelize the interior finer debris, allowing this finer debris to run for longer distances. The mechanics of this behavior are still not well understood and cannot be investigated using the tools that are currently available in the flume. Consequently, the ability to measure internal pressures, debris accelerations and to track the flow would be an invaluable tool.

The USGS has begun to undertake this project. Small, rugged, instrumented packages were created to be placed in the flume to measure acceleration. Richard LaHusen, a volcanologist at U.S.G.S., has developed small "Smart Rocks" using a MEMS accelerometer and a Rabbit microcontroller to log the data. This project is to be expanded upon.

1.2 Objective

The objective of this research is the development of an instrumented "rock" to be placed in the debris flow to track particle movement. The proposed Smart Rock package will include a six-degree-of-freedom inertial measurement unit (IMU) to measure three axes of acceleration as well as rotation about three axes. In addition, two pressure sensors will be instrumented to measure fluid pressure on the surface of the Rock. This information will be stored on-board on a μSD card for post-processing and analysis. The instrumentation must be powered and be completely enclosed in a durable and impermeable metal shell. It is desired that the entire device approximate the size and density of a representative piece of the coarser debris that will be in the flume.

The inertial measurements of acceleration and rotation rate can be processed through a series of navigation calculations to find the position and orientation over time. The inertial navigation process introduces many challenges. Because the measurements must be integrated in this calculation, twice for the acceleration data to transform into position, the errors are also integrated. This means, any noise in the signals can cause position errors that grow very quickly over time. This is especially true of any uncorrected bias on the signals as is described later.

Microelectromechanical systems (MEMS) sensors will allow the device to be compact, low power, and inexpensive. However, their accuracy and noise characteristics are less desirable than sensors made with other technologies. Without additional sensors such as a magnetometer, GPS, or sun tracker, the error in position will grow exponentially. This will severely limit the amount of time over which the calculated position will be accurate.

Experiments at the U.S.G.S. flume show that accelerations of approximately 10 g may appear in the slide. Thus, the dynamic range of the accelerometer must surpass this. The rock rotation rates have not been measured and are estimated to be minimal, not exceed 200 rpm for short durations or 30 rpm for sustained periods. The slide created by the flume lasts only up to 15 s. With this information the tracking goal has been set at tracking position within 1 m over 10 s. It is also desired to record data from the inertial sensors as well as the fluid pressure sensors for a duration of at least 10 minutes. From the data gathered by the Smart Rock, theoretical models of the debris flow can be verified and refined.

1.3 Thesis Overview

Chapter 1 discussed the background and motivation for this project. Next, Chapter 2 introduces the process by which the position is calculated from accelerometer and gyroscope measurements. Chapter 3 discusses the components of the "Smart Rock" package including inertial sensors, pressure sensors, microprocessor and firmware, along with the sensor calibration. Next, noise sources and the Allan Variance method for characterizing these different types of noise are introduced in Chapter 4. These methods are employed on the inertial sensors to quantify their inherent noise. Simulations using the results of the noise analysis are performed in Chapter 5 along with testing of the sensor package to determine accuracy. It is found that MEMS sensors are not adequate to provide the desired accuracy. Thus an ad hoc filtering method is developed in order to constrain the error and improve accuracy. This is discussed in Chapter 6. Final testing of the package with the newly developed filtering algorithm and results are presented in Chapter 7. Conclusion and recommendations for future work are summarized in Chapter 8.

Chapter 2

Inertial Navigation Basics

2.1 Introduction

The concepts of inertial navigation are central to the development and implementation of the Smart Rock. In this chapter, we begin by discussing the basics of inertial navigation and the different physical implementations of inertial navigation systems. This is followed by the strapdown navigation algorithm, including calculation of attitude and position using quaternion representation.

2.2 Inertial Navigation Implementations

Inertial navigation uses the inertial properties of sensors in order to calculate the position and orientation of the system being analyzed. As summarized by Britting [2]:

> "All inertial navigation systems must perform the following functions:
>
> - Instrument a reference frame

- Measure specific force
- Have knowledge of the gravitational field
- Time integrate the specific force data to obtain velocity and position information"

This first function is performed by gyroscopes which measure rotation rate by examining the effect of rotation on a vibrating mass. The gyroscope output is proportional to rotation rate about the sensing axis. From the information regarding rotation about three mutually perpendicular axes, the orientation, or attitude, can be found.

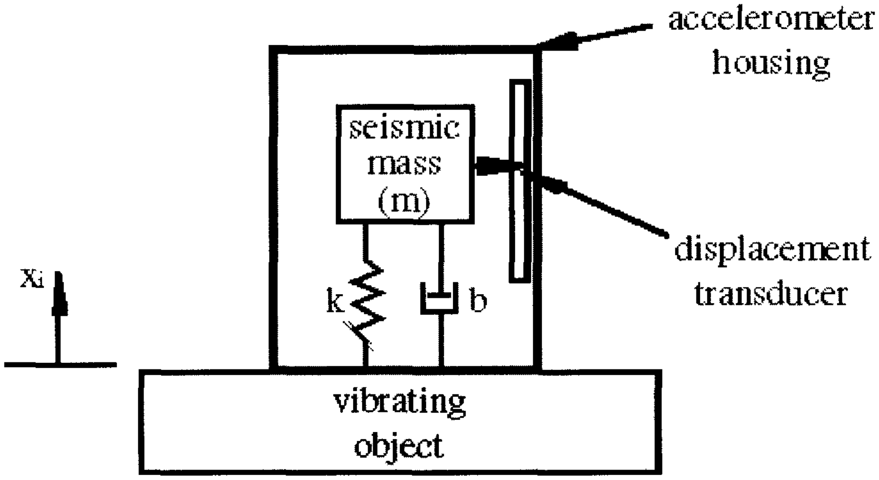


Figure 2.1: Accelerometer diagram [3]

Measurement of specific force is performed by accelerometers. Accelerometers can take several forms, but many are variations of the simple pendulum as shown in Figure 2.1. The motion or deflection of this mass can be related to the acceleration of the body to which it is mounted. However, accelerometers are not able to distinguish between inertial acceleration and a gravitational field. This means that information about the gravitational field must be known to compensate for this being added to the acceleration measurement.

Finally, with known attitude and acceleration, the necessary integrations can be performed to compute position over time. Typically, this is performed by an on-board processor in real-time.

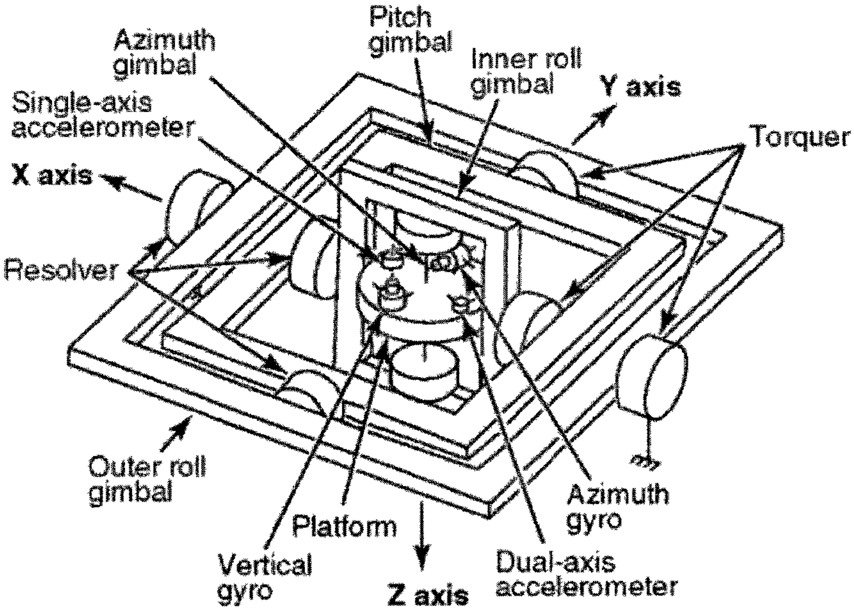


Figure 2.2: Gimballed IMU diagram [4]

There are traditionally three ways to implement this inertial navigation system. The first, and oldest, is called geometric. This uses at least five mechanical gimbals, as shown in Figure 2.2, to orient the sensors as needed, keeping the axes in a constant configuration via a servo motor system. This allows the measurements of angular displacement to be taken directly from the relative position of the gimbals. The next system, called semi-analytic, is similar to a geometric system but uses only three gimbals.

Finally, an analytic system uses no gimbals and has the inertial sensors directly attached to the body in motion. This requires the most computation as direct measurements of attitude are no longer available, and it is the least accurate of the three configurations. However, the mechanical gimbals and the associated servo positioning system require a significant amount of physical space to instrument whereas the analytic, or strapdown, system can be implemented in very small packages. Because of the physical size, short duration of time that will be investigated, and because the computations do not need to be performed in real-time, a strapdown system will be used and this will be the only type of system considered throughout the rest of this thesis.

2.2.1 Unaided Inertial Navigation

The system described here uses only rate gyros and accelerometers, giving relative inertial measurements (measurements of changes in attitude and position). Typically, an inertial navigation system uses some other sensor to measure absolute position or attitude. In a terrestrial or aerospace application, this could be a global positioning system (GPS) or magnetometer. For a satellite or spacecraft, this sensor could be a star tracker or sun sensor, giving information on the orientation with respect to fixed celestial bodies. These sensors are not practical in this application due to space constraints, depth under debris, and proximity to ferrous material. The lack of additional sensors proves to be a difficult challenge as discussed in the subsequent chapters.

2.3 Strapdown Navigation Algorithms

Strapdown, or analytic, inertial navigation takes measurements of rotation rate and acceleration along with the initial state, and calculates position and attitude over time. The calculation algorithm to track an object instrumented with strapdown inertial sensors is fairly straightforward. This process assumes the axes of the triaxial gyroscope and triaxial accelerometer are aligned in what is to be referred to as the "body axes". An overview is given here, and this algorithm is discussed more in depth in Appendix A.

2.3.1 Attitude Calculations

Attitude Representations

To calculate attitude over time, the only measurements needed are those from the three gyroscopes. These measurements can be taken and processed using three primary attitude representations:

1. Direction Cosines: A 3×3 direction cosine matrix (DCM) with columns representing unit vectors in body axes expressed in reference coordinates.

2. Euler Angles: A coordinate transformation can be described as three successive rotations about different axes. Euler angles represent these angles which correspond to angles on mechanical gimbals.

3. Quaternions: A coordinate transformation can also be described by a single rotation about a vector in the reference coordinates.

Euler angles are theoretically correct, but will not be used because they have inherent singularities at certain angles. The quaternion method and direction cosine method are generally regarded as the most accurate, each having their own merits [5]. Accuracy is generally not the only criterion for selecting attitude representation; processing efficiency is also very important for systems that are navigating in real-time. While this application does not require real-time processing, quaternion representation is still chosen over direction cosines because of the computational efficiency, and simple correction to adjust for numerical inaccuracies.

Quaternion Algorithm

A quaternion is a four-element tensor which can be thought of as a three-dimensional vector, corresponding to a vector in 3-D space, appended by a scalar. The quaternion can also be regarded as a complex number with one real part and three complex parts, i, j and k. In the context of inertial navigation, the quaternion is defined as follows:

$$\mathbf{q} = \begin{bmatrix} \cos\left({}^{\mu}/_{2}\right) \\ (\mu_x/\mu)\sin\left({}^{\mu}/_{2}\right) \\ (\mu_y/\mu)\sin\left({}^{\mu}/_{2}\right) \\ (\mu_z/\mu)\sin\left({}^{\mu}/_{2}\right) \end{bmatrix} \tag{2.1}$$

where

$$\boldsymbol{\mu} = \begin{bmatrix} \mu_x & \mu_y & \mu_z \end{bmatrix}^T \tag{2.2}$$

and

$$\mu = ||\boldsymbol{\mu}||. \tag{2.3}$$

Notice how $\mathbf{q}$ is defined, with a scalar and three components relating to the components of a 3-D vector.

To transform one coordinate system to another, the first coordinate system can be rotated about a single vector, $\boldsymbol{\mu}$, by an angle of magnitude μ, into the final coordinate system. Note that this definition differs from the traditional mathematical definition of the quaternion. The rotation magnitude, here shown as μ, is typically a separate variable in applications outside of inertial navigation.

As we can see, the last three elements of the quaternion are proportional to the three components of $\boldsymbol{\mu}$, while the first element is proportional to the magnitude of $\boldsymbol{\mu}$. One nice property of the quaternion is that it always has a magnitude of unity. This allows for a simple correction at any point in time. If the quaternion is divided by its magnitude, it will then have the required unity magnitude. Because computational speed is not a priority in this application, this correction can be applied at every time step.

The quaternion attitude algorithm is shown in equations 2.4 - 2.7.

1. Compute the rotation angle, σ, over the update time interval by integrating the rotation rate, ω.

$$\boldsymbol{\sigma} = \int_{i}^{i+1} \boldsymbol{\omega}\mathrm{dt} = \begin{bmatrix} \sigma_x \\ \sigma_y \\ \sigma_z \end{bmatrix} \tag{2.4}$$

2. Arrange the skew symmetric Σ matrix from these rotation angles.

$$\Sigma = \begin{bmatrix} 0 & -\sigma_x & -\sigma_y & -\sigma_z \\ \sigma_x & 0 & \sigma_z & -\sigma_y \\ \sigma_y & -\sigma_z & 0 & \sigma_x \\ \sigma_z & \sigma_y & -\sigma_x & 0 \end{bmatrix} \tag{2.5}$$

3. Update the quaternion with this Σ matrix.

$$\mathbf{q_{i+1}} = \exp\left(\frac{\Sigma}{2}\right)\mathbf{q_i} \tag{2.6}$$

4. Finally, make any necessary magnitude correction.

$$\mathbf{q_{i+1}} = \frac{\mathbf{q_{i+1}}}{||\mathbf{q_{i+1}}||} \tag{2.7}$$

This can be iterated with new gyroscope measurements to compute the attitude over time. Note that updating the quaternion $\mathbf{q_i}$ to $\mathbf{q_{i+1}}$ can also be done using quaternion algebra. A quaternion can be made from $\boldsymbol{\sigma}$ (in the same way that it is created from $\boldsymbol{\mu}$ in equation 2.1) and multiplied in the sense of the quaternion. The matrix multiplication from equation 2.6 is mathematically equivalent, but clearer and more straightforward. Refer to [2], [5], or [11] or Appendix A for more details of the quaternion attitude algorithm.

2.3.2 Position Calculations

With the attitude computed, we can transform the acceleration measurements from body coordinates to local geographic coordinates, correct for gravity, integrate to find the velocity, and integrate again to find position. The equations for computing velocity are shown here [5]:

1. The direction cosine matrix (DCM) $C_\imath$ and A_i matrix are derived from the quaternion, $\mathbf{q_i}$, as shown in Appendix A. These transform measurements in body axes to local geographic axes.

2. Convert specific force in body axes, $\mathbf{f_i}$, to inertial force in local geographic coordinates, $\mathbf{f_i^g}$

$$\mathbf{f_i^g} = \mathbf{C_i A_i f_i} \tag{2.8}$$

3. Correct for gravity and integrate to find velocity

$$\mathbf{v_{i+1}} = \mathbf{v_i} + \int_{\mathrm{i}}^{\mathrm{i+1}} (\mathbf{f_i^g} - \mathbf{g})\, \mathbf{dt} \tag{2.9}$$

4. Corrections can be applied for the rotation of the earth, $\mathbf{\Omega_{ie}}$, and the Coriolis effect, $\mathbf{\Omega_{en}}$.

$$\mathbf{v_{i+1}} = \mathbf{v_{i+1}} \left(\mathbf{I} - \mathbf{2} \int_{\mathrm{i}}^{\mathrm{i+1}} \mathbf{\Omega_{ie}} \mathrm{dt} - \int_{\mathrm{i}}^{\mathrm{i+1}} \mathbf{\Omega_{en}} \mathrm{dt} \right) \tag{2.10}$$

We now have a velocity as it changes in time which can simply be integrated to find position over time.

2.4 Summary

In this chapter, we introduced the concepts and equations behind inertial navigation. The system considered here is an unaided inertial navigation system consisting of only a triaxial gyroscope and triaxial accelerometer. Typically, these inertial sensors are not relied on completely and an aided system is used. In an aided system, another sensor such as a GPS or magnetometer is used to assist in determining position or attitude. Meeting the desired performance with no additional sensors proves to be a challenge as described in later chapters.

Chapter 3

Instrumentation

3.1 Introduction

In this chapter, we discuss the individual components comprising the instrumentation of the Smart Rock. We begin with the inertial measurement unit, followed by the pressure sensors, and microcontroller. Calibrations for each of the sensors are presented. For each of the components, the selection process and implementation considerations are discussed.

3.2 Sensors

3.2.1 Inertial Measurement Unit

Overview

The inertial sensor used in the Smart Rock is the ADIS16367 from Analog Devices (see Figure 3.1). This is an inertial system with a triaxial accelerometer and triaxial gyroscope. Each of the sensor outputs are processed digitally through various levels

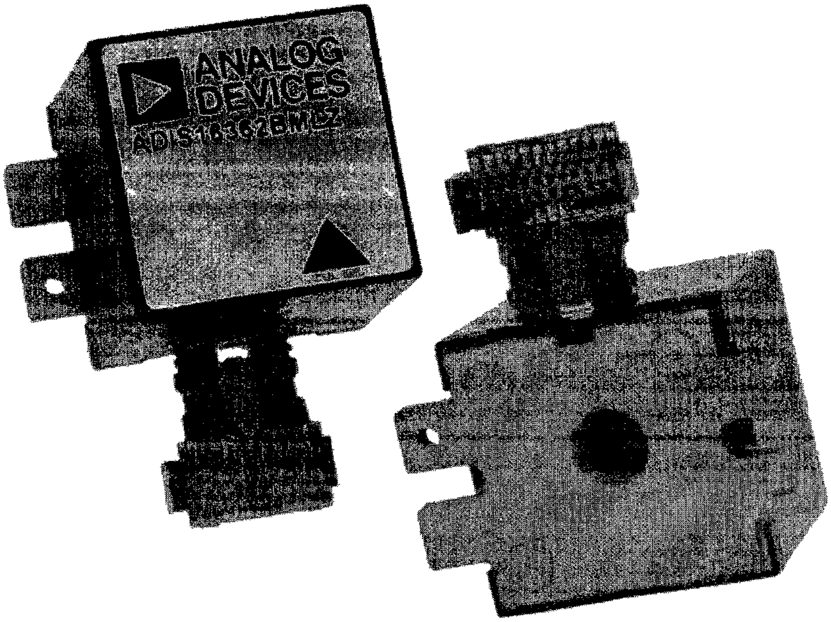


Figure 3.1: ADIS16367 package (different device shown, courtesy of Digikey)

of calibrations and corrections. They come from the factory calibrated for bias (zero shift), sensitivity (ratio of change in output over change in input), alignment of sensor axes, and acceleration-dependent bias on the gyroscopes. It is completely temperature compensated within the device with internal temperature sensors at each axis. The data is also passed through two averaging filters, in the form of a Bartlett window, to act as a low-pass filter, removing any high frequency noise. This filter is adjustable with maximum bandwidth of 330Hz. Finally, the wide dynamic range of each sensor is necessary for the dramatic movements that may occur in the debris flow. The accelerometers span $\pm 18g$, and it is expected that flow accelerations may reach $\pm 10g$; and the gyroscopes can measure up to $\pm 1200^\circ/s$ or 200 rpm, larger than the expected angular velocities in the flow.

Interfacing and Communication

Communication is accomplished via a Serial Peripheral Interface (SPI) Bus. This is a four-wire serial communication protocol in which one device acts as the "Master" and the other acts as the "Slave". The ADIS16367 is configured to act only as an SPI slave and thus needs the system processor to act as the SPI master as indicated in Figure 3.2.

The communication lines of an SPI Bus as shown in Figure 3.2 are as follows:

- SS → CS : Selects the SPI device with which to communicate, this allows several slave devices to communicate with a single master.
- SCLK → SCLK : Clock signal sent from master to slave
- MOSI → DIN : "Master Out Slave In" sends digital signals to "Digital In"
- MISO ← DOUT : "Digital Out" sends digital signals to "Master In Slave Out"
- IRQ ← DIO1 : Additional digital signal from the ADIS16367

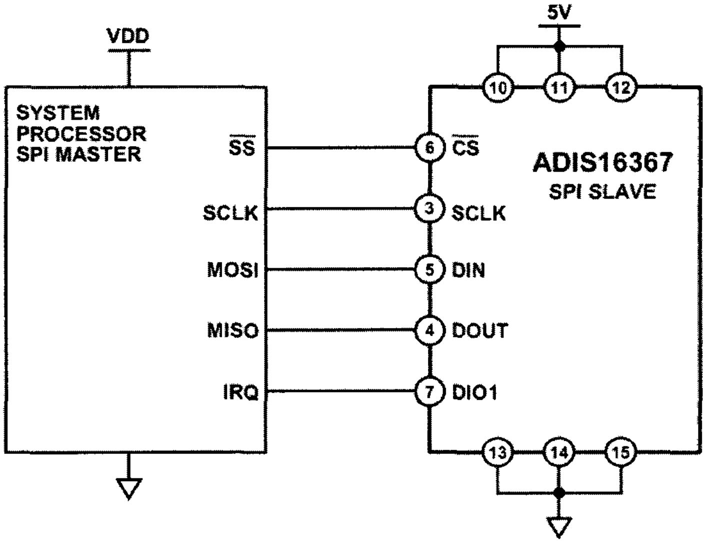


Figure 3.2: ADIS16367 communication diagram [6]

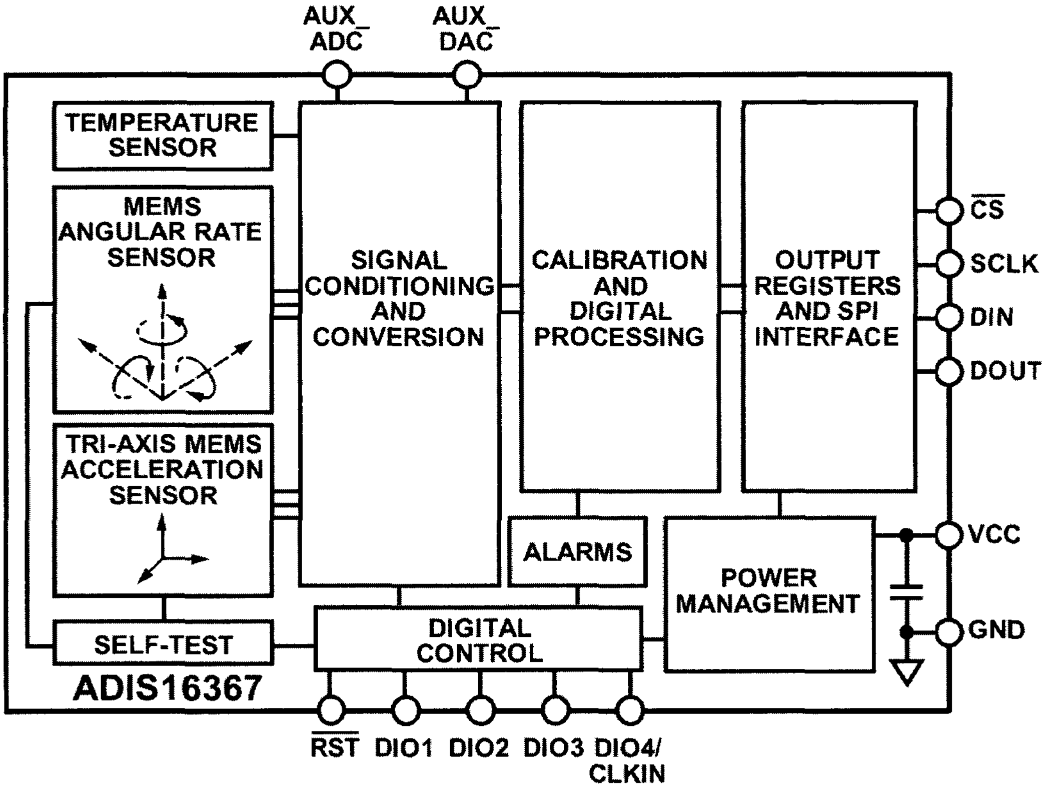


Figure 3.3: ADIS16367 block diagram. SPI bus shown on top right

The first four lines comprise a standard SPI bus. Additionally, DIO1 sends a "data ready" signal to the Interrupt Request pin to let the processor know when the next sample is ready. This is also shown in the ADIS16367 block diagram of Figure 3.3, with the pins for the SPI bus shown on the top right.

The SPI protocol operates in full duplex mode. This means data can be transfered both ways between the master and slave on each clock pulse. This is accomplished via the two data lines, MOSI and MISO, contributing to the high data rates that are possible as compared with other digital communication protocols such as I^2C.

The ADIS16367 uses SPI mode 3. There are four SPI modes which are designated by clock polarity (CPOL) and clock phase (CPHA). In mode 3, CPOL=1 and CPHA=1. This means the polarity is high, or the default clock level is high. Also,

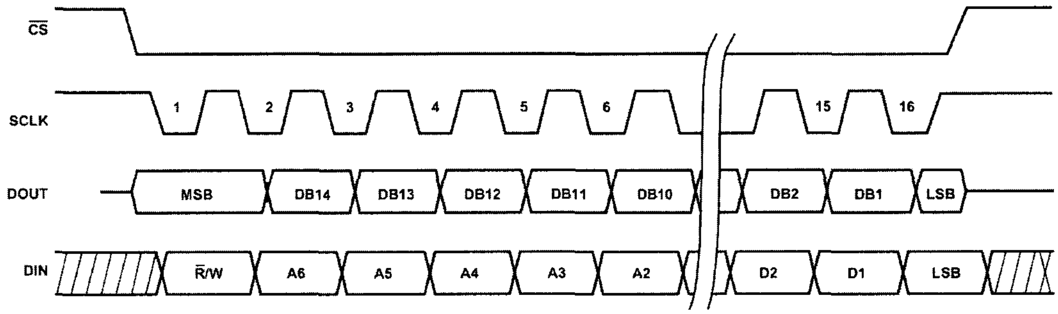


Figure 3.4: SPI timing diagram [6]

the clock phase is such that data is sent on the leading (falling) edge and data is retrieved on the trailing (rising) edge of the clock pulse. This is shown in the SPI timing diagram of Figure 3.4.

The sensor stores data in a series of 16-bit registers. To read from or write to one of these registers, two 8-bit data sequences must be transfered via the SPI bus. The upper byte contains a 0 or 1 in the most significant bit (MSB), to signify a read or write respectively, followed by the 7-bit address of the register of interest. The lower byte is a "don't care" byte for a read, or is the data to be written to the register indicated in the upper byte for a write operation.

For a read operation, once this 16-bit data transfer is completed the subsequent 16-bit sequence contains the data to be read. For different registers, the data may be represented in a variety of binary representations such as 12-bit offset, 12-bit twos complement, or 14-bit twos complement.

3.2.2 Pressure Sensors

The pressure sensors selected for this application are Honeywell's 26PC05SMT as shown in Figure 3.5. In selecting a pressure sensor the following criteria were considered,

- 0-5 psi range

- Small size

- Able to measure fluid pressure

- Low voltage and current draw for battery power

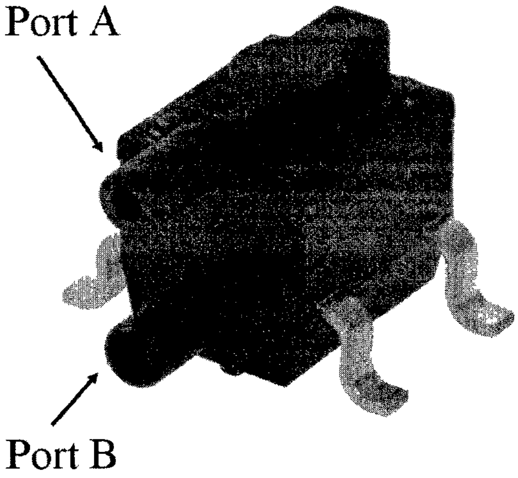


Figure 3.5: 26PC05SMT Pressure Sensor

This MEMS sensor has the required 0-5 psi range for liquid or gas pressure and has a package that measures only $6 \times 7 \times 8$ mm. It has a suggested input voltage of 10 V at an average current draw of only 2 mA. A Wheatstone bridge configuration is used to provide a differential output proportional to the difference in pressure between ports A and B. This interface is described in further detail in Appendix C.

3.2.3 Calibration

Because of the tight calibration specifications on the gyroscopes and accelerometers (see Appendix G), the procedures discussed here are simply to confirm those specifications. For the pressure sensors, a full calibration must be performed as the specifications are not as tight and the direct sensor output is amplified by an instrumentation amplifier.

Gyroscope Calibration

To perform the calibration, a mounting unit was created with a rapid prototyping machine. This unit, shown in figure 3.6, uses two machine screws in the precision alignment holes to keep each axis in line with those of the mounting unit. This unit also has a custom printed circuit board to attach the IMU connector to a small solderless prototyping board. With this, leads can easily be attached for powering and communication. There are three points to mount the unit on a motor. Each of these points aligns one of the sensor axes with the motor's axis of rotation. It also has sides that are perpendicular to each sensor axis to allow gravity to be sensed on a single axis of the accelerometer when it is placed on a flat surface.

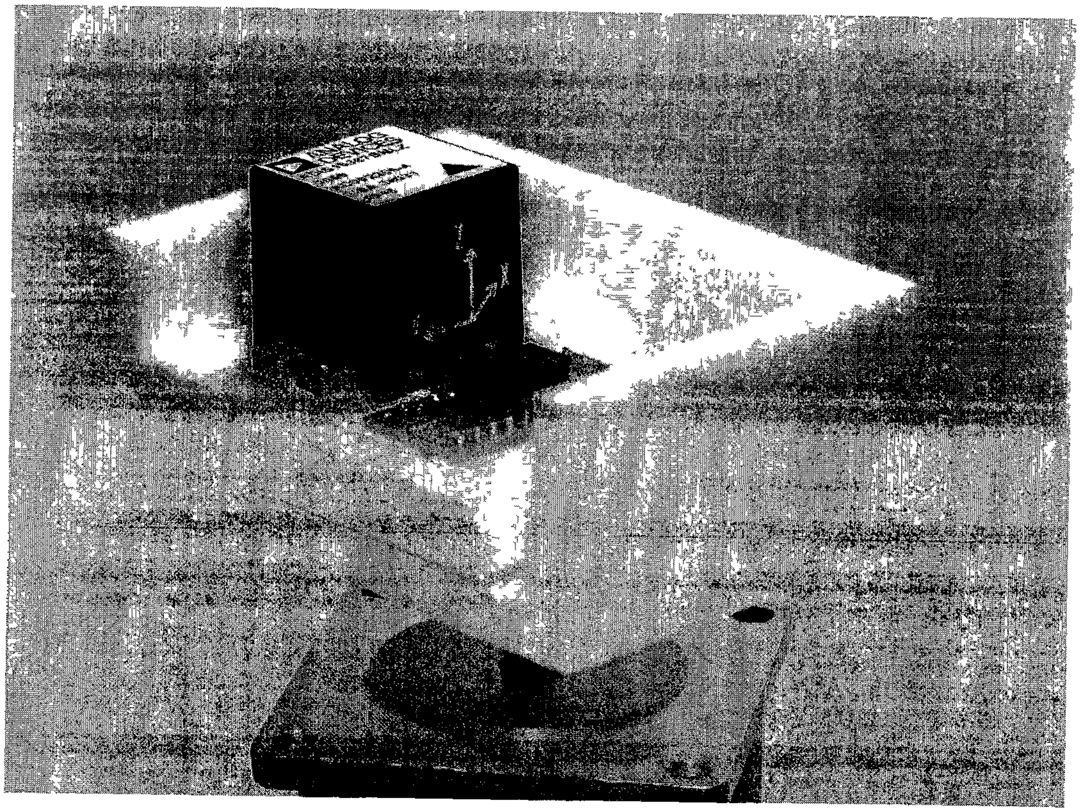


Figure 3.6: IMU mounting unit

To calibrate the gyroscopes, the mounting points are used to align the sensor axes with the axis of rotation of a stepper motor. A stepper motor position and speed can be precisely controlled, making this very useful in calibration. For each of the axes, the unit is rotated at several know rates in each direction. This gives a total of 39 calibration points between $\pm 1710^o/s$, extending 40% beyond the rated range. In Figure 3.7 all of these plots are shown with the best-fit line. As can be see, each of the coefficients of determination is greater than 0.999, thus the sensors show very linear response well beyond the nominal operating range.

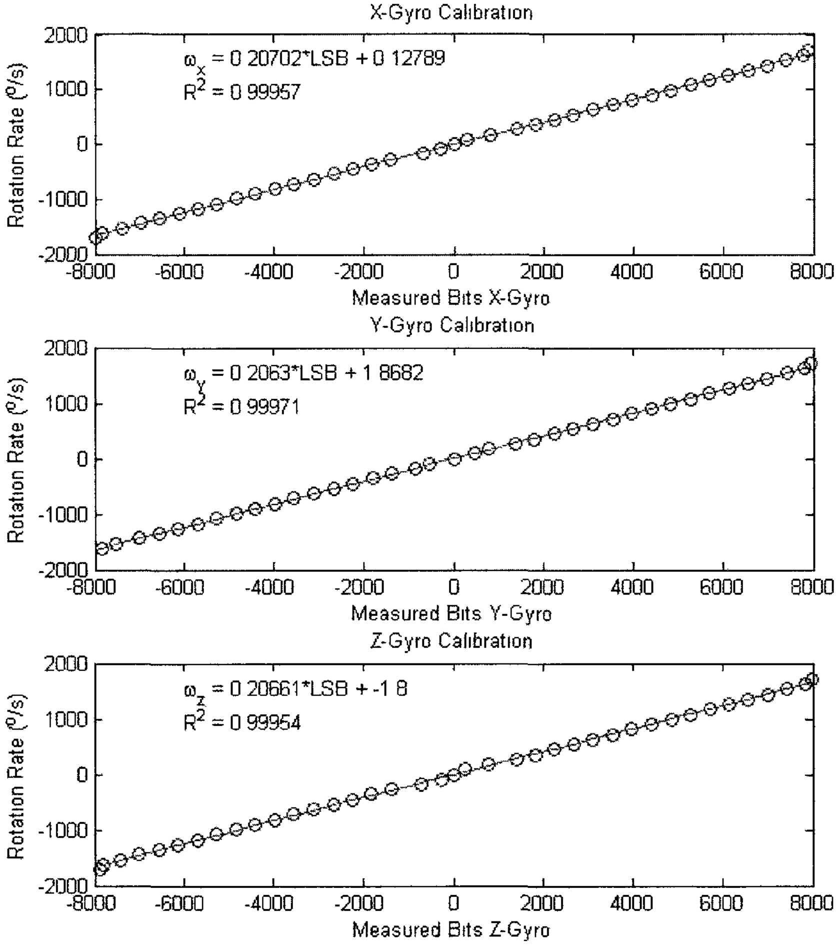


Figure 3.7: ADIS16367 gyroscope calibration plots

The datasheet gives the sensitivity as $0.20 \pm 0.002^o/s/LSB$ (degree per second per least significant bit). The sensitivities found in this calibration are out of the rated range, but are rather precise. It was found that the stepper motor control hardware was driving the motor slower than was commanded. This systematic error comes from the overhead associated with the pulse commands, causing an additional delay. With this small delay, the stepper motor will rotate slower than expected, causing this apparent increased sensitivity. Correcting for this overhead in post-processing brings the sensitivity within the expected range.

This calibration also shows that the maximum bias is $1.8^o/s$, which is within specifications. At a sensitivity of $0.20^o/s/LSB$, this leads is a bias of nine bits. This seems very significant, but can be eliminated in post-processing. The data taken with the Smart Rock will begin and end with a stationary period. This period of no rotation can show us what the biases on the gyroscopes are, which can then be subtracted from the output.

The apparent nonlinearities near 0 are attributed to the noise caused by the stepping action of the stepper motor. When it rotates slowly, the movement becomes choppy, causing erroneous measurements. This calibration is determined to confirm the nominal calibration presented on the datasheet with a sensitivity of $0.20^o/s/LSB$ and zero bias when corrected in post processing.

Accelerometer Calibration

The accelerometer calibration is confirmed by using gravity to apply accelerations to each axis. The method followed here is that described in [7]. When placed with gravity acting along one of the positive sensing axes, one point can be taken, and another when gravity is acting along the negative axis. This gives us a total of six configurations. From the data taken in these six configurations, the following matrices can be created,

$$U_+ = \begin{bmatrix} a_{xx+} & a_{yx+} & a_{zx+} \\ a_{xy+} & a_{yy+} & a_{zy+} \\ a_{xz+} & a_{yz+} & a_{zz+} \end{bmatrix}, U_- = \begin{bmatrix} a_{xx-} & a_{yx-} & a_{zx-} \\ a_{xy-} & a_{yy-} & a_{zy-} \\ a_{xz-} & a_{yz-} & a_{zz-} \end{bmatrix} \tag{3.1}$$

These matrices have elements $a_{\alpha\beta\pm}$ with α as the axis along which gravity is acting, β as the sensing axis, and ± indicating whether gravity is acting along the positive or negative axis. The values of these matrices are found to be,

$$U_+ = \begin{bmatrix} 292.18 & 3.25 & -20.63 \\ 4.31 & 298.09 & -0.13 \\ -7.76 & 0.5649 & 294.54 \end{bmatrix}, U_- = \begin{bmatrix} -305.01 & -13.24 & -0.68 \\ 0.06 & -299.55 & 5.09 \\ -8.33 & 7.71 & -304.51 \end{bmatrix} \tag{3.2}$$

The gain matrix, K, can be found from this data by taking the change in input $(2g)$, and dividing it by the change in the output $(U_+ - U_+)$. This is found to be,

$$K = \begin{bmatrix} 0.003349 & & \\ & 0.003346 & \\ & & 0.003339 \end{bmatrix}. \tag{3.3}$$

Since only the diagonal terms are of interest, the rest are omitted for clarity. These terms indicate how each axis is affected by an input on that axis. These elements shown are within the specifications and only 0.5% off from the nominal sensitivity of $0.00333g/LSB$.

$$K = \begin{bmatrix} & 0.1212 & -0.1003 \\ 0.4705 & & -0.3824 \\ 3.4750 & -0.2796 & \end{bmatrix}. \tag{3.4}$$

The off-diagonal terms indicate apparent cross-sensitivities between different axes.

The most significant of these could be caused by misaligning the sensor axes with the gravitational field by only 5.74^o. This is within the experimental error of this procedure. This also confirms the calibration shown on the datasheet.

The bias matrix, B, can also be formed by averaging the U_+ and U_- matrices element by element. It shows the apparent bias on each axis based on different inputs. We find that,

$$B = \begin{bmatrix} -6.419 & -5.00 & -10.66 \\ 2.19 & -0.73 & 2.48 \\ -8.05 & 4.14 & -4.98 \end{bmatrix} . \tag{3.5}$$

The maximum of these biases is approximately 11 bits, which, using the nominal sensitivity, is equivalent to $37mg$ or 0.23% of full scale. This is well within the specifications given as $50mg$. However, aligning the sensor with gravity can only be done within approximately $\pm 3^o$. It would only take a misalignment of 2.8^o to create an apparent bias of $50mg$ on any axis. Therefore, this misalignment must be considered.

To more precisely find these accelerometer biases, the following procedure is developed. This is based on the industry standard tumble test and static multipoint test as described in [8]. This expanded procedure only requires a flat surface and a right angle reference on the surface with which to align the sensor. These right angle reference axes will be called axes 1 and 2 with axis 3 perpendicular to the plane. Because the surface is not normal to gravity, the axes 1 and 2 are at some unknown angles with respect to the horizontal, θ and ψ respectively. When the triaxial accelerometer is placed on the surface with axes aligned with the reference directions, a reading is taken. From this, three equations can be written. In this example, the x axis is aligned with 1, y is aligned with 2, and z is aligned with 3,

$$a_x = b_x - g\sin(\theta) \tag{3.6}$$

$$a_y = b_y - g\sin(\psi) \tag{3.7}$$

$$a_z = b_z + g\cos(\theta)\cos(\psi) \tag{3.8}$$

Here, the measurements are represented by a_{axis} and the bias that is desired is b_{axis}. These equations can be linearized about a given (θ_0, ψ_0), and the magnitude of g is 1 (measured in units of gs), leaving the following,

$$a_x = b_x - \alpha\theta \tag{3.9}$$

$$a_y = b_y - \beta\psi \tag{3.10}$$

$$a_z = b_z - \gamma \tag{3.11}$$

$$\tag{3.12}$$

with,

$$\alpha = \cos(\psi_0)\sin(\theta_0) \tag{3.13}$$

$$\beta = \cos(\theta_0)\sin(\psi_0) \tag{3.14}$$

$$\gamma = \cos(\psi_0)\sin(\theta_0)\theta_0 + \cos(\theta_0)\sin(\psi_0)\psi_0 + \cos(\theta_0)\cos(\psi_0) \tag{3.15}$$

These equations are now linear with respect to the biases and misalignment angles. With the accelerometer placed in several configurations, many equations can be developed.

$$\begin{bmatrix} a_x \\ a_y \\ a_z \\ \vdots \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & -\alpha & -\beta \\ \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \\ \theta \\ \psi \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ -\gamma \\ \vdots \end{bmatrix} \tag{3.16}$$

or

$$y = Ax + B \tag{3.17}$$

This overdetermined matrix system can be solved using the least-squares method, or the "pseudo-inverse",

$$x = \left(A^T A\right)^{-1} A^T (y - B) \tag{3.18}$$

This finds the accelerometer biases and the angles of inclination of the surface in a simple fashion. It does not require precise alignment with gravity or any other special equipment. It is also insensitive to misalignments with the reference axes because the signal will vary by the cosine of the misalignment, leading to an error of only 1.5% for a misalignment of 3°.

$$b = \begin{bmatrix} 6.45 \\ 0.73 \\ 4.99 \end{bmatrix} LSB. \tag{3.19}$$

Each of these biases for all of the sensors tested are found to be less than 8 bits, or 27 mg. The bias values for each axis of each sensor can then be subtracted to eliminate said bias.

Pressure Sensor Calibration

The pressure sensor's differential output is amplified by the AD620 instrumentation amplifier and a 150Ω gain resistor, then recorded by the microprocessor A/D converter. These are both powered by a single $5VDC$ supply. Because this excitation is lower than what is recommended for each of these devices, the calibration is found to be nonlinear. To perform these tests, a shell with the sensors and amplifiers was placed in a small pressure chamber, and the pressure was varies in small increments. Two tests are performed to ensure reproducibility, and the results are shown here.

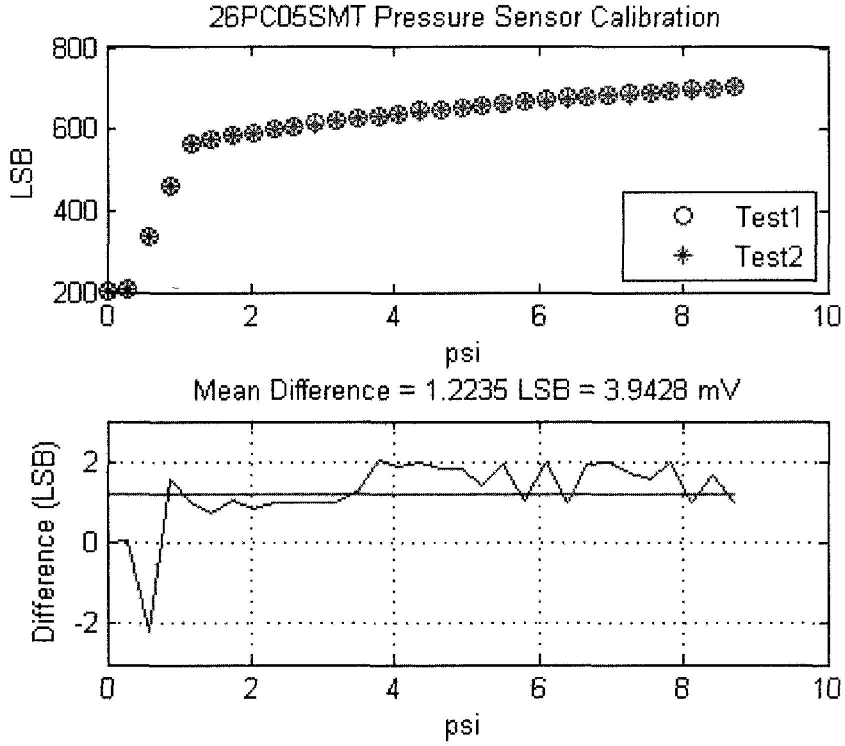


Figure 3.8: 26PC05SMT pressure sensor calibration plots

The difference between the two tests is shown to be within just a few bits in Figure 3.8, showing that this is reproducible. However, the nonlinearity is very clear, and a piecewise fit must be made. The fit equation is shown here:

$$p(psi) = \begin{cases} 5.7664 \times 10^{-2} LSB - 11.7636 & \text{for } LSB < 209 \\ 2.4080 \times 10^{-3} LSB - 0.2224 & \text{for } 209 \leq LSB \leq 565 \\ 2.1309 \times 10^{-4} LSB^2 - 0.2168 LSB + 55.6052 & \text{for } LSB > 565 \end{cases} \tag{3.20}$$

The piecewise fit here, with $R^2 = 0.9990$, gives a very high sensitivity in the low pressure range, which is the greatest region of concern. The calibration has a reduced sensitivity at higher pressures, but does extend to nearly double the rated pressure range of $5psi$. The need for this piecewise fit can be attributed to the way in which the sensor and amplifier are powered. The sensor is powered at $5VDC$ rather than the suggested $+10VDC$, and the AD620 amplifier is powered in the same manner as opposed to the suggested $\pm 10VDC$. Had the suggested power supplies been available in the Smart Rock, this could have been made linear.

3.3 microSD Card

The data that is read from these sensors is written to a text file on a μSD card. A card socket breakout board is provided by SparkFun Electronics. This provides gigabytes of storage in a very small form factor. A standard 4 GB card can store data for 250 hours. Refer to Appendix C for wiring diagram.

3.4 Microcontroller

The logging capabilities are driven by the Arduino Pro Mini 328 based on the Atmel ATmega328 microprocessor (see Figure 3.9). This is a 3.3V 8 MHz microcontroller unit that includes analog inputs for the pressure sensors as well as an SPI bus for digital communication with the inertial measurement unit and the microSD card socket At just 0 7″ × 1 3″ its small size makes it ideal for this space constrained application. Refer to Appendix C for a wiring diagram and Appendix G for additional detail.

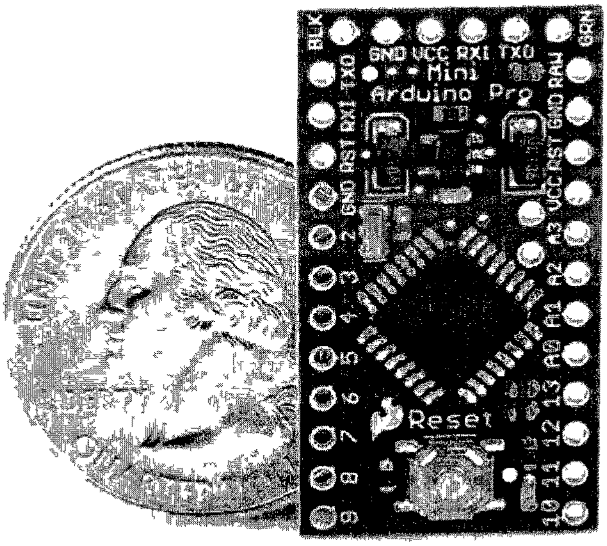

BLK
GND VCC RXI TXO
Mini
Arduino Pro
GRN
TXO RXI RST GND 2 3 4 5 6 7 8 9
RAW GND RST VCC A3 A2 A1 A0 13 12 11 10
Reset

Figure 3.9: Arduino Pro Mini 328

3.5 Firmware

The firmware running on this Arduino microcontroller was developed specifically for this project. Several issues regarding communication between devices are addressed including the μSD card, IMU, and pressure sensors. In addition, the methods for storing the data are discussed as well as other issues and features of the firmware. Interested readers can consult Appendix B for further information on the program.

3.5.1 Communications

First, the microcontroller communicates with the μSD card through a standard SPI interface. The Arduino has one hardware SPI port which is used for this communication line. In order to implement the specific communication protocols required by the μSD card, Bill Greiman's SdFat library (licensed under the GNU General Public License v3) was used (`http://code.google.com/p/sdfatlib/`). This library includes all basic functionality for high level commands to create files and write blocks of data to the card as well as many low level commands. The "raw write" commands were used to write a simple string of bytes to a file as quickly as possible.

As previously discussed, the microcontroller must also communicate with the IMU which uses the same SPI protocol. SPI allows for one "master" (the Arduino) and multiple "slaves" (the IMU or μSD card) which can be selected by the master via a simple high or low digital signal. This would allow the card and IMU to share the single available SPI port. However, these two devices require different SPI modes. The SPI protocol can be implemented in any of four different flavors. These differ by the polarity of the clock signal (whether it begins high or low) and the phase of the clock signal (whether it shifts data out first or latches incoming data first) and cannot be mixed as would be required for this application. Thus, the hardware SPI port is dedicated to the μSD card and a software implementation of an SPI port is created on four other general purpose input/output (GPIO) lines. This allows for the SPI protocol to be implemented alongside the hardware bus, but at a lower rate as the software bus is less efficient.

Finally, the Arduino must communicate with the pressure sensors. These sensors have a differential voltage output on two analog lines. These are amplified in the AD620 instrumentation amplifier which changes the differential signal to a single ended signal. These analog signals are read by the Arduino on-board A/D converter.

3.5.2 Initialization and Configuration

To begin, a configuration file (`CONFIG.TXT`) is searched for on the μSD card. This file shall contain a time for which to delay before recording begins, the desired sampling frequency, and the duration of sampling. Once this initial data is read, a new file is created on the μSD card to record data.

It is here that the IMU is also configured. This sets the gyroscope measurement range to $\pm 1200^o/s$, sets the parameters for the internal Bartlett filter to reduce the bandwidth to approximately $50Hz$, initializes a correction on the gyroscope bias based on linear acceleration, sets the DIO1 line as a data ready indicator as mentioned previously, and commands a self-test to determine if there are any problems with the sensor.

3.5.3 Data Acquisition and Storage

Using the communication protocols discussed above, the sensor readings are taken from the accelerometers, gyroscopes, and pressure sensors. This data is stored temporarily in SRAM (static random access memory). From the two byte word representation, a string of hexadecimal characters is created such that it will be readable in the destination text file. Every new sample is converted to a hexadecimal string and concatenated onto the end of this character buffer. The SdFat library writes blocks of 512 bytes. Once the buffer is filled to this level a card write sequence is initiated. While writing, a new buffer begins with the next set of incoming data. If there is a communication failure (i.e. power is lost or the μSD card becomes dislodged), the samples that had been written will be uncorrupted and readable, but no further data can be taken.

3.5.4 Timing

The IMU and pressure sensors are polled for a reading at an interval specified by the desired sampling frequency. This sampling is driven by an interrupt on the Arduino Timer1. This is driven by the TimerOne library provided by Jesse Tane and issued under Creative Commons Attribution 3.0 United States License (`http://www.arduino.cc/playground/Code/Timer1`). It allows an interrupt to trigger an interrupt service routine (ISR) at a given interval. With this configuration, writing to the μSD card is the primary operation while acquiring new data is interrupt driven. In the ISR, the sensors are polled, the data is converted to the correct format in the hexadecimal string buffer, and a counter is incremented. The counter is used to determine when it is time to write the next block of data to the text file.

3.6 Packaging

With the components selected, packaging must be designed to allow them to function together while:

1. Keeping the entire Rock as small as possible

2. Matching the density and mass distribution to that of real rock

3. Providing an appropriate interface for the pressure sensors

Adhering to these criteria ensure that the dynamics of the Rock will match that of true rock. This allows for comparisons with models for the movement of courser debris.

3.6.1 Electrical Packaging - PCB Design

The electrical components are held together on a printed circuit board (PCB), see Figure 3.10. This board is a 7 *cm* diameter circle that fits into a shell close to the shape of a real rock. The board has surface mount pin headers for the Arduino; the appropriate surface mount header and mounting through holes for the IMU; holes for a power connection, LEDs, and μSD card socket; as well as pads for the instrumentation amplifiers, pressure sensors, capacitors and resistors. More details, including a schematic and board layout, can be seen in Appendix C.

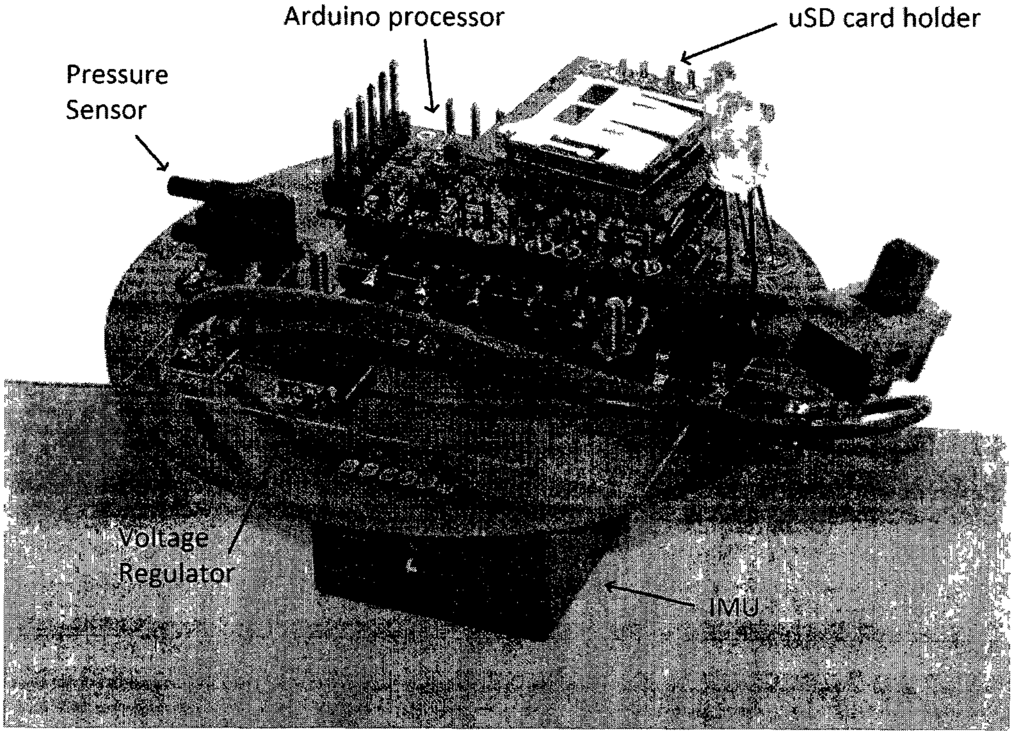


Figure 3.10: Populated PCB

3.6.2 Mechanical Packaging - Shell Design

With the electrical components on the circular PCB, a shell is designed to contain them. This shell, and the interface in particular, are designed to meet the most stringent ingress protection rating: IP68. This requires that the shell completely protects the contents from dust as well as from water during extended immersion in water at over 1 *m* depth. This shell has two halves that are joined with threads. An o-ring is used to seal the interface.

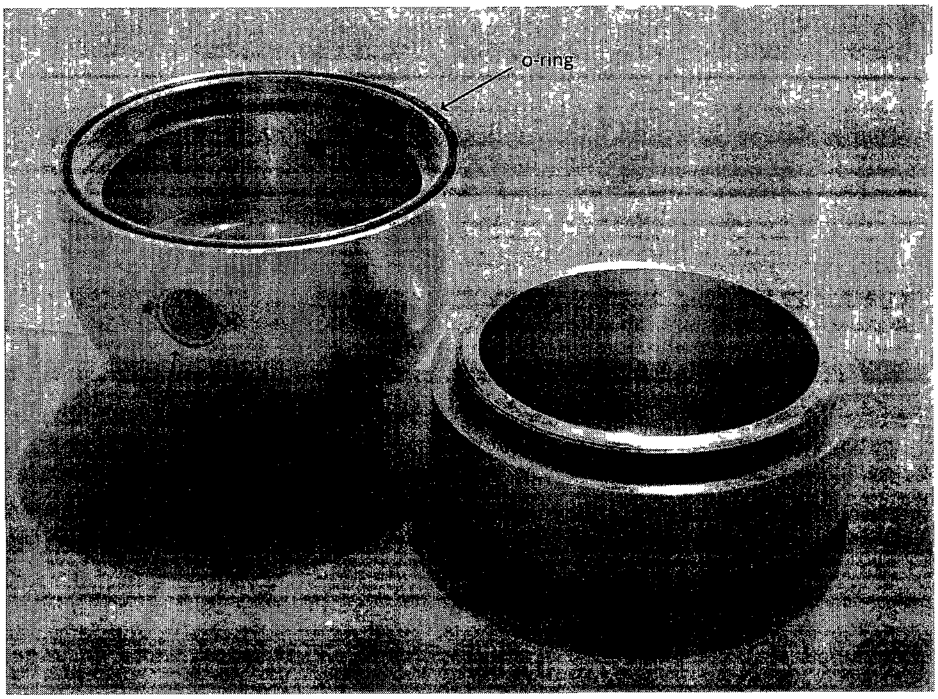


Figure 3.11: Smart Rock shell

This shell is made of Aluminum 6061-T6 with thickness of at least 3 *mm* to protect it from impact with other rocks and the concrete flume itself This shell has a known volume and density, as well as a known cavity volume. From this, the required mass to be placed in the cavity is found in order to match the overall density of real rock, $\approx 2\ 7\, {}^{g}/_{cm^3}$. Adding a mass of 250 g ensures the correct density.

To provide an interface for the pressure sensors, holes on opposite sides of the rock are made. These holes allow a small tube to connect to the pressure sensor inside the rock. These holes also have a countersink, allowing a brass sinter piece with filtration rating of 40 microns to be inserted and held in place with two set screws

This sinter piece will protect the pressure sensor from any material with which it comes into contact, and provides a conduit between the sensor and the surrounding groundwater. Further details on the shell dimensions can be seen in Appendix D.

3.7 Summary

In this chapter, a detailed discussion of the instrumentation in the Smart Rock was given. This included all applicable features of the sensors and Arduino microcontroller. In addition, the calibration procedures are described for the gyroscopes, accelerometers, and pressure sensors. The datasheet calibrations are confirmed for the inertial sensors and a piecewise calibration curve is created for the amplified pressure sensor output. Also, the firmware running on the microcontroller is presented with details on the communication protocols, timing, and data formatting. Finally, the packaging is discussed in terms of electrical interfacing (PCB), and mechanical considerations (shell).

Chapter 4

Noise Characterization and Allan Variance

4.1 Introduction

In this chapter, noise types and their sources in electronic devices are investigated. Their effects related to gyroscopes and accelerometers are then discussed.

Next, the Allan Variance test procedure that is used to quantify noise sources is presented including the historical context and the computation procedure. It is shown that noise sources can be determined from the plot of the Allan Deviation (ADEV), and its application to gyroscopes and accelerometers is discussed. Finally, the testing procedure and results for the ADIS16367 IMU are presented. The resulting plots are shown with full analysis in the following chapter.

4.2 Noise Types

4.2.1 White Noise

When signal noise is discussed, one of the first types that comes to mind is so-called white noise. By definition, white noise is a random signal with a flat power spectral density. This means it has a completely flat spectrum from DC to infinite frequency. If this is true, the constant power integrated over the frequency means the signal has infinite energy, which is clearly impossible. In reality, this flat power spectrum is only an approximation. It appears nearly flat over a very broad range, but drops off at some upper boundary, often in the far-infrared region.

In addition to frequency content, the magnitude distribution is also needed to characterize the noise. This is typically taken to be a Gaussian distribution with a given standard deviation, σ_w, and zero mean.

In electronics, white noise is caused by thermal noise and shot noise. Any dissipative process has some thermal noise caused by thermal fluctuations. For example the spectral density of thermal noise over a resistor is,

$$S_n(f) = 4k_B T R \tag{4.1}$$

where k_B is the Boltzmann's constant, T is temperature and R is the nominal value of the resistor. Clearly, this is independent of frequency, and thus is characterized as white noise.

Shot noise is associated with a DC current flowing through a semiconductor. The discrete electrons must cross potential barriers, such as those in p-n junctions, causing discrete changes in current fluctuating about the DC mean. The spectral density of shot noise is,

$$S_n(f) = 2q_e I_{DC} \tag{4.2}$$

where q_e is the electron charge and I_{DC} is the DC current. Again, this spectrum is independent of frequency and thus falls under the category of white noise [9].

Applied to inertial sensors, white noise introduces Angle Random Walk (ARW) in gyroscopes and Velocity Random Walk (VRW) in accelerometers. When integrating a noisy series of zero mean data, a "random walk" phenomenon occurs, with steps of varying magnitudes in random directions. This causes a drift in the signal whose standard deviation grows in time.

4.2.2 Colored Noise

In addition to white, noise can come in a variety of colors. These other varieties are also categorized by their frequency spectrum. White noise is independent of frequency, and thus the power spectrum is proportional to f^α, where $\alpha = 0$. Other colored noises are characterized by other exponents. The color designations are not standardized among all disciplines, but the designations used in this paper are summarized in Table 4.1.

Table 4.1: Noise colors, $S(f) \propto f^\alpha$ [10]

Noise Color	Proportionality Exponent, α
White	0
Pink	-1
Brown	-2
Black	<-2
Blue	1
Violet	2

Of these other noise types, the only one that will be considered is pink noise. According to Woodman [11], pink and white noise are the primary types associated with MEMS inertial sensors. This is both because many of the other colored noises decay faster than pink noise and because of their sources in electronics.

Pink noise is often called "$1/f$" noise due to its power spectrum, or "flicker noise" because "it can make the light-emitting filament in an old-fashioned vacuum tube flicker like a candle in the wind" [10]. It arises because of the capture and release of electrons in certain localized "trap" states in the semiconductor [10]. There is a distribution of time for this capture and release process to take place which depends logarithmically on the binding energy. It is found that the log of the distribution of this binding energy forms a $1/f$ spectrum, giving the same spectrum to the movement of the electrons. The spectrum is given as,

$$S_n(f) = K\frac{(I_{DC})^a}{f} \tag{4.3}$$

where I_{DC} is the DC current, and K and a are device dependent constants. Because this is a low-frequency phenomenon, in terms of sensor output, it is often regarded as a change in the bias over time. Thus, this bias instability must also be considered.

In a device where both pink and white noise are present, at some frequency the pink (flicker) noise will equal the white (shot and thermal) noise. This is known as the "flicker-noise corner frequency". Above this frequency, white noise dominates and the overall spectrum is generally flat, while below this frequency the spectrum obeys the $1/f$ power law.

Now that the different types of noise in which we are interested are introduced, a method to characterize and quantify them is found in the Allan Variance.

4.3 Overview of Noise Effects in Inertial Navigation

In addition to the noise types discussed in the previous section, an uncorrected constant bias will also be considered. In table 4.2, we see a summary of how each type of error that is being considered affects attitude and position calculations. This only considers a single error type (constant bias, white noise, or pink noise) on a single sensor (gyroscope or accelerometer).

Table 4.2: Summary of Sensor Error Effects [11]

Error Type	Description	Gyroscope Effects	Accelerometer Effects
Constant Bias	A constant bias ϵ	Steadily growing error $\theta(t) = \epsilon t$	Quadratically growing error $s(t) = \epsilon\frac{t^2}{2}$
White Noise	Gaussian noise with standard deviation σ	Angle Random Walk $\sigma_\theta(t) = \sigma\sqrt{t/f}$	Velocity Random Walk $\sigma_s(t) = \sigma\sqrt{\frac{t^3}{3f}}$
Pink Noise	Bias instability modeled as bias random walk	2^{nd} Order Random Walk $\sigma_\theta(t) \propto t^{3/2}$	3^{rd} Order Random Walk $\sigma_s(t) \propto t^{5/2}$

As seen here, errors in the accelerometer signals grow faster in time than the same error on the gyroscope signals. This is because acceleration data must be integrated twice to obtain position, whereas angular rate is integrated only once. From this analysis, it may appear that noise on the accelerometer is the most significant. However, noise in the gyroscopes cause errors in attitude. When gravity is subtracted from the sensed body forces, the correction is then misaligned and adds an apparent acceleration in a different direction. Thus, both accelerometer noise and gyroscope noise are very important. This is discussed further in Chapters 5.

4.4 Allan Variance

It has now been shown that there are various sources of noise in MEMS sensors and these different noise types have different effects on the navigation calculations. The Allan Variance is a method to quantify the noise sources such that the theoretical errors in calculated position can be estimated.

4.4.1 Background

During his time at the National Bureau of Standards (now the National Institute of Standards and Technology, NIST), Dr. David Allan worked with characterizing the stability and accuracy of oscillators for time keeping. He introduced the two-sample variance, now known as the Allan Variance, as a means of doing so. The Allan Variance (AVAR) is a time domain analysis technique that can be used to determine the underlying processes that bring about the noise in data. It has since become a preferred method by IEEE [12]. While it was originally developed to quantify the error statistics of a cesium beam frequency standard employed by the U.S. Frequency Standards from the 1960s, it can be generalized to analyze the noise in any measurement instrument [13].

4.4.2 Computation Procedure

The computation of the Allan Variance is straightforward. Consider a time series of data from a gyroscope, x_i. Divide this time series into clusters of length τ, the averaging time, and take the mean of each of these cluster. This results in a series of N cluster averages, y_i. Square the difference between successive y values and sum them all. Finally, divide by twice the number of degrees of freedom, $N - 1$, less one [14]. This can be calculated for different averaging times τ, yielding the Allan Variance, σ_a^2, as a function of τ. This is represented in the following equation:

$$\sigma_A^2(\tau) = \frac{1}{2(N-1)} \sum_i (y(\tau)_{i+1} - y(\tau)_i)^2. \tag{4.4}$$

The Allan Deviation, ADEV, is the square root of the Allan Variance, and is used in a similar fashion as the standard deviation. The value of the Allan Deviation, σ_A, can then be calculated for a series of τ values, and plotted on a log-log scale.

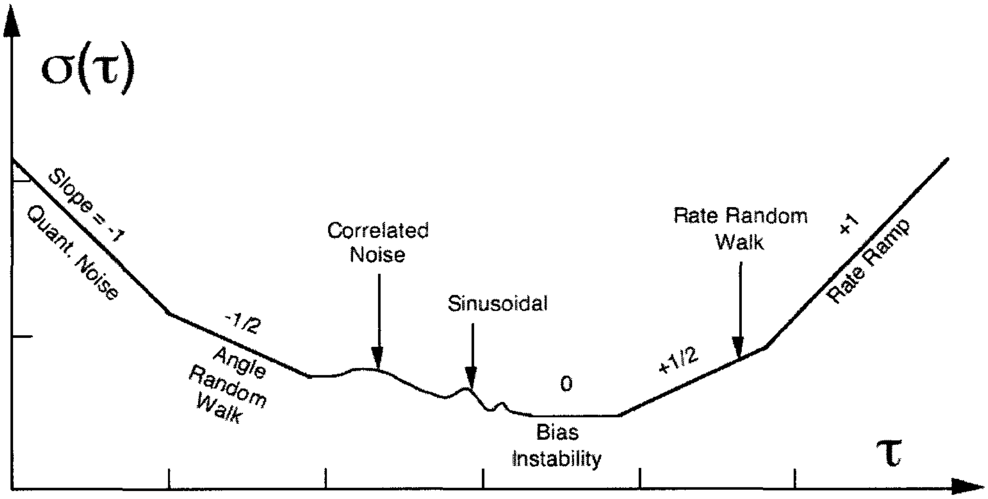


Figure 4.1: Example of Allan Variance slopes. [15]

4.4.3 Noise Source Determination

The Allan Variance computation procedure will produce a plot similar to the one shown in Figure 4.1. Beginning at the lower averaging times, the ADEV decreases with increasing τ. This is analogous to taking a mean over a longer time and having a lower standard deviation because of the increased number of data points. This portion of the graph is referred to angle random walk (ARW). This trend changes at a point where the ADEV begins to flatten and increase. This is due to rate random walk (RRW), a separate and completely independent noise source in the data [14].

Herein lies the beauty of the Allan Variance procedure. Based on the slopes of σ_A vs. τ on a log-log scale, the different error types can be identified and quantified.

When presented with an Allan Deviation plot such as the one in Figure 4.1, these characteristics are very evident. Each error type corresponds to a given slope on the graph, and the deviation due to each can be calculated as shown in Table 4.3. The values for N, B, etc., are found by finding a portion of the plot with the required slope. The σ value is taken off the plot at a given τ in this region, and the value can be calculated using the given equations. Note that the error types listed here are in reference to those of gyroscopes, but each of these has an analogous errors for accelerometers.

Table 4.3: Allan Deviation noise sources [13]

Error Type	Symbol	Allan Deviation	Slope of ADEV
Quantization	Q	$\sigma_{quant} = \sqrt{3}Q/\tau$	-1
Angle Random Walk	N	$\sigma_{white} = N/\sqrt{\tau}$	-1/2
Bias Instability	B	$\sigma_{bias} = B/0.6648$	0
Sinusoidal	ω_0	$\sigma_s = \omega_0\,(sin^2(\pi f_0 \tau)/\pi f_0 \tau)$	Sine Curve
Rate Random Walk	K	$\sigma_{rw} = K\sqrt{\tau/3}$	+1/2
Rate Ramp	R	$\sigma_{ramp} = R\tau/\sqrt{2}$	+1

4.5 ADIS16367 Allan Variance Testing Procedure

To apply the Allan Variance technique to the ADIS16367, two data sets are recorded with the sensor held stationary, once for 45 minutes at 200 Hz and once for 2 hours at 100 Hz. The Allan Deviation algorithm is performed on every data string, resulting in six accelerometer and six gyroscope ADEV plots. A sampling of these can be seen in Figure 4.2. As discussed previously, the primary noise types of concern are white noise and pink or flicker noise, causing angle random walk (ARW) and bias instability respectively, both of which are clearly identified in the ADEV plots. The plots that are generated all follow the trend as expected with a slope of $-{}^{1}/_{2}$ representing white noise and ARW followed by a flat portion representing pink noise and bias instability.

It can be noted that an upward slope is present in many of these plots, indicating rate random walk. However, this occurs at averaging times much greater than the duration of the runs and thus will have minimal effects on the system

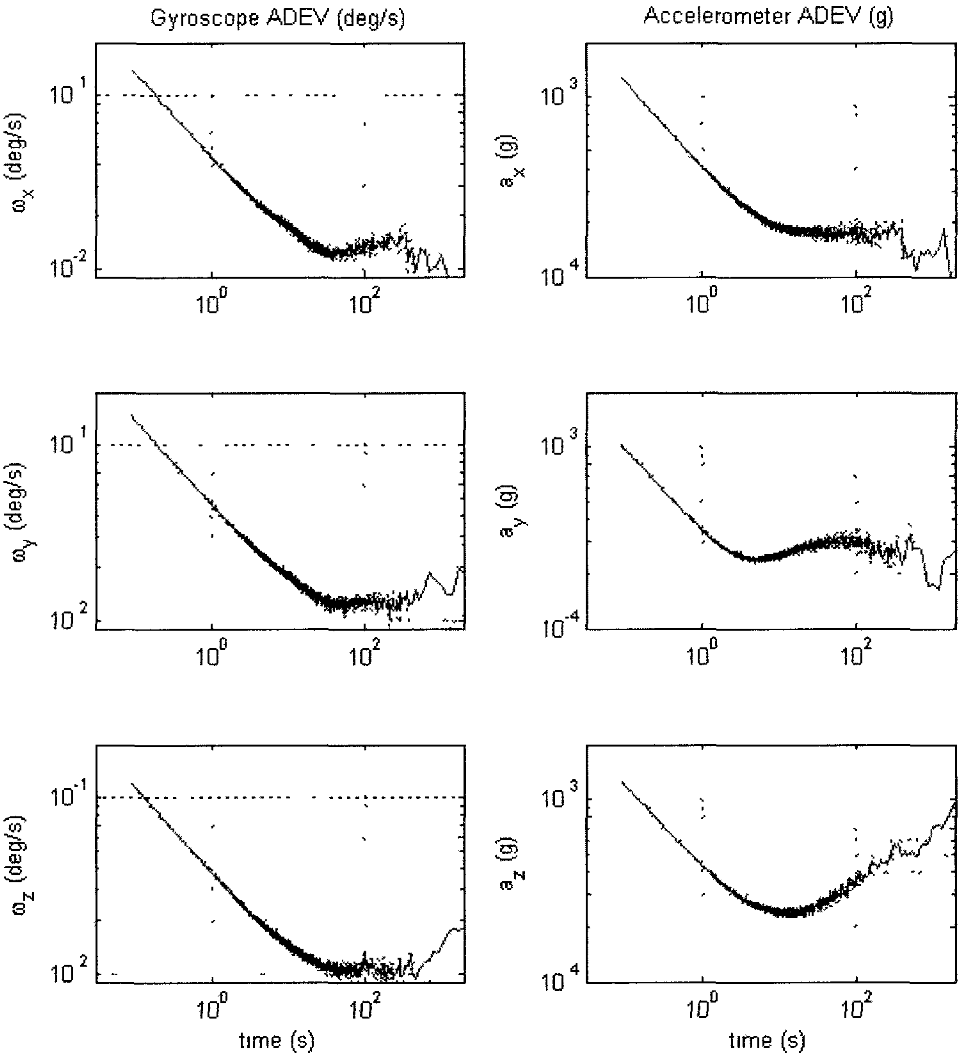


Figure 4.2: Allan Deviation computed from recorded data sets.

4.6 Noise Level Comparisons

In addition to the data collected by our test setup, Mark Looney, an iSensor Applications Engineer at Analog Devices, provided us with sample data from the ADIS16367 taken on their test setup This, along with the noise specifications on the datasheet can be compared with our own data sets and calculations. Plots for the Allan Deviation created by our own tests are shown in Figure 4.2. Those created by the data provided by Analog Devices are shown in Figure 4.3 and the plots from the datasheet in Figure 4.4.

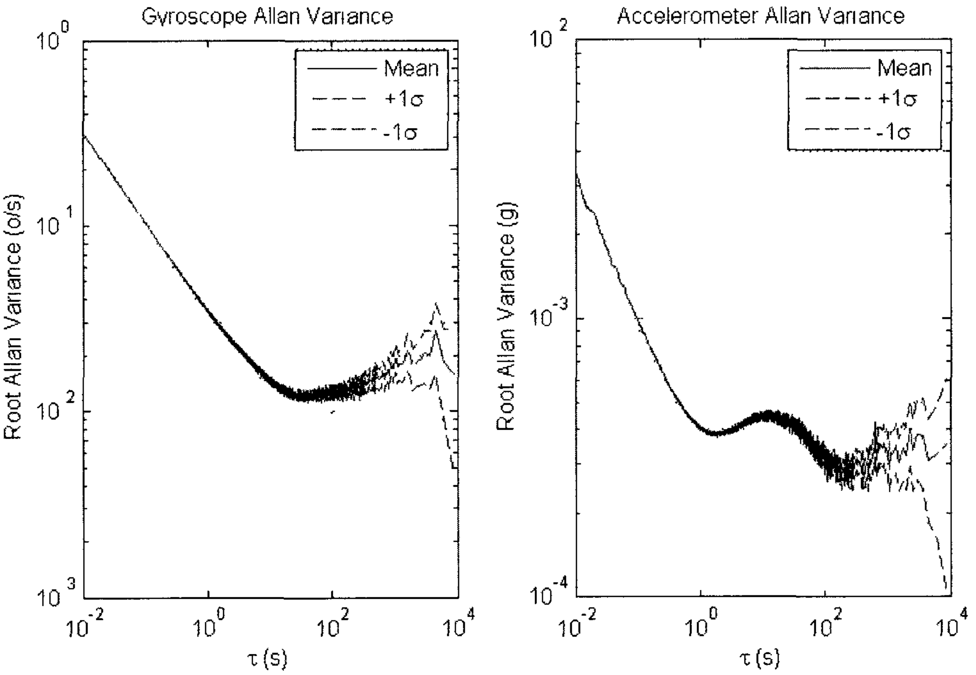


Figure 4.3: AVAR plot from ADIS16367 raw data recorded by Analog Devices

These seem to also have the same shape, beginning with the slope of $-1/2$ indicating white noise and flat area indicating pink noise However, the exact values on the plots seem to be very different This is because the data was recorded at different sampling rates To find a directly comparable measurement, we must first obtain

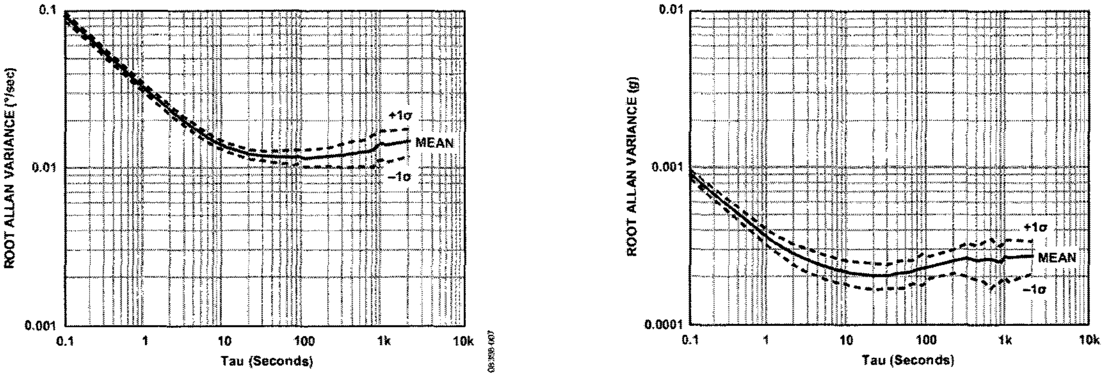


Figure 4.4: AVAR plot from ADIS16367 datasheet [6]

the N and B (see Table 4.3) values off the plots and convert them to the standard deviation of the underlying noise sequence.

For Angle Random Walk (ARW) of gyroscopes and Velocity Random Walk (VRW) of accelerometers, the conversion to find the white noise standard deviation is

$$\sigma_{white} = N\sqrt{BW} \tag{4.5}$$

where N is the ARW/VRW value with units of $deg/\sqrt{s}$ or $g\sqrt{s}$, BW is the bandwidth in Hz and the white noise standard deviation value is in units of deg/s or g [16]. Similarly, for bias instability,

$$\sigma_{pink} = \frac{B}{\sqrt{t \times BW}} \tag{4.6}$$

where B is the bias instability coefficient from the ADEV plot in units of deg/s or g, t is the time at which the ADEV reaches its minimum, BW is the bandwidth in Hz, and the pink noise standard deviation value is in units of deg/s or g.

Summarized in tables 4.4 and 4.5 are these parameters for the gyroscopes. Also, in tables 4.6 and 4.7 are these parameters for the accelerometers. Each table provides values from the datasheet, from Analog Devices' sample data, and from our own two experiments indicated by the subscripts 1 and 2 (i.e. $\omega_{x,1}$ and $\omega_{x,2}$).

Table 4.4: Gyroscope white noise parameter summary

	ARW ($^\circ/\sqrt{s}$)	$\sigma_{white}(^\circ/s)$
Datasheet	0.033	0.954
Analog Devices Data	0.034	0.967
$\omega_{x,1}$	0.151	2.142
$\omega_{y,1}$	0.158	2.233
$\omega_{z,1}$	0.131	1.855
$\omega_{x,2}$	0.044	0.435
$\omega_{y,2}$	0.046	0.459
$\omega_{z,2}$	0.037	0.373

Table 4.5: Gyroscope pink noise parameter summary

	Bias Stability ($^\circ/s$)	Stability Time (s)	$\sigma_{pink}(^\circ/hr)$
Datasheet	0.0130	100.00	0.164
Analog Devices Data	0.0183	33.03	0.400
$\omega_{x,1}$	0.0762	24.00	3.960
$\omega_{y,1}$	0.0770	38.36	3.165
$\omega_{z,1}$	0.0738	22.39	3.967
$\omega_{x,2}$	0.0187	37.91	1.094
$\omega_{y,2}$	0.0189	38.79	1.094
$\omega_{z,2}$	0.0162	45.54	0.862

Table 4.6: Accelerometer white noise parameter summary

	VRW ($mg\sqrt{s}$)	$\sigma_{white}(mg)$
Datasheet	0.340	9.728
Analog Devices Data	0.331	9.480
$a_{x,1}$	0.200	2.830
$a_{y,1}$	0.132	1.860
$a_{z,1}$	0.369	5.214
$a_{x,2}$	0.405	4.053
$a_{y,2}$	0.347	3.471
$a_{z,2}$	0.431	4.314

The final comparisons can be made for the rightmost columns in each of these tables: the standard deviation of the underlying noise signals. While there is some

Table 4.7: Accelerometer pink noise parameter summary

	Bias Stability (mg)	Stability Time (s)	$\sigma_{pink}(\mu g)$
Datasheet	0.200	11.50	2.06
Analog Devices Data	0.579	1.726	15.39
$a_{x,1}$	0.130	30.97	1.65
$a_{y,1}$	0.135	16.34	2.36
$a_{z,1}$	0.347	10.24	7.67
$a_{x,2}$	0.272	15.95	6.80
$a_{y,2}$	0.363	4.34	17.42
$a_{z,2}$	0.361	12.41	10.25

variation, all of the values in each of these rightmost columns are of the same order of magnitude. This now gives us a mean and a range for each of these noise levels.

4.7 Final Noise Model

In order to use this data, a final noise model must be created with the parameters shown in tables 4.4, 4.5, 4.6 and 4.7. We now know the magnitudes of the white and pink noise on the signals and these can be simulated easily with the average σ value from the appropriate table.

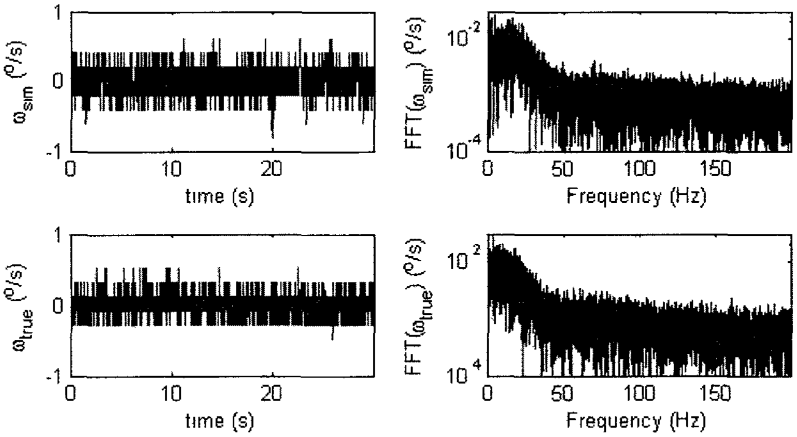


Figure 4.5: Model of IMU noise in time and frequency domain

When the sensor is being used in the Smart Rock, it will employ an internal 4-tap Bartlett filter. This cuts the bandwidth down to approximately $50Hz$. Additionally, the data is quantized in magnitude, as the sensor has a digital output. Adding the Bartlett filter and magnitude quantization yields the following noise model. In Figure 4.5, we see the simulated gyro output (top) and the actual sensor output (bottom) compared in the time (left) and frequency (right) domains. As can be seen, this simulated data using the information from the Allan Variance technique and processing it in the same manner as the actual IMU produces nearly identical data strings.

4.8 Summary

In this chapter the general types of noise were described and their sources in electronics were discussed. A method to characterize a variety of noise sources has been introduced in the Allan Variance. This time domain technique takes a long series of data and calculates variances based on averaging times. The slopes of this Allan Variance on a log-log plot distinguish between the noise sources.

The Allan Variance was calculated for the inertial sensors being used in the Smart Rock, Analog Device's ADIS16367. It is seen that the plots follow a general trend as expected indicating ARW and bias instability. From these, the underlying noise standard deviations are determined.

Finally, a noise model is made which includes the white noise, pink noise, the IMU's internal digital filter, as well as quantization effects.

Chapter 5

Simulation and Testing

5.1 Introduction

With the data that has been gathered, a "virtual IMU" can be made with simulated data and this can be compared to the true inertial measurement unit (IMU). The virtual IMU will allow us to vary parameters to distinguish the important effects on the signals. Conditions where the IMU is stationary and undergoing various motions are considered and compared. Additionally, testing procedures for the pressure sensors are discussed.

5.2 Stationary Simulation

Now that these noise parameters have been determined and a satisfactory noise model has been created, signals can be created with these levels of noise to see their effect on the navigation process. A stationary virtual IMU will first be considered with white and pink noise added to the clean signals. Once one of these signals is run through the navigation algorithm, the Euclidean drift can be computed simply by taking the norm of the position vector. This gives us a single performance measure to consider.

First, using the mean values of the noise levels from tables 4.4, 4.5, 4.6 and 4.7, a simulation is run. Because of the stochastic nature of these noisy signals, several simulations are run with the same parameters and an average can be taken to find the mean Euclidean drift. In Figure 5.1, 100 simulated signals are performed. This shows there is a range over which the drift can reach in the given 10 *s* with a mean of 1.06 *m* and a standard deviation of 0.55 *m*. This drift appears approximately linear on the log-log plot.

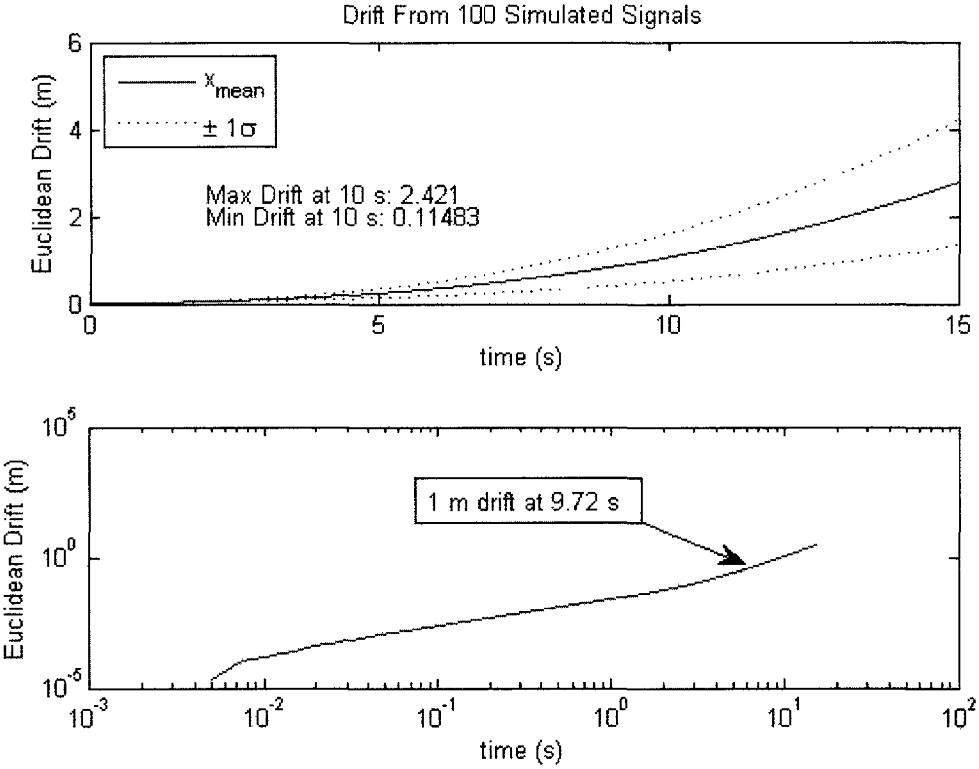


Figure 5.1: Mean Euclidean Drift for fabricated signals with average noise values

These initial results for average noise values look promising, but other noise parameters must be considered. The effects of different levels of white noise and pink noise must now be considered. First using the average values for pink noise, the levels of white noise are varied on both accelerometers and gyros, creating a contour plot. The color represents the time it takes to reach 1 m drift with different combinations of white noise on each of the sensors. The ranges considered here include the range found from the ADEV analysis found in tables 4.4, 4.5, 4.6 and 4.7, plus and minus one order of magnitude. Thus, the sensor noise levels will most likely fall near the center of the plots. In Figure 5.2, the center of the plot shows approximately 10 s, which is exactly what is desired. The broader region in the middle of the graph shows between 8 and 12 s, which is also acceptable.

A similar analysis can be made for variations in pink noise. In Figure 5.2, we see another surface plot representing the time it takes to reach 1 m drift for different noise levels, this time varying pink noise. This appears to be a random pattern sug-

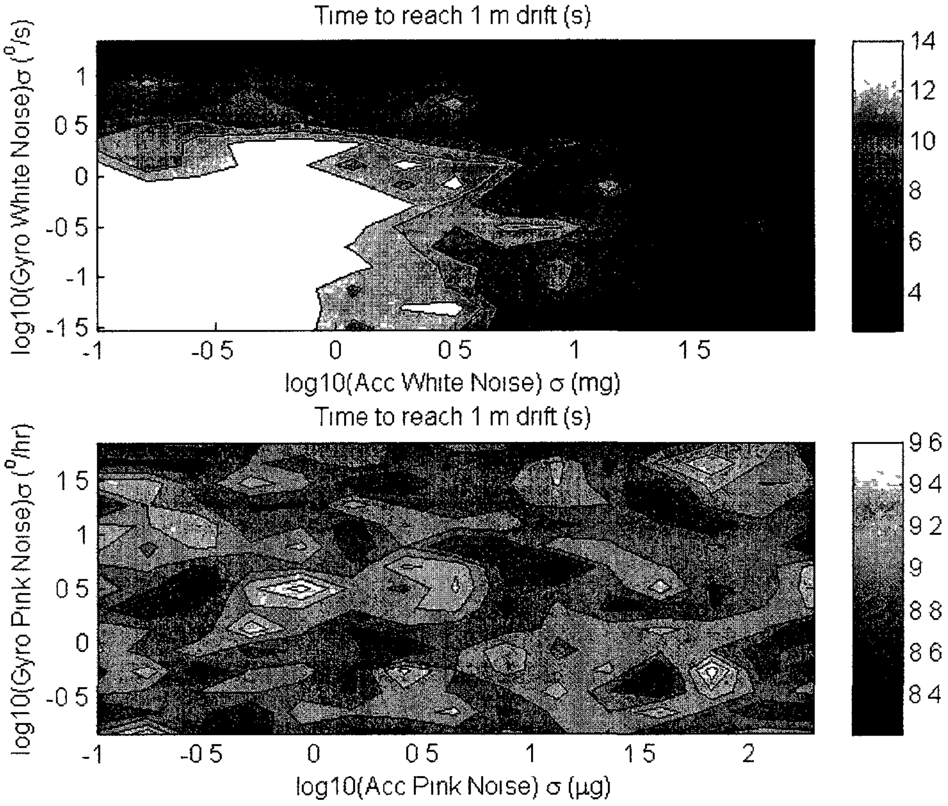


Figure 5.2: Time to reach 1 m drift varying white noise level.

gesting very little effect of variations in pink noise on overall performance. Woodman concludes that, for periods of 1 - 10 seconds, white noise has a much more dominant effect than pink noise [11]. This is in good agreement with our findings here, and only white noise will be considered in further simulations.

5.3 Stationary Testing

Several strings of data were then taken from the actual IMU to compare how the drift propagates. In Figure 5.3, an example of the raw data taken during these tests is shown.

Figure 5.4 shows a plot similar to 5.1, but this time using the actual IMU data. For 20 actual signals from the IMU, we see an average drift at 10 s of 0.57 m with a standard deviation of 0.19 m. While this is somewhat lower than the simulated

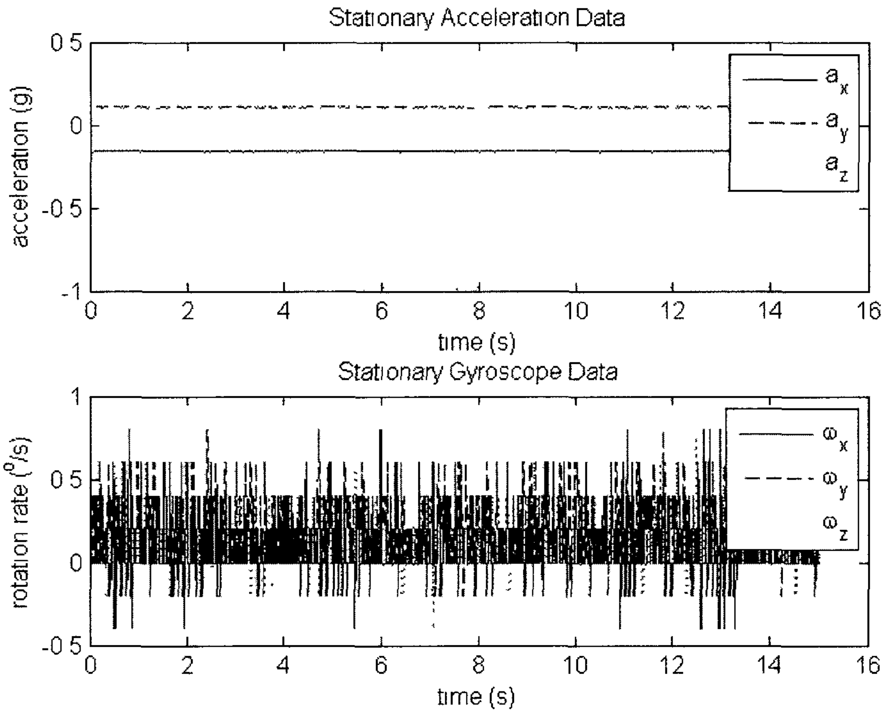


Figure 5.3: Raw IMU signals from stationary testing

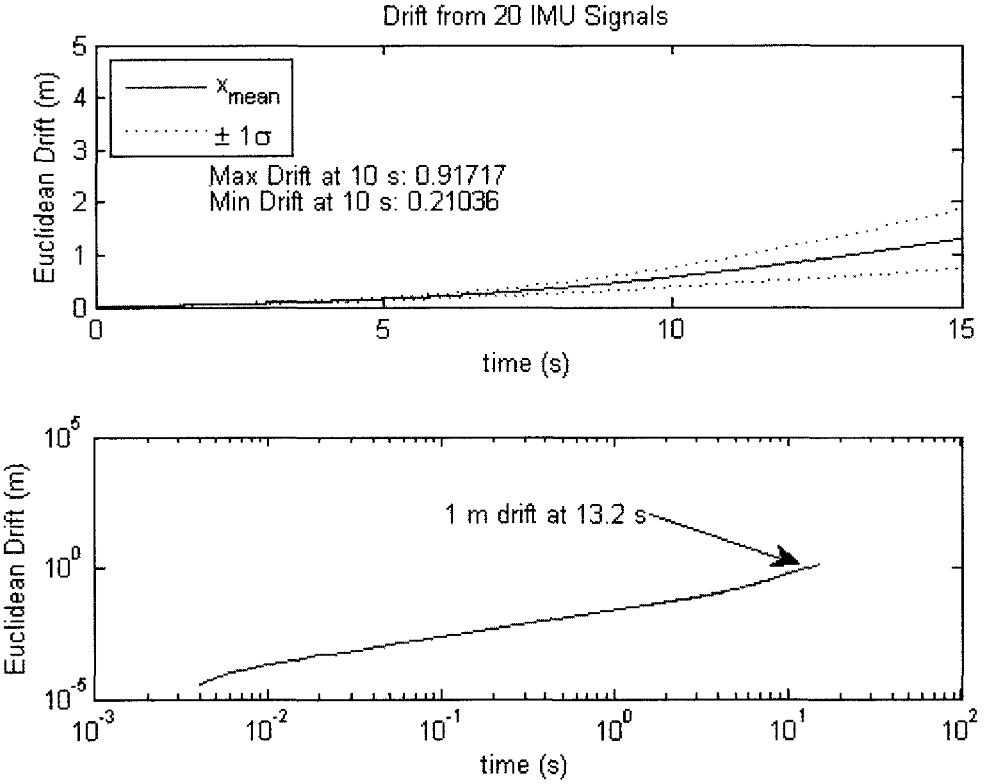


Figure 5.4: Mean Euclidean Drift for actual IMU signals

results, the mean of the test data is within the 1σ of the simulated mean. Thus, it is confirmed that the drift, when stationary, will remain in the range of 1 m after a period of 10 s.

5.4 Motion Simulation

In order to simulate motions, four virtual IMU data sets were created to represent four different motions down a 31° slope over a length of 60 m. These sets represent:

1. A straight slide down 60 m

2. Sliding down the 60 m while oscillating side to side

3. Rolling straight down the entire length

4. Sliding down while oscillating in the two transverse directions

These are all simple motions that are representative of what the Smart Rock may actually encounter in the slide. The fourth motion was simulated because it can be easily tested by carrying the IMU over a 60 m length, as it will experience these random vibrations in the transverse directions.

For each of these sets, the prescribed noise was added and run through the navigation calculations. The results for the second set is shown in Figure 5.5 as an example.

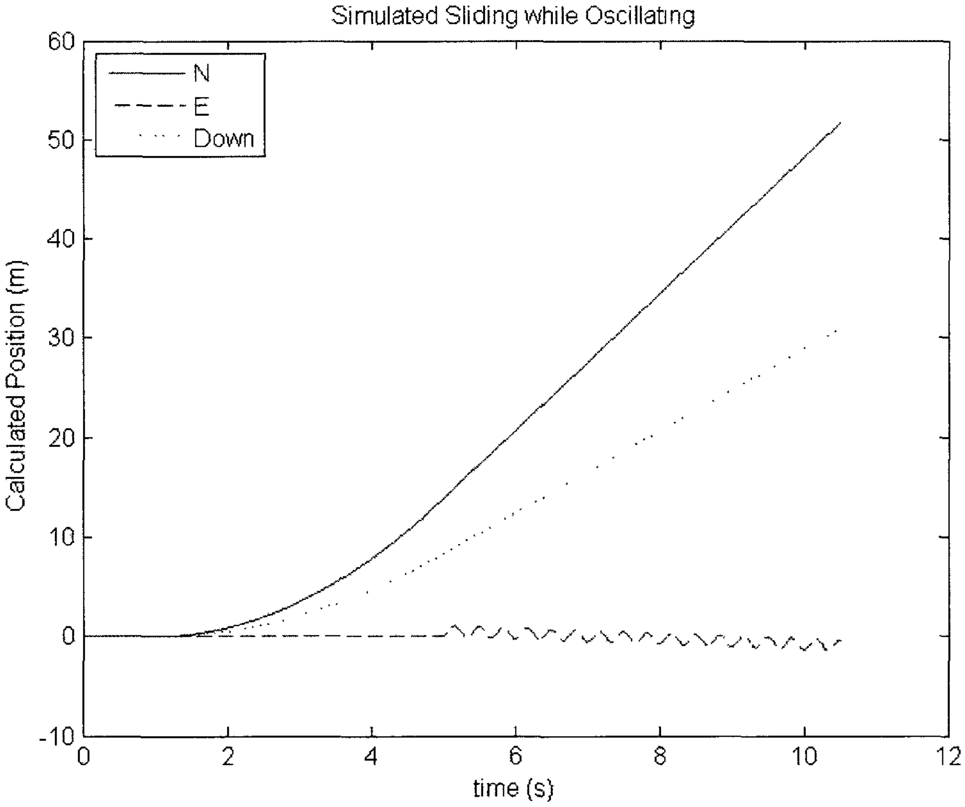


Figure 5.5: Simulated results for a downhill slide with side-to-side oscillations

This shows the rock moving along the 31^o slope over distances of approximately $60\sin(31^o) = 30.90m$ in the downward direction and $60\cos(31^o) = 51.43m$ in the North direction. This also distinguishes the 1 m amplitude oscillations in the East direction. At the end of the slide, the error is 0.13 m. Similar results are found for the other motions. Each of these has errors less than 1.5 m after ten seconds. This appears to meet the original goal, but in the next testing section, we see this is not realistic.

5.5 Motion Testing

In order to experimentally validate these simulations, actual data sets must be taken of similar motions. The second simulated data set as described in Section 5.4 is used for this. This simulated data set is recreated experimentally by having a subject hold the Rock in one hand while running 60 m. This moves the IMU the desired 60 m while introducing oscillations in a transverse direction. The results from this data set are now discussed.

In Figure 5.6, we see what should be 5 seconds of no movement, followed by 19 seconds of movement over the 60 m, and 6 seconds with no motion at the end. After 10 seconds of motion (at the 15 second mark), the error has grown to 200 m. At the end of the run, the error has grown to 927 m and continues to increase quadratically, past the 24 second mark when it should be stationary. This clearly does not match the simulations as described in Section 5.4.

While the previous test was primarily translation with limited rotation, now a test with primarily rotational motion and limited translation is discussed. This test rotates the Rock about all three axes at varying rates with translational motion that is limited to 5 cm in any direction. This is to further exemplify the errors caused by different motions. In Figure 5.7, the error caused during this motion is shown.

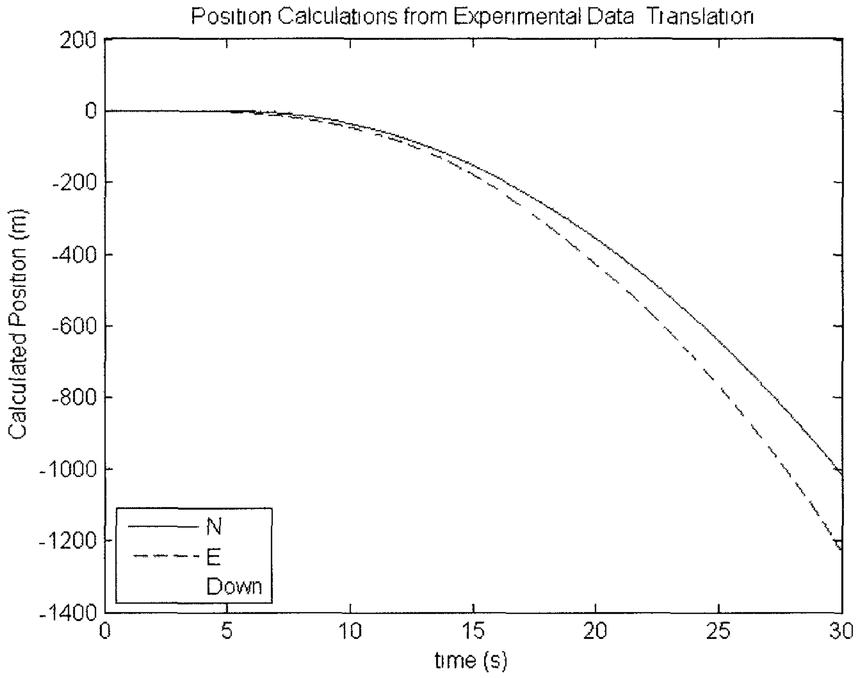


Figure 5.6: Experimental results for primarily translational motion

Again, this begins with a period of no motion, followed by the remaining time under rotation. In this experiment, the errors accumulate very quickly, reaching 1 m after 0.5 s, and over 800 m after 10 s.

5.6 Discrepancy Reconciliation

Because the experimental data and simulations do not match, this difference must be reconciled. What was not yet considered in the simulations was a constant bias on either the accelerometer or gyros. A first order analysis shows that a bias on the accelerometer would integrate twice and directly cause a quadratic drift. The specifications for the ADIS16367 show that this bias is within $\pm 50mg$ on each axis. If all three axes are at the extreme value while staying in spec, this would cause an error of 220 m after 24 seconds. This is not enough to account for the errors seen in the previous section. Gyroscope bias cannot be examined as easily.

To examine gyroscope bias and to fully understand the interaction between each sensor bias, noise, etc., simulations from set 2 as introduced in section 5.4 are run with varying levels of bias on each sensor to find what amount of drift this may cause. This produces plots similar to those shown in the Stationary Simulation section 5.2. Figure 5.8 shows the drift levels caused by noise and varying levels of accelerometer and gyro bias.

Here we see the error is rather insensitive to accelerometer bias, but very sensitive to gyroscope bias. With all three gyroscopes in spec, but at the maximum allowable value of bias at $3^o/s$, an error of over $3000m$ will occur. A bias of less than $1^o/s$ would be necessary to account for the errors that were shown experimentally. An example time series of this is shown in Figure 5.9.

Additionally, it is seen that errors in the calculated position over time is motion-dependent. For simple motions, such as a simple translation in one or two dimensions,

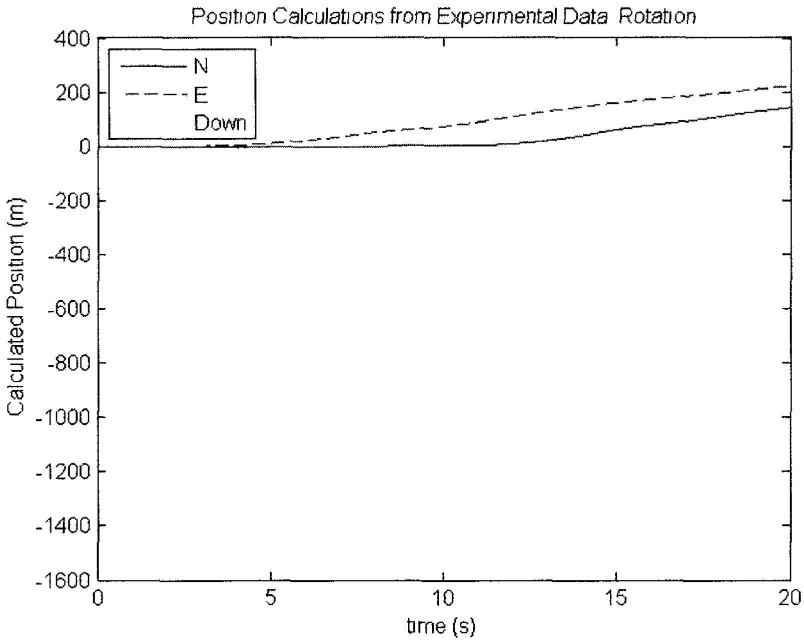


Figure 5.7: Experimental results for primarily rotational motion

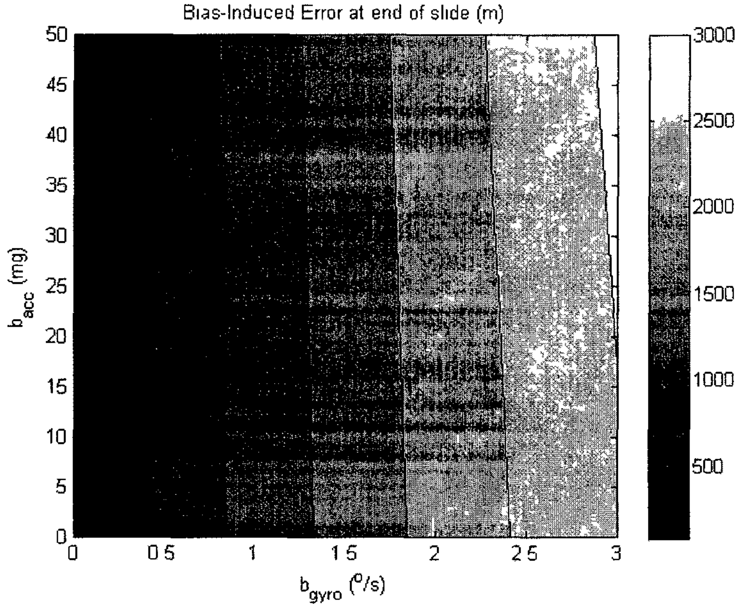


Figure 5.8: Drift caused by noise and varying levels of sensor bias

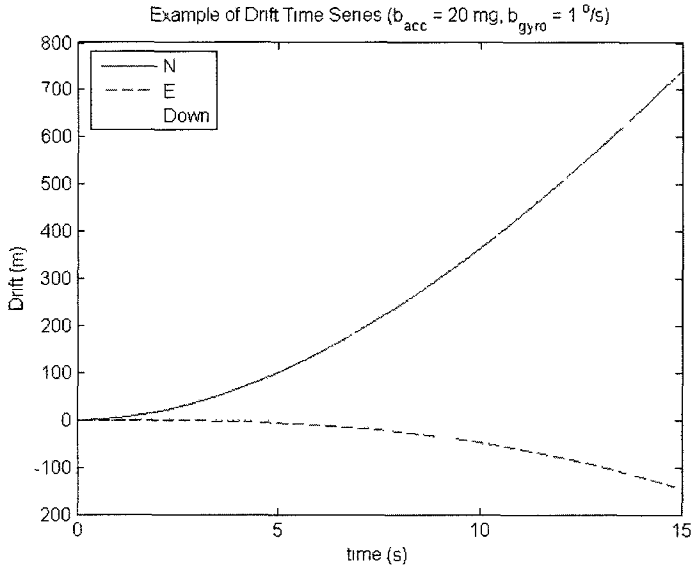


Figure 5.9: Sample time series plot of bias effects.

the calculated errors will be much lower than what is calculated from a more complex motion, such as the oscillations described in the Motion Testing Section 5.5. To show an example of this, cases 1 and 4 from section 5.4 are compared in Figure 5.10.

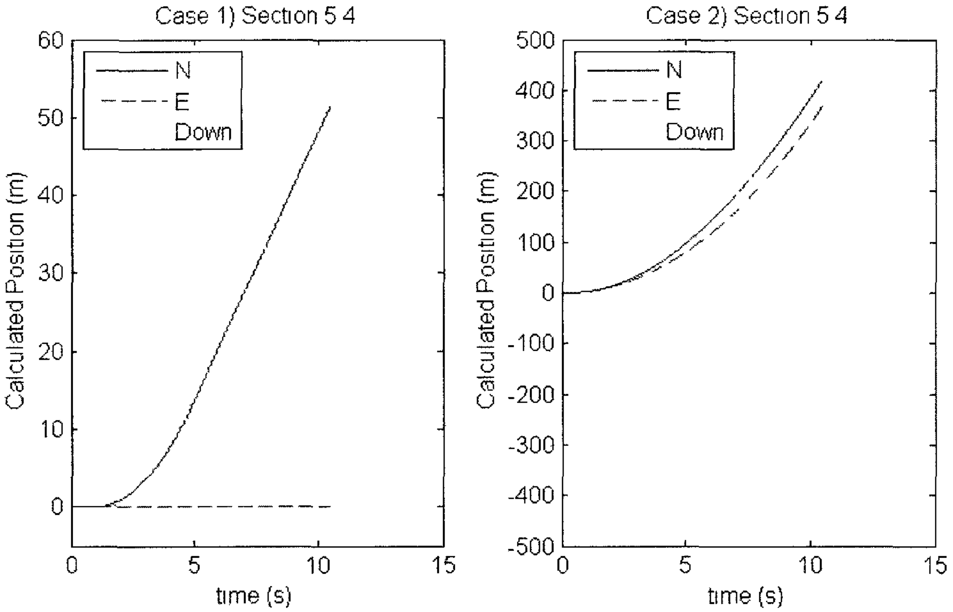


Figure 5.10: Comparison of errors caused by simple and complex motions with sensor noise and bias.

On the left, the calculated position from a two dimensional translation is shown. This is calculated from a simulated signal with the noise model described previously with no bias. This position shows the movement is almost exactly 60 m as desired. On the right, the error calculated over a motion experiencing oscillations is shown for comparison. This simulates the same motion, with added rotational oscillations. As can be seen, the errors accumulate quadratically and the error at the end of the slide is well over 300 m due to the increased complexity of the motion.

This type of oscillatory motion will necessarily be encountered during a slide. Thus, it can be concluded from this simulation and testing that it will not be sufficient to use only the inertial data to calculate position.

5.7 Pressure Sensor Testing

In order to test the pressure sensors, the Rock was first placed in a pressure chamber of air. A known pressure can be applied in air and this is measured by the sensors. This was the same procedure used to calibrate the sensors. Unsurprisingly, these results match perfectly with the applied pressure in air.

This was also tested in water, submerging the Rock to a depth of 27.7" for 1 psi and to 55.4" for 2 psi. This proves the sensors' ability to measure both gas and liquid pressure.

Finally, the dynamic response was tested by submerging quickly submerging the Rock in water and removing it. This was also done after sitting in dry air for 10 minutes, then again after sitting in dry soil for 10 minutes. Each of these showed the same dynamic response. While it was only tested up to $\approx 5\ Hz$, this is near the maximum we expect to see. This shows that preparing the pressure interface in advance and allowing it to partially dry will not allow enough air to enter the system that it would affect the dynamics.

5.8 Summary

Simulation of a stationary IMU suggests the errors caused by this noise should not exceed a few meters after $10s$. This stationary simulation is verified experimentally. However, testing and simulations of the IMU in motion show the errors will grow at an unacceptable rate. This error can be attributed to the uncertainty in attitude. When correcting for gravity in the navigation calculations, 1 g is subtracted from the global downward direction. Because the attitude is now known with high accuracy, this subtraction of 1 g to correct for gravity is subtracted in the incorrect direction. This causes a fictitious acceleration in a different direction. This is illustrated in

Figure 5.11.

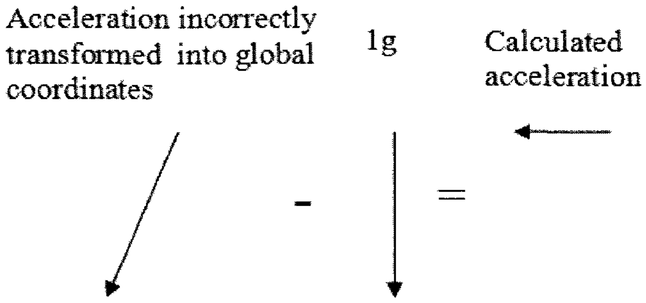

Acceleration incorrectly transformed into global coordinates
1g
Calculated acceleration
-
=

Figure 5.11: Illustrated effect on calculated acceleration by miscalculated attitude.

It is deemed that, using only the inertial data will not be sufficient to accurately calculate the position of the Smart Rock during a slide. In the next chapter, an ad hoc filter is created to bound the position errors, improving the performance of the navigation algorithm. This new algorithm will then be tested via simulation and with field tests.

It has also been shown that the pressure interface will perform adequately in both air (gas pressure) and water (liquid pressure), and that the dynamics will not be affected by the minimal amount of air that may enter the system during preparations.

Chapter 6

Application Specific Inertial Navigation Filter

6.1 Introduction

It has been shown that the unaided inertial navigation system will not provide sufficient accuracy in this application over the time frame that is desired. In this chapter, an ad hoc filter is created using the information that is known about the trajectory and velocity. The filter will constrain the calculated position to a cylindrical region about a known nominal trajectory.

6.2 Filter Assumptions

In order to create this ad hoc filter, the known information about the position and velocity must be considered.

- Initial position will be set as the origin.

- Final position will be known (after finding the Smart Rock at the bottom of the flume).

- Intermediate positions may be known from cameras focused on the slide. However, this will not give accurate information about orientation or velocity.

- The beginning, end, and any known locations in between will be referred to as "set points".

- Nominal trajectories can be found, connecting subsequent known set points with straight lines.

- The Smart Rock will stay within a finite distance ($\approx$1m) of nominal trajectory between the known set points. This forms cylinders between each of the set points.

- When reaching the outer edges of these cylinders, the velocity can have no component pointing further away from the nominal trajectory.

We will now use these assumptions to develop a simple correction algorithm that will increase the accuracy of the calculated position.

6.3 Filter Overview

In this section, an illustrated overview of the filter is given, which is built upon in the next section with further detail.

First, the set points are found (points of known position). Between these, a nominal trajectory can be created. This is shown in Figure 6.1.

The position is constrained to a region about this set point. As illustrated in Figure 6.2, a 1 m deviation is allowed.

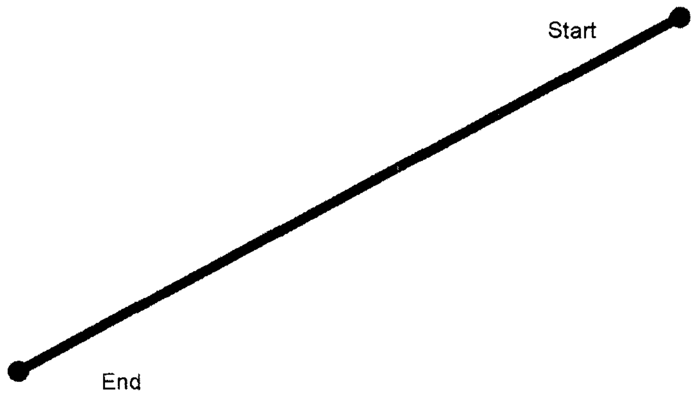


Figure 6.1: Filter nominal trajectory example.

When beginning the filter, the typical inertial navigation equations are used. At some point, the position of the Rock will be calculated to cross the allowed boundary, as shown in Figure 6.3.

Once this occurs, it is known that the position is inaccurate and a correction must occur. The position is back tracked until the angle between the velocity and the nominal trajectory is some selected θ_{crit}. In Figure 6.4, this is shown as 15°.

At this point the correction is made, both the attitude and velocity is rotated

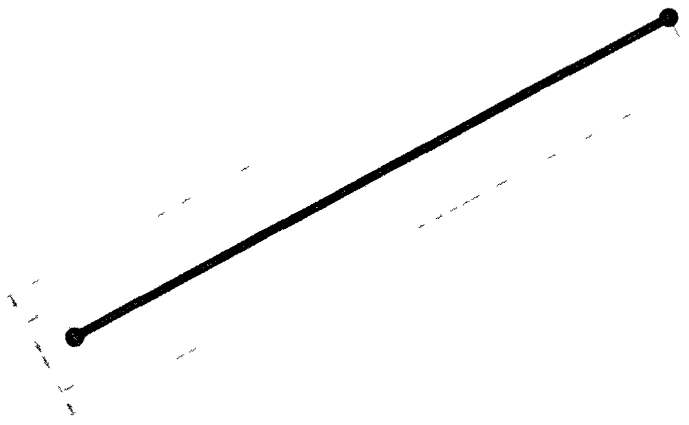


Figure 6.2: Filter nominal trajectory with boundaries.

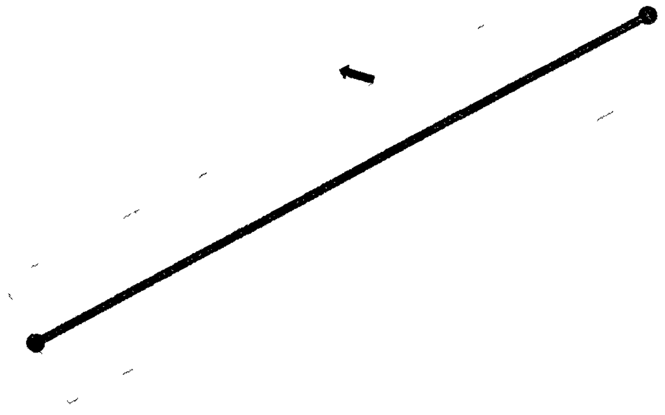


Figure 6.3: Rock crossing boundary after typical inertial navigation equations

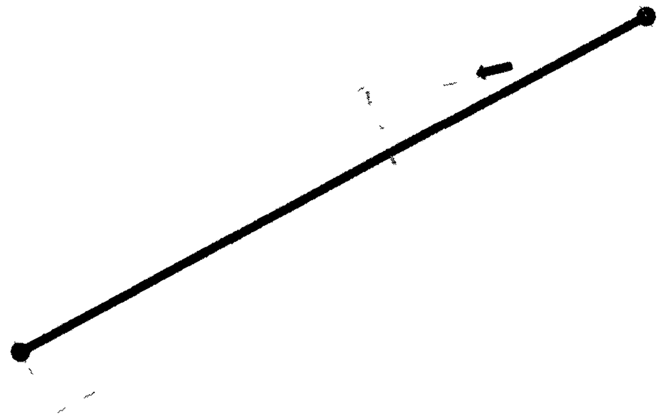


Figure 6.4: Filter backtracked to $\theta_{crit} = 15°$.

such that it is pointing down the slope. This allows the typical inertial navigation algorithm to continue from this point on a corrected trajectory.

6.4 Correction Algorithm

The filter algorithm uses all the components described in Chapter 2 regarding the basics of inertial navigation. Just as described, this algorithm loops through time steps calculating the attitude quaternion from the gyroscope measurements, transforming accelerometer measurements from body axes to local geographic axes, correcting for gravity, and integrating twice to obtain position.

In addition to these basic calculations, at each time step, the distance between the current position and the nominal trajectory is calculated. If this distance is greater than some selected d_{crit}, a correction must be made. The angle, θ, between the nominal trajectory and the velocity vector is calculated. This angle is a measure of how quickly the position is diverging from its assumed nominal trajectory. The filter then steps backwards in time until this angle is less than some selected θ_{crit}. This condition indicates that the Smart Rock is heading close to the correct direction. At this point, the velocity vector is rotated such that it is parallel to the nominal trajectory, ensuring that the position is no longer diverging. The attitude quaternion is rotated in the same manner.

The parameters d_{crit} and θ_{crit} are chosen based on the expected motion for a given experiment. A greater d_{crit} allows greater deviation from the expected nominal trajectory. The angle θ_{crit} is chosen based on the expected volatility of the motion. An example for 2D motion is shown in Figure 6.5.

Lower values of either d_{crit} or θ_{crit} put greater restrictions on the motion of the Rock and should be chosen according to the individual situation. This correction scheme is summarized in the following pseudo-code.

```
At each time step
    Calculate attitude quaternion from gyro measurements
    Calculate acceleration in local geographic coordinates
    Correct for gravity
    Integrate to find velocity
    Integrate to find position
```

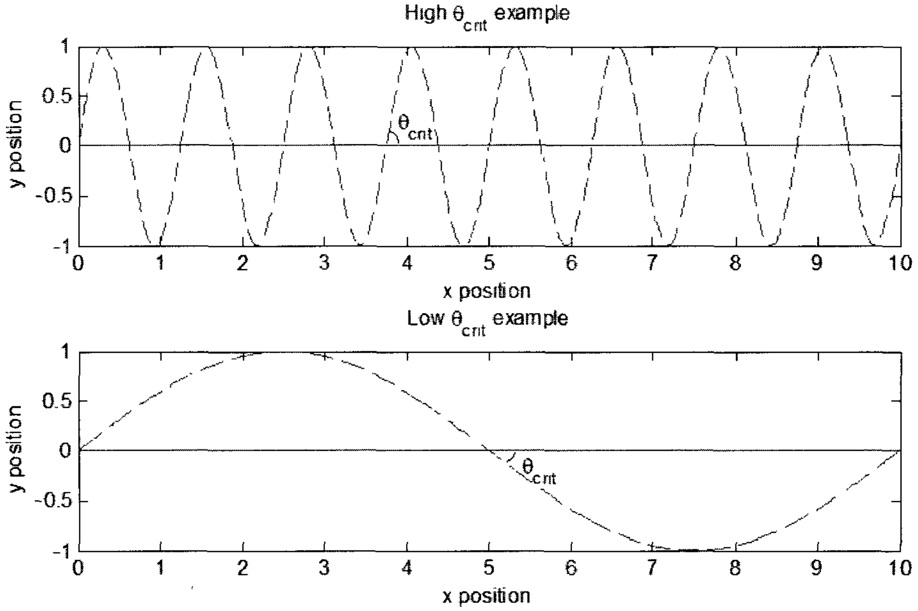


Figure 6.5: Example to show how to choose θ_{crit}

```
    d = distance away from nominal trajectory
    if d > d_crit
        theta = angle between velocity and nominal trajectory
        backtrack to where theta < theta_crit
        rotate velocity and attitude towards nominal trajectory
    end
end
```

With this method, a nominal trajectory is created from known positions at known times. The calculated position of the Smart Rock is forced to be within d_{crit} meters of the nominal trajectory.

In addition to the corrections shown in the preceding pseudo-code, situation dependent options can be used in the filter. If it is determined that the Smart Rock is moving primarily in a vertical plane, its motion can be restricted to two dimensions. This is done by removing any portion of the sensed body force (in local geographic coordinates) normal to this plane. Similarly, if it is determined that the Smart Rock is moving primarily in a horizontal plane, such as at the bottom of the slide where it may be moving on flat ground, the vertical component of the body force is removed. restricting motion to the horizontal plane.

Finally, this entire filter and correction algorithm is incorporated into something similar to the Kalman Smoother [17]. This takes the whole filter, runs it forwards in time as well as backwards in time. The two resulting data sets can then be averaged together in some fashion. A simple weighting that changes linearly in time can be used, allowing the early points to be mostly affected by the forward filter and the later points to be mostly affected by the backwards filter. This results in three total sets of results to examine; forward, backward, and smoothed; each which may be used for extracting different information about whatever may be desired.

While this does bound the error of the calculated position, there are some concerns. It is known that noise on the gyros causes errors in calculated attitude. Because of this attitude error, the measured body force from the accelerometers is not correctly transformed from body axes to local geographic axes. When the gravity vector in the local geographic downward direction is subtracted from this inaccurate body force, a fictitious acceleration component is added. When the acceleration is double integrated, a parabolic trajectory can be seen, caused by the fictitious acceleration component.

This algorithm is also illustrated in the flow chart in Figure 6.6.

6.5 Summary

In this chapter, due to the need for more accurate position calculations, a simple filtering algorithm has been created. This takes any known locations of the Smart Rock, incorporating at a minimum the start and end points, and forces the calculated

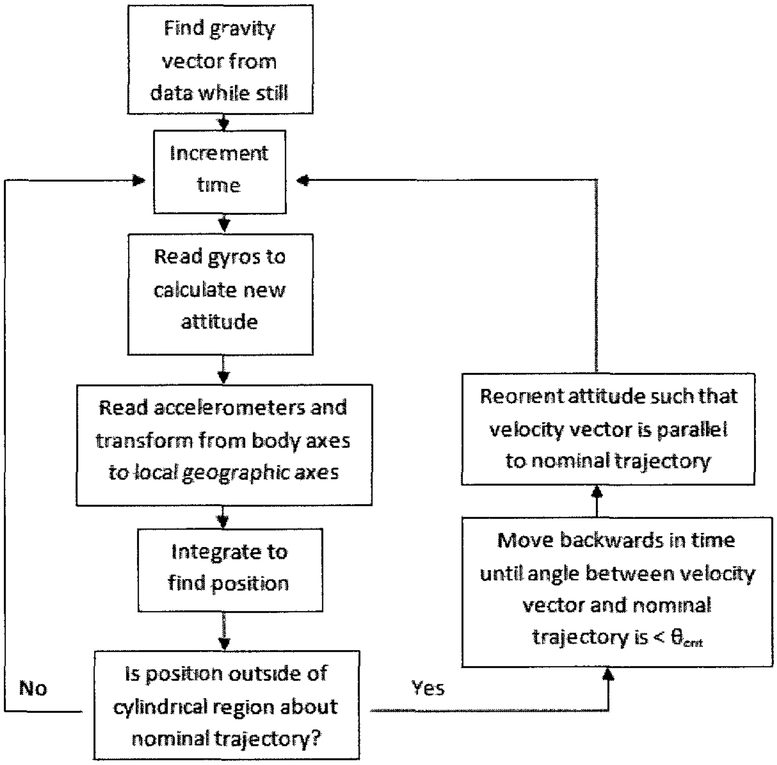

Find gravity vector from data while still
Increment time
Read gyros to calculate new attitude
Read accelerometers and transform from body axes to local geographic axes
Integrate to find position
Is position outside of cylindrical region about nominal trajectory?
No
Yes
Move backwards in time until angle between velocity vector and nominal trajectory is < θ_{crit}
Reorient attitude such that velocity vector is parallel to nominal trajectory

Figure 6.6: Flow chart of navigation filter

position to follow a trajectory moving through these points.

Motion is restricted to a cylindrical region about a nominal trajectory, bounding the error as desired. If more restrictions are deemed appropriate for a given experiment, the motion can be restricted to two dimensions in either a horizontal or vertical plane. This data can also be run through a forward and backward smoother to give a final result for the calculated position.

Chapter 7

Results

7.1 Introduction

Now that the need for a filter has been determined and the filter has been developed, this chapter discusses the results and performance of the IMU system with the filter. In order to properly evaluate its performance, datasets that are representative of that in the slide must be taken experimentally. Using various tuning parameters and different numbers of set points, the overall efficacy of the filter can be determined. It is found that the filter is able to provide highly accurate position estimates given few known intermediate positions.

7.2 Experimental Data Acquisition

In order to obtain the necessary experimental data sets, the Smart Rock is transported over a distance of 50 m by holding the device and quickly running along a straight path. This induces movement primarily in one direction, with low amplitude oscillations in the position, as well as low rotation rates. These characteristics described for this experimental data set match very well with what is expected in the flume.

In order to measure the absolute position, video tracking is used. Several markers are placed along the course that are used to mark the distance traveled. The video captures at what times the rock passes each of these points, providing 26 known 3D coordinates along the path.

7.3 Tracking Position

The data from the previously described test has been obtained and is now used in several ways to track position. First, just the raw data is used in the process described in Chapter 2. Following this, the filter developed in Chapter 6 is implemented. This is done with different numbers of known positions, or set points, taken from the video to find how this affects the overall accuracy.

7.3.1 Raw Data Results

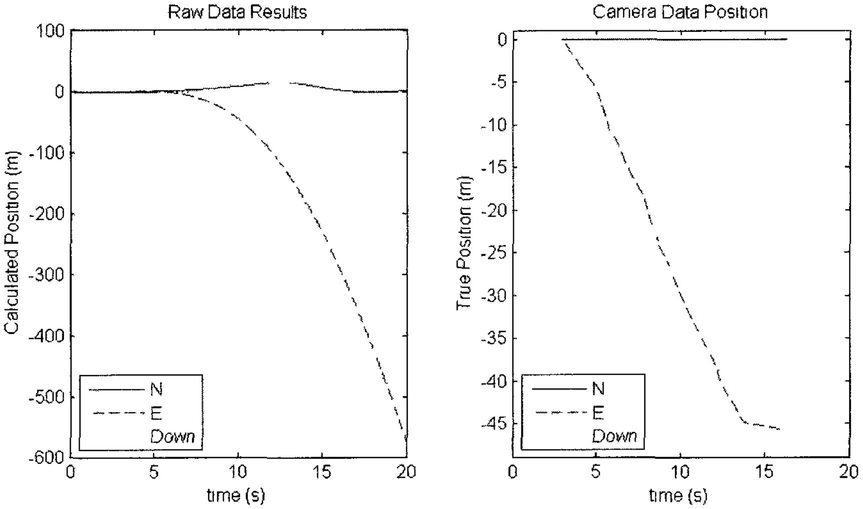


Figure 7.1: Experimental results using only raw data

Using solely the raw data, the position is calculated. The results from this are presented in Figure 7.1. The Smart Rock is oriented such that East is pointing in the direction of motion. When compared with the true 3D coordinates found from the video, the calculated position is seen to have errors that grow very quickly. The error grows to 33.1 m after 10 s and to over 200 m at the end of this run.

7.3.2 Filtered Results: 2 Set Points

As was seen, the errors grow unacceptably when using only the raw data. Here, the filter is implemented with two set points, at the beginning and end of the run. In this run, as well as those following that use the filter, the parameters for the critical distance and critical trajectory angle are set at $d_{crit} = 1.5m$ and $\theta_{crit} = 30°$ based on the type of motion.

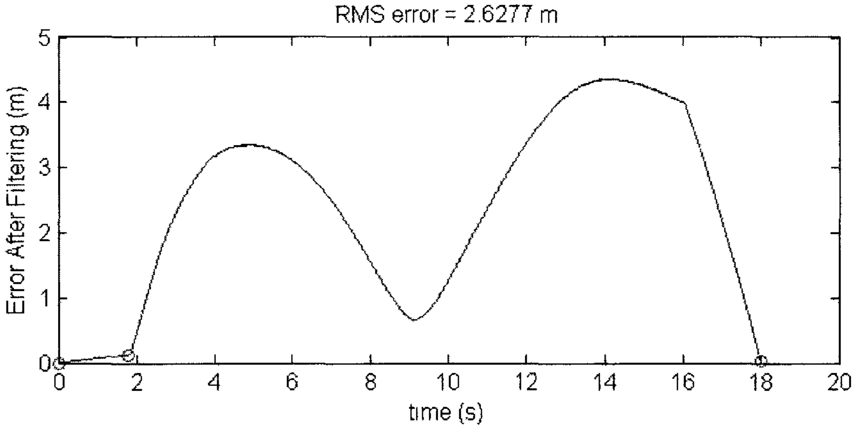


Figure 7.2: Experimental results using the filter with 2 set points

Figure 7.2 shows the results from the smoother, using data from the forward and backward filters. This forces the position to begin and end at given locations. As marked by the circled points, the set points constrain the error to relatively low values. In the middle of the run, there is another section where the error decreases. This results from the averaging method used to create the smoothed data set. The accuracy can be quantified by the root-mean-square (RMS) error over this run, which is 2.63 m.

7.3.3 Filtered Results: 3 Set Points

Now, one additional set point is added in the middle of the run to see how the accuracy improves. This result is shown in Figure 7.3.

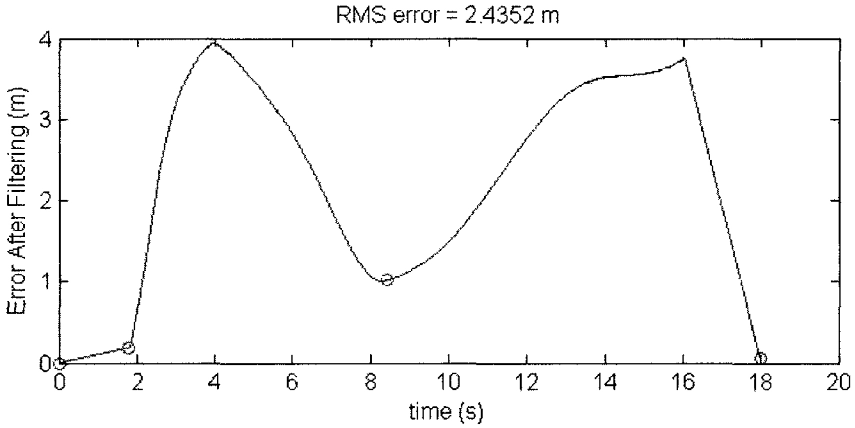


Figure 7.3: Experimental results using the filter with 3 set points

The error at the beginning and end remain low, while the error in the middle of the run is further decreased. Overall, this has the effect of decreasing the RMS error slightly from 2.63 m to 2.44 m. While these data sets naturally have a drop in the calculated error in the middle of the run with just two set points, this may not appear when experiencing other motions. In another case where this is not present, the added set point in the middle of the run will improve the RMS error to a greater extent.

It can be noted that the middle set point, a point with known position, appears to have a non-zero error. This can be explained as follows. Set points are used to create the nominal trajectories, and these nominal trajectories are used to bound the position. As long as the position remains within the bounded region, it is assumed to be of reasonable accuracy. Bringing the error at these points to identically zero would require further correction. Additional corrections reduce the filter's ability to distinguish small scale motions. Thus, having zero error has been sacrificed for small scale motion resolution, while maintaining the desired accuracy.

7.3.4 Filtered Results: 5 Set Points

Continuing with this trend, two more set points are added, for a total of five. This greatly reduces the portion of time when the error grows relatively high. The overall effect is to reduce the RMS error further to 1.39 m.

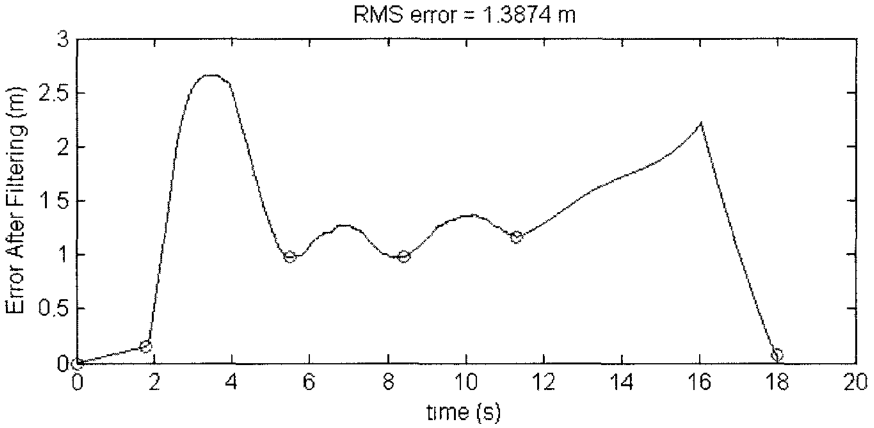


Figure 7.4: Experimental results using the filter with 5 set points

7.3.5 Filtered Results: 26 Set Points

Finally, all of the set points gained from the video data will be used with the filter. This will help to set a theoretical minimum for the RMS error. These points are now spaced very tightly throughout the time where motion is greatest. The total of 26 set points decreases the RMS error to 1.14 m. While this does lead to a modest improvement in RMS error, it does not result in significantly greater knowledge about the process.

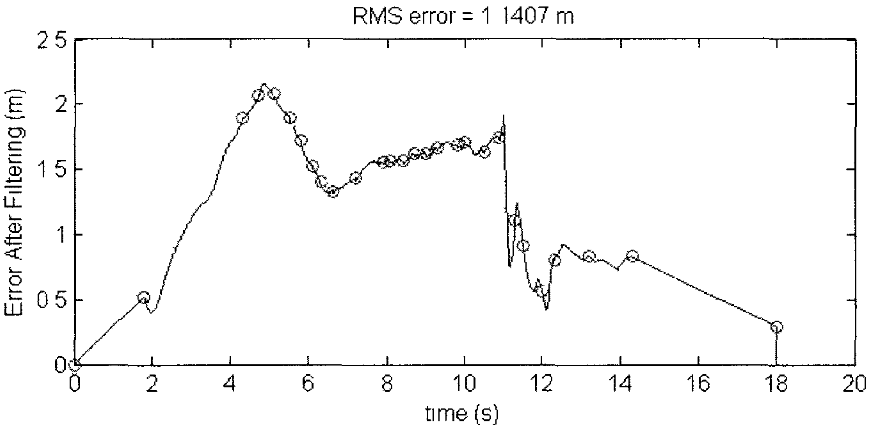


Figure 7.5: Experimental results using the filter with 26 set points

7.4 Small Motion Resolution

In addition to calculating position for the large scale motions as was investigated above, small motions must also be resolved in order to gain the most information

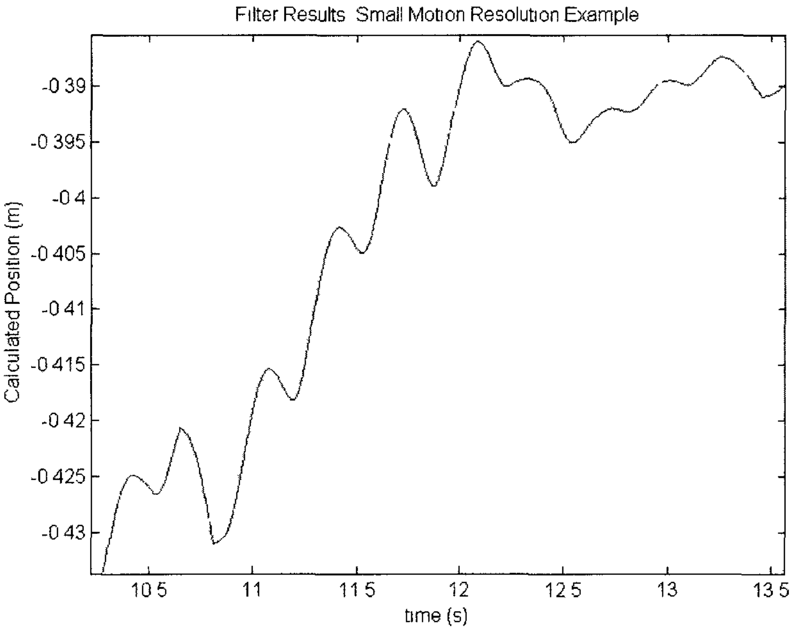


Figure 7.6: Experimental results using the filter, examining small motion resolution

from the Smart Rock. To evaluate the filter's ability to resolve these small motions, the same data sets will be investigated over a shorter time period.

Here, we see a zoomed in region of the results from the forward filter using 5 set points. Oscillations with amplitudes on the order of 5 *cm* can be seen in both of the transverse directions. In various other motions, similar types of oscillations can be resolved. Similar motions can be seen in the forward and backward filter. When averaging these, some of the small scale motions are corrupted and are not as easily distinguished. This shows that, in addition to tracking position over a large scale (tens of meters), small scale motions (on the order of centimeters) can be identified and resolved.

7.5 Rolling Motion

While it is not expected that the Smart Rock experience great rates of rotation, this situation is still briefly investigated. The Rock is placed on an inclined, grassy surface and allowed to roll 6 *m*. This motion nearly saturates the gyroscopes. Results are compared with and without the filter.

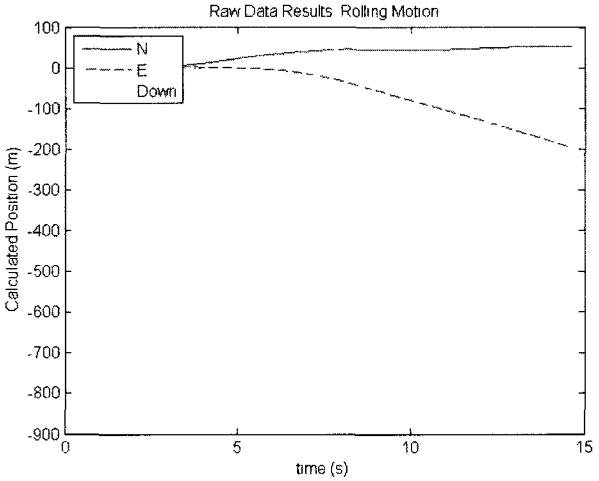


Figure 7.7: Experimental results from rolling motion without filtering

This shows a similar result to what has been shown previously. The error grows very quickly once motion begins. In this run, motion began at 3.5 s. Now this can be compared to the filtered results.

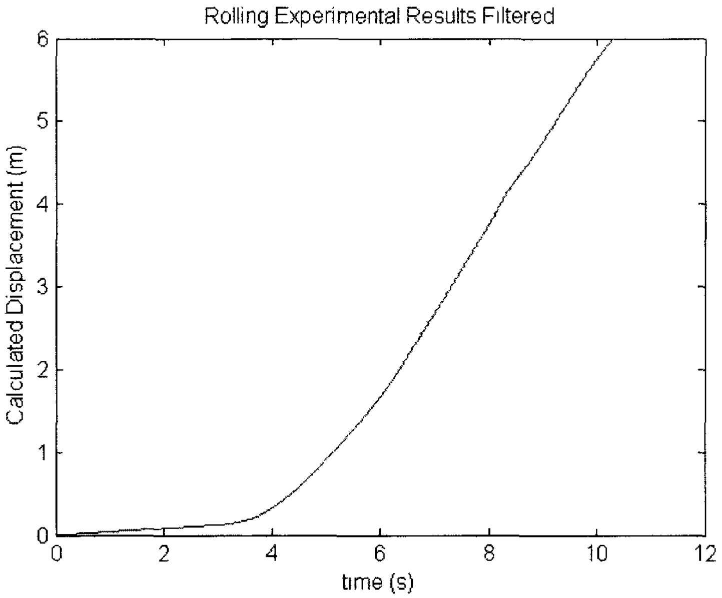


Figure 7.8: Experimental results from rolling motion with filtering

Figure 7.8 shows the filter's calculated displacement using the known set point of 6 m. This comes to precisely 6 m at the end of the slide. Thus, even in extreme motions involving high rotation rates that nearly saturate the sensors, the filter still provides very accurate position information.

7.6 Straight Sliding Motion

In addition, the Smart Rock was tested under a straight sliding motion. This motion was created by placing the Rock on a smooth, straight, inclined surface at approximately 20^o (i.e. a playground slide). This allowed the motion to be constrained to essentially a single degree of freedom. Again, this results in hundreds of meters of error, shown in Figure 7.9. While this is lower than the error found in a more complex

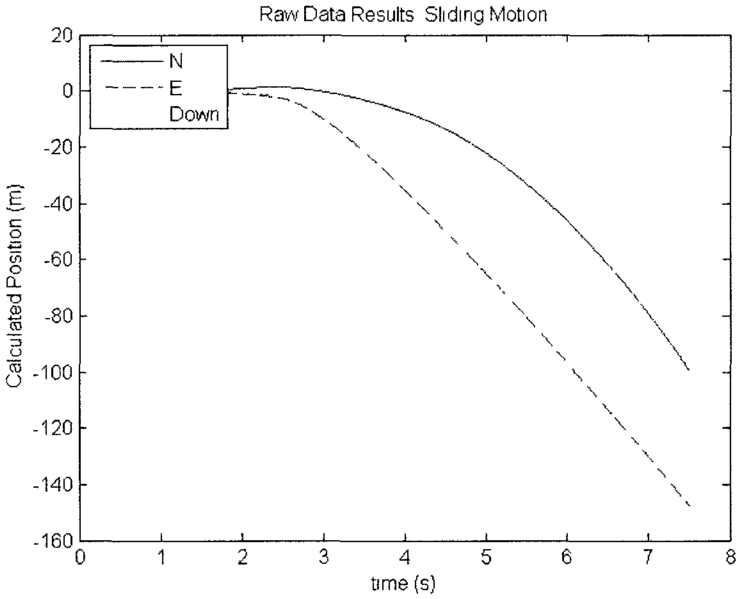


Figure 7.9: Experimental results from sliding motion without filtering

motion, it is still unacceptable.

Illustrated in Figure 7.10, the filtered results show, again, precisely 6 *m* of displace-

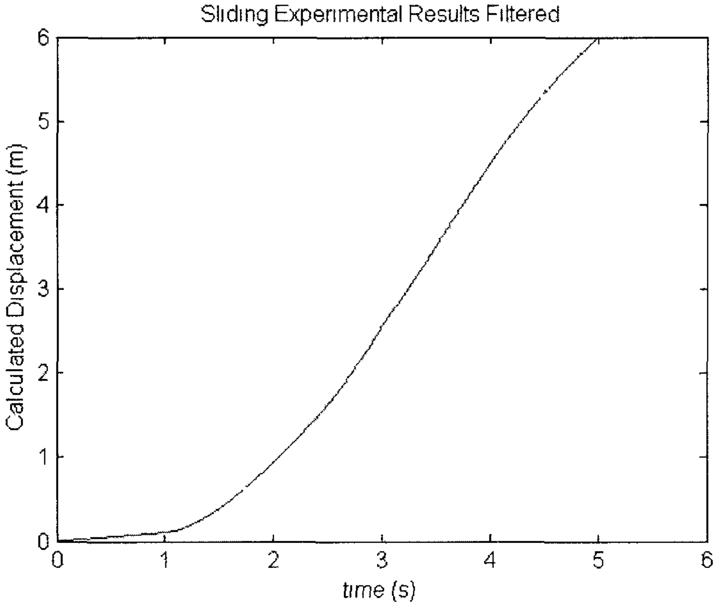


Figure 7.10: Experimental results from sliding motion with filtering

ment over the 4 s motion after 1 s remaining stationary. This continues to exemplify the ability of the filter to take the data from any given motion and accurately calculate position with only the beginning and end positions being known.

7.7 Summary

In this chapter, the IMU data is used in several different ways to calculate the position over time. First, only the raw data is used. This is again seen to induce errors that are not acceptable for our application.

Next, the ad hoc filter was implemented using various numbers of set points. Starting with just two, one at the beginning and one at the end, a great improvement in error is seen, reducing the end error to less than $1.5m$, with an RMS error of 2.63 m.

When adding one additional point in the middle of the run, the RMS error is reduced to 2.44 m. A further reduction is seen in RMS error by adding two more set points, bringing it down to 1.39 m. Finally, all the set points from the video data are used, totaling 26 set points used in the filter. This reduces the RMS error to 1.14 m.

Additionally, the filter is further verified by allowing the Smart Rock to undergo two more motions. These included a motion rolling down an incline (primarily rotation) and sliding down a similar incline (primarily translation). These filtered results showed similar improvements of the calculated position.

From this, we see that just the beginning and end points will provide a great improvement in accuracy. Further, any additional points in the middle of the run will increase accuracy to a greater extent, but are not necessary to adequately calculate the position. Based on these findings, 3 to 5 would be the optimal number of set points for a run of up to 16 s.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This study has focused on creating a device to measure the internal pressures, accelerations, and rotation rates of particles in a landslide flow. This was inspired by work from the U.S. Geological Survey (USGS). In order to study this type of debris flow phenomenon, the USGS has constructed a concrete flume in which these events can be reproduced in a controlled environment. This allows for measurements of debris depth, pressure and shear forces at the walls, but these are only measurements at fixed points. In order to take measurements along the flow, a new instrument is needed. Thus, the Smart Rock has been designed, assembled, and tested. It is desired that this Rock be able to gather data from inertial sensors and pressure sensors for over 10 minutes and to use this data to track the position with an accuracy of 1 m over the course of 10 s and to examine the decay of pore water pressure.

This Smart Rock includes several sensors with a microprocessor that can log the data to a μSD card. The sensors onboard include two fluid pressure sensors on opposite sides of the Rock as well as a triad of accelerometers and a triad of gyroscopes. With the six inertial sensors, a series of calculations can be performed to calculate the position of the Rock over time. This inertial navigation process is very sensitive to errors in the signals.

In order to quantify the different errors and noise types on the gyroscopes and accelerometers, the Allan Variance procedure was employed. This time domain calculation takes a long series of data and takes the averages of this data broken into equal-sized bins. It then finds how the variance of these averages changes with respect to the size of the bins. This procedure is able to distinguish what types of noise are on the signal and to quantify each of these. This reveals that the primary noise sources of concern are white and pink noise. These are quantified and it is found that the errors are quite low compared with other MEMS sensors. However, the errors on these sensors are orders of magnitude greater than those typically used in an inertial navigation system.

With the sensor noise quantified, the signals can be simulated for various motions. These simulations are compared to experimental data. It is found that these match quite well for a stationary IMU and that the drift is below the desired 1 m over the course of a slide. However, when in motion, the errors grow much more quickly. Several factors account for this, such as inaccuracies of numerical integration and saturation. The primary reason, however, relates to the inaccuracies in calculated attitude and gravity correction as was illustrated in Figure 5.11. When in motion, the acceleration magnitudes are greater, and when the attitude is not known accurately, this larger acceleration magnitude creates a larger velocity component in an incorrect direction, thus the growing error.

In order to improve the accuracy of the calculated position, an ad hoc filter is created. This filter takes the inertial data as well as any known positions, and constrains the calculated position to a small region. It was evaluated with various parameters and numbers of known positions. This significantly improves the calculation and meets the desired accuracy.

From this, it has been found that without any aiding from additional information or other sensors, the calculated position is not reasonably accurate. The errors grow to hundreds of meters within 10 s. With the filtering method that has been developed, these errors are constrained, and the accuracy improves to less than 5 m error over 10 s. This will allow the debris to be tracked with great accuracy and to verify and improve the current analytical debris flow models.

8.2 Future Work

The accuracy of the calculated position can be improved with the addition of other sensors. Typically, inertial navigation systems are aided with a GPS or a magnetometer. However, these were found to be impractical in this application. If another sensor can be included, this will undoubtedly improve the accuracy of the calculated position and reduce the need for the filter.

Something such as a ultrasonic range finder or RFID system was considered, but was deemed beyond the scope of this project. Implementing one of these types of sensor systems would require on site development and testing to determine feasibility. If something like these is found to be possible, it could be installed in the current Rock, as there is additional space in the top of the shell.

Beyond improvements to the Rock, the data must be gathered from a flume experiment and analyzed. Data on acceleration and rotation rate can be correlated with the pressure measurements and movements can be compared to those suggested by analytical models.

Bibliography

[1] 2005, Mount St. Helens - From the 1980 Eruption to 2000, *U.S. Geological Survey*, `http://pubs.usgs.gov/fs/2000/fs036-00/`, (January 21, 2011)

[2] Britting, K. R., 1971, *Inertial Navigation Systems Analysis*, John Wiley & Sons, Inc., New York City, p. 1-10.

[3] 2011, MECH307, Mechatronics and Measurement Systems, *Colorado State University*, `http://www.engr.colostate.edu/~dga/mech307/`, (May 16, 2011)

[4] 2011, Spacecraft Guidance, Navigation and Control Systems, *what-when-how*, `http://www.http://what-when-how.com/space-science-and-technology/spacecraft-guidance-navigation-and-control-systems/`, (July 11, 2011)

[5] Titterton, D., and Weston, J., 2005, *Strapdown Inertial Navigation Technology*, The Institution of Engineering and Technology, Stevenage, UK, p. ...

[6] Analog Devices, "Six Degrees of Freedom Inertial Sensor," ADIS16367 datasheet, Jan. 2010.

[7] Olivares, A., Olivares, G., Gorriz, J.M., Ramirez, J., 2009, "High-efficiency low-cost accelerometer-aided gyroscope calibration ," International Conference on Test and Measurement 2009, pp. 4-7.

[8] IEEE STD 1293, 1998, IEEE Standard Specification Format Guide and Test Procedure for Linear, Single-Axis, Non-Gyroscopic Accelerometers, pp. 55-59.

[9] Senturia, S., 2001, *Microsystem Design*, Kluwer Academic Publishers, Norwell, MA, p. 436-441.

[10] Kosko, B., 2006, *Noise*, Penguin Group, New York, p. 90-94.

[11] Woodman, O., 2007, "An introduction to inertial navigation," Technical Report No. 696, University of Cambridge, Cambridge, United Kingdom.

[12] IEEE STD 647, 2006, IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Laser Gyros, pp. 62-73.

[13] Kim, H., Lee, J.G., Park, C.G., 2004, "Performance Improvement of GPS/INS Integrated System Using Allan Variance Analysis," International Symposium on GNSS/GPS, pp. 2-6.

[14] Stockwell, W., "Bias Stability Measurement: Allan Variance:" Crossbow, Milpitas, CA, http://www.xbow.com/pdf/Bias_Stability_Measurement.pdf.

[15] IEEE STD 952, 1997, IEEE Standard Specification Format Guide and Test Procedure for Single-Axis Interferometric Fiber Optic Gyros, pp. 62-72.

[16] Stockwell, W., "Angle Random Walk," Crossbow, Milpitas, CA, http://www.xbow.com/pdf/AngleRandomWalkAppNote.pdf.

[17] Gelb, A., 1974, *Applied Optimal Estimation*, The MIT Press, Cambridge, MA, p. 157-173.

Appendix A

Detailed Strapdown Inertial Navigation Calculations

A.1 Introduction

Here, we discuss the inertial navigation calculations in full detail. A strapdown system is considered with the sensor axes aligned with the "body axes". The attitude calculations use the quaternion method introduced in Chapter 2 to find the relationship between the "body axes" and the "local geographic" axes. These "local geographic" axes point North, East, and down toward the center of the earth. The acceleration calculations then take the acceleration readings, convert them from body axes to local geographic axes, and track the position. This algorithm is taken from the Titterton and Weston [5].

A.2 Attitude Calculations

To solve for the attitude using quaternions, the following equation must be solved

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \cdot \mathbf{p} \tag{A.1}$$

Here, $\mathbf{q}$ is the attitude quaternion and $\mathbf{p} = [0, \omega^T]$ represents how the attitude is changing. It can also be rewritten as

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{W}\mathbf{q} \tag{A.2}$$

where

$$\mathbf{W} = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{A.3}$$

and $\boldsymbol{\omega}$ are the gyroscope readings. It is reasonable to assume that $\boldsymbol{\omega}$ remains constant over the time interval being considered. This is possible because the calculations are performed at every sample. Thus, there is no information showing that $\boldsymbol{\omega}$ would change over a given interval. This assumption allows the discretization of the differential equation.

$$\mathbf{q_{i+1}} = \left[\exp\frac{1}{2}\int_i^{i+1} \mathbf{W}\mathrm{dt}\right]\mathbf{q_1} \tag{A.4}$$

This can be simplified using

$$\int_i^{i+1} \mathbf{W}\mathrm{dt} = \Sigma = \begin{bmatrix} 0 & -\sigma_x & -\sigma_y & -\sigma_z \\ \sigma_x & 0 & \sigma_z & -\sigma_y \\ \sigma_y & -\sigma_z & 0 & \sigma_x \\ \sigma_z & \sigma_y & -\sigma_x & 0 \end{bmatrix} \tag{A.5}$$

This is because $\int_i^{i+1} \boldsymbol{\omega}\mathrm{dt} = \boldsymbol{\sigma}$. Therefore, we have

$$\mathbf{q_{i+1}} = \exp\left(\frac{\Sigma}{2}\right)\mathbf{q_i} \tag{A.6}$$

This can also be represented in a different form eliminating the matrix exponential.

$$\mathbf{r_i} = \begin{bmatrix} \cos\left(\frac{\sigma}{2}\right) \\ (\sigma_x/\sigma)\sin\left({}^{\sigma}/_{2}\right) \\ (\sigma_y/\sigma)\sin\left({}^{\sigma}/_{2}\right) \\ (\sigma_z/\sigma)\sin\left({}^{\sigma}/_{2}\right) \end{bmatrix} \tag{A.7}$$

$$\mathbf{q_{i+1}} = \mathbf{q_i} \cdot \mathbf{r_i} \tag{A.8}$$

From the definition of a quaternion in Chapter 2, we see that $\mathbf{r_i}$ is a quaternion representing a rotation of magnitude σ about $\boldsymbol{\sigma}$.

This means that from the integral of the gyroscope output, we can find Σ, allowing us to find the new $\mathbf{q_{i+1}}$. The quaternion can also be normalised because its magnitude must always be unity.

$$\mathbf{q_{i+1}} = \frac{\mathbf{q_{i+1}}}{||\mathbf{q_{i+1}}||} \tag{A.9}$$

A.3 Position Calculations

The specific force measured by the accelerometers is represented in body axes as $\mathbf{f^b}$. This must be transformed to local geographic navigation axes, $\mathbf{f^n}$.

$$\mathbf{f^n} = \mathbf{C^n_b A f^b} \tag{A.10}$$

The matrix $\mathbf{C^n_b}$ is the direction cosine representation of the quaternion found in the previous section. If the quaternion has components a, b, c, and d;

$$\mathbf{C^n_b} = \begin{bmatrix} 1-2\,(c^2+d^2) & 2\,(bc-ad) & 2\,(bd+ac) \\ 2\,(bc+ad) & 1-2\,(b^2+d^2) & 2\,(cd-ab) \\ 2\,(bd-ac) & 2\,(cd+ab) & 1-2\,(b^2+c^2) \end{bmatrix} \tag{A.11}$$

The matrix $\mathbf{A}$ represents the transformation from body axes at time t_k, to body axes at time t_{k+1}. The matrix C^n_b makes the transformation from body axes at time t_{k+1} to local geographic coordinates at time t_{k+1}.

$$\mathbf{A} = I + \frac{\sin(\sigma)}{\sigma}[\boldsymbol{\sigma}\times] + \frac{1-\cos(\sigma)}{\sigma}[\boldsymbol{\sigma}\times]^2 \tag{A.12}$$

In this equation for $\mathbf{A}$, $\boldsymbol{\sigma}\times$ is a skew symmetric form of $\boldsymbol{\sigma}$.

$$\boldsymbol{\sigma}\times = \begin{bmatrix} 0 & -\sigma_z & \sigma_y \\ \sigma_z & 0 & -\sigma_x \\ -\sigma_y & \sigma_x & 0 \end{bmatrix} \tag{A.13}$$

With the specific force in the local geographic axes, gravity is subtracted and the next velocity can be found.

$$\mathbf{v_{i+1}} = \mathbf{v_i} + \int_{\imath}^{\imath+1} (\mathbf{f^n} - \mathbf{g})\,\mathrm{d}t \tag{A.14}$$

Finally, corrections can be made for the Coriolis force and the rotation of the earth.

$$v_{\imath+1} = v_{\imath+1}\left(I - 2\int_{\imath}^{\imath+1} \boldsymbol{\Omega_{ie}}\mathrm{dt} - \int_{\imath}^{\imath+1} \boldsymbol{\Omega_{en}}\mathrm{dt}\right) \tag{A.15}$$

The matrices $\boldsymbol{\Omega_{ie}}$ and $\boldsymbol{\Omega_{en}}$ are skew symmetric forms of the vectors $\boldsymbol{\omega_{ie}}$ and $\boldsymbol{\omega_{en}}$.

$$\boldsymbol{\omega_{ie}} = \begin{bmatrix} \boldsymbol{\Omega}\cos L \\ 0 \\ -\boldsymbol{\Omega}\sin L \end{bmatrix}, \boldsymbol{\omega_{en}} = \begin{bmatrix} \frac{v_{E,\imath}}{R_0+h} \\ \frac{-v_{N,\imath}}{R_0+h} \\ \frac{-v_{E,\imath}\tan L}{R_0+h} \end{bmatrix} \tag{A.16}$$

In these equations, R_0 is the radius of the earth,L is the body's current latitude and $\boldsymbol{\Omega}$ is the rotation rate of the Earth. The velocities in the east and north directions are shown as v_E and v_N, with the height above the Earth shown as h.

To find position, a simple integration must be performed.

$$x_{i+1} = x_i + \int_i^{i+1} v\mathrm{d}t \tag{A.17}$$

This now gives a complete description of how to find the position over time.

Appendix B

Firmware

This section discusses the firmware that is running on the Arduino microprocessor. We will now walk through each step of the program to explain all of its inner workings. For those who are interested in the entire code, it is protected under the GNU GPL v3 and can be found at `http://code.google.com/p/smart-rock-logger/`.

This code depends on the Timer1 library created by Jesse Tane (`http://code.google.com/p/arduino-timerone/`), and the SdFat library by Bill Greiman (`http://code.google.com/p/sdfatlib/`). It also requires that an SD card socket be connected to the standard Arduino SPI bus, an ADIS16367 IMU be connected to the software SPI bus as outlined in the code, as well as two analog inputs on A0 and A1, and two LEDs from GPIO pins 7 and 8.

The code begins by initializing many global variables and defining the addresses of all of the IMU registers. It then goes into the `setup()` section. In this section, all communication lines are initialized. This includes the hardware SPI bus to the SD card, the software SPI bus to the IMU, the ADC for reading pressures, as well as initializing the pins to drive the LEDs. It also initializes a volume on the card in which to create the text file.

In the next section of code, the SD interface is put to use. It searches the card's root directory for a configuration file. If one is not found, the default values that were set when the variables were initialized are used. If a configuration file is found, it reads through each character, taking the first grouping of numbers as the delay time, second as the sampling frequency, and third as the time to record. If there are any further characters in the document, they are ignored.

A configuration file contains the three values discussed in the previous paragraph; delay time, sampling frequency, and sample time; as shown in this example.

```
10 400 300
```

This sample configuration file contents indicates a 10 s delay time, followed by 300 s of recording data at 400 samples per second.

If there is a delay time, it occurs at this point. After the delay, the sensor is configured. Registers are set to configure the digital filter to four taps, apply a correction on the gyro bias, setup the data ready indicator, and perform a self-test.

It then tries to find a text file titled DATA00.TXT. If this already exists, the number is incremented until the file does not yet exist, and it is created. A cache is initialized that points to the volume containing this newly created text document. Another data buffer is created to compile the samples as they are read. It is initialized as a series of spaces with new line characters. This allows for the data to be read in and written, one sample per line.

Now that everything has been configured satisfactorily, the recording begins. The timer is initialized that will trigger an ISR (interrupt service routine). In the ISR, all sensors are read, the data is formatted as a series of hex characters, and they are written into the appropriate location of the data buffer. The ISR also keeps track of how many samples are in the buffer with a simple counter. Once the buffer is full, it is copied to the cache location such that it can be appended to the text file. A flag is set when the data begins to be written to the file. If the buffer becomes full before the previous block of data is written completely, the flag indicates this and the data in the buffer is dumped. This prevents any hangups that may occur in the SD communication lines.

After the prescribed amount of time, the timer is then deactivated. At the bottom of the text file, a few lines are written to indicate the recording has completed successfully. It also prints the number of skipped data blocks, maximum amount of time it took to write a single data block, the sampling frequency, and the contents of the IMU diagnostic register to show if any errors occurred.

Finally, the file is closed and communication to the card is cut off. The appropriate register bits are set to sleep the IMU. The Arduino is then put into standby mode to conserve power further. The code then enters an infinite loop of NOP (no operation) calls for the remainder of the time that power is supplied.

This process creates a text file with columns of data in hexadecimal format, as well as certain diagnostic information as supplementary information. The recording process was streamlined to facilitate the fastest sample rates possible. It also conserves power by using certain functionality of the IMU and microcontroller.

Here is a sample excerpt of a resulting data text file. It has columns of hex numbers for time, gyroscope x, y, and z readings, accelerometer x, y, and z readings, as well as the two pressure senor readings.

```
002845 0002 3FF7 3FFB 3F68 00E5 0071 0CC 097
002847 0002 3FF7 3FFB 3F68 00E4 0071 0CC 0B6
002849 0003 3FF7 3FFC 3F68 00E4 0071 0CC 0C1
00284B 0003 3FF7 3FFD 3F68 00E4 0071 0CC 0C6
00284F 0003 3FF6 3FFE 3F68 00E3 0071 0CC 0C7
002851 0003 3FF6 3FFE 3F68 00E3 0071 0CC 0C7
002853 0003 3FF6 3FFF 3F68 00E3 0071 0CC 0C9
```

The full text of this program can be seen here:

```
/*
 SmartRockLogger

 Creates bit-banged SPI implementation for ADIS16367
 This is able to use readAllToString() which reads
 data from sensor, writes to string, and prints it
 once the buffer has been filled

 It does so using an interrupt triggered by Timer1

 Writes to SD card once there has been a sufficient
 number of samples to fill the buffer.

 If it has not completed writing a block, but a new
 block of data is ready to be written, it skips this
 new block to finish the one in progress.

 It requires a file CONFIG.TXT to exist in the root
 directory of the SD card. It must contain 3 numbers
 the delay time (s), sampling frequency (1/s), and
 recording time (s) in that order. All other characters
 in the file are ignored. If there is no CONFIG.TXT or
 config.txt, default values are used.

 After logging, the sleep mode is initialized on the
 IMU and on the microcontroller to minimize power usage
 after logging is complete.

 One must add the files TimerOne.cpp and TimerOne.h to
 this Sketch from http://code.google.com/p/arduino-timerone/

 IMU:
 Arduino          ->   ADIS 16367
 IRQ: pin 2         ->  DIO1: pin 7
 CS: pin 6          ->  CS: pin 6
 DI/MOSI: pin 3     ->  DIN: pin 5
 DOUT/MISO: pin 5   ->  DOUT: pin 4
 SCLK: pin 4        ->  SCLK: pin 3

 SD Card:
 Arduino          ->    SD Card
 MOSI: pin 11       ->  DI
 MISO: pin 12       ->  DO
 SCK: pin 13        ->  SCK
 CS: pin 10         ->  CS
 Other:
 Arduino          ->    Other
 power LED: pin 7   ->  Red LED
 status LED: pin 8  ->  Green LED
 A0                 ->  Inst Amp 1
 A1                 ->  Inst Amp 2

 Notes:
   -  If left running for over ~4 hours 30 minutes,
     time will roll over to 0
   -  Will skip blocks of data if SD card has not
     finished writing when the new block of
     data has become available
   -  Tested and works consistently up to 540
     Hz using 1 GB Transcend uSD card
   -  Performance may vary with different uSD cards

 created 22 Feb 2011
 modified 27 June 2011
 by Matt Harding

 Thank you to:
 Bill Greiman for his SdFat library
   http://code.google.com/p/sdfatlib/
 Jesse Tane for the Timer1 library
   http://perfectverse.com/jesse/portfolio/projects/timerOne/...
    index.html

    Copyright 2011 Matthew Harding

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

```
#include <SdFat.h>
#include <SdFatUtil.h>
#include "TimerOne.h"

unsigned int fs = 100;
unsigned long time = 5;
const unsigned int sample_size = 46;
unsigned int samples_per_block = 512/sample_size;
unsigned long delaytime = 1;

// counters etc
volatile unsigned int samplecounter = 0;
volatile unsigned int counter = 0;
volatile boolean readyForNextBlock = true;
volatile unsigned int skippedBlocks = 0;

// data strings
volatile char dataString[512];
uint8_t* pCache;

// pins
const byte dataReadyPin = 2;
const byte chipSelectPin = 6;
const byte mosiPin = 3;
const byte misoPin = 5;
const byte sclkPin = 4;
const byte SDcs = 10;
const byte powerLEDpin = 7;
const byte statusLEDpin = 8;

Sd2Card card;
SdVolume volume;
SdFile root;
SdFile file;
SdFile configfile;

uint32_t bgnBlock, endBlock;

// store error strings in flash to save RAM
#define error(s) error_P(PSTR(s))

void error_P(const char* str) {
  PgmPrint("error: ");
  SerialPrintln_P(str);
  if (card.errorCode()) {
    PgmPrint("SD error: ");
    Serial.print(card.errorCode(), HEX);
    Serial.print(',');
    Serial.println(card.errorData(), HEX);
  }
  while(1);
}

//Sensor's memory register addresses:
const byte SUPPLY_OUT = 0x02;    // Power supply reading
const byte XGYRO_OUT = 0x04;     // X gyro
const byte YGYRO_OUT = 0x06;     // Y gyro
const byte ZGYRO_OUT = 0x08;     // Z gyro
const byte XACCL_OUT = 0x0A;     // X acceleration
const byte YACCL_OUT = 0x0C;     // Y acceleration
const byte ZACCL_OUT = 0x0E;     // Z acceleration
const byte XTEMP_OUT = 0x10;     // X temperature
const byte YTEMP_OUT = 0x12;     // Y temperature
const byte ZTEMP_OUT = 0x14;     // Z temperature
const byte AUX_ADC = 0x16;       // Auxilary ADC
const byte XGYRO_OFF= 0x1A;      // X-gyro offset
const byte YGYRO_OFF= 0x1C;      // Y-gyro offset
const byte ZGYRO_OFF= 0x1E;      // Z-gyro offset
const byte XACCL_OFF= 0x20;      // X-accelerometer offset
const byte YACCL_OFF= 0x22;      // Y-accelerometer offset
const byte ZACCL_OFF= 0x24;      // Z-accelerometer offset
const byte ALM_MAG1 = 0x26;      // Alarm 1 amplitude threshold
const byte ALM_MAG2 = 0x28;      // Alarm 2 amplitude threshold
const byte ALM_SMPL1 = 0x2A;     // Alarm 1 sample size
const byte ALM_SMPL2 = 0x2C;     // Alarm 2 sample size
const byte ALM_CTRL = 0x2E;      // Alarm control
const byte AUX_DAC = 0x30;       // Auxiliary DAC
const byte GPIO_CTRL = 0x32;     // Auxiliary digital I/O control
const byte MSC_CTRL = 0x34;      // Data-ready, self-test, etc.
const byte SMPL_PRD = 0x36;      // internal sample rate control
const byte SENS_AVG = 0x38;      // range and filter control
const byte SLP_CNT = 0x3A;       // Sleep mode control
const byte DIAG_STAT = 0x3C;     // System status
const byte GLOB_CMD = 0x3E;      // System Command
const byte PROD_ID = 0x56;       // Product ID number (16367)
const byte SERIAL_NUM = 0x58;    // Serial number

//Sensor commands
const byte READ = 0b00000000;  // ADIS16367 read command (2 bytes)
const byte WRITE = 0b10000000; // ADIS16367 write command
const byte BURST_READ = 0x3E;  // ADIS16367 burst read command

void setup() {
  // Clear sleep mode just to make sure it doesn't shut off
  clear_sleep();

  // Show power LED
  pinMode(powerLEDpin, OUTPUT);
  pinMode(statusLEDpin, OUTPUT);
  digitalWrite(powerLEDpin, HIGH);
  digitalWrite(statusLEDpin, LOW);

  // Setup communications
  Serial.begin(115200);

  // Set ADC to clock prescale of 8 (~10x faster than default)
  initADC();

  Serial.flush();

  // Initialize ADIS SPI bus
  SPIbegin(chipSelectPin);

  // Initialize SD card
  unsigned long start = millis();
  pinMode(SDcs, OUTPUT);
  digitalWrite(SDcs, LOW);
  // initialize the SD card at SPI_FULL_SPEED for best performance.
  // try SPI_HALF_SPEED if bus errors occur.
  if (!card.init(SPI_FULL_SPEED)) error("card.init");

  start = millis() - start;

  // initialize a FAT volume
  if (!volume.init(&card)) error("volume.init");

  // open the root directory
  if (!root.openRoot(&volume)) error("openRoot failed");
}



void loop() {
  char nextchar;
  // Look for configuration file
  if (configfile.open(&root, "CONFIG.TXT", O_READ)...
    || configfile.open(&root, "config.txt", O_READ)) {

    // Start with delay time
    delaytime = 0;
    nextchar = configfile.read();

    // Read all numbers
    while(nextchar>=’0’ && nextchar<=’9’) {
      delaytime = delaytime*10 + (nextchar-’0’);
      nextchar = configfile.read();
    }
    // Then skip all the non-numbers
    while(nextchar<=’0’ || nextchar>=’9’) {
      nextchar = configfile.read();
    }


    // And do the same for sampling frequency
    fs = 0;
    while(nextchar>=’0’ && nextchar<=’9’) {
      fs = fs*10 + (nextchar-’0’);
      nextchar = configfile.read();
    }
    while(nextchar<=’0’ || nextchar>=’9’) {
      nextchar = configfile.read();
    }


    //And finally for recording time
    time = 0;
    while(nextchar>=’0’ && nextchar<=’9’) {
      time = time*10 + (nextchar-’0’);
      nextchar = configfile.read();
    }

    // All done close the config file
    configfile.close();
  }
  else {
//    No CONFIG.TXT found, use default values set
// at variable definitions
  }
```

```
unsigned int block_count = (fs*time)/samples_per_block + 1;

// Begin Delay
delay(delaytime*1000);

// Begin device configuration
// Set range
byte whichBits1[] = {2,1,0};
byte setToWhat1[] = {1,0,0};
setRegister(SENS_AVG,HIGH,whichBits1,setToWhat1,3);
// Set filter to 4 taps
setRegister(SENS_AVG,LOW,whichBits1,setToWhat1,3);
// Precision Autonull, delay 30s (decided not to use it)
//  writeRegister(GLOB_CMD,0x10);
 //GLOB_CMD[4] = 1 (DIN = 0xBE10)
//  delay(33000);
// Correct gyro bias for acceleration
setRegister(MSC_CTRL,LOW,(byte*)7,(byte*)1,1);
 //MSC_CTRL[7] = 1 (DIN = 0xB486)
// Set DIO1 as data ready indicator
setRegister(MSC_CTRL,LOW,whichBits1,setToWhat1,3);
 //MSC_CTRL[2:0] = 100
// Command self-test, this allows diagnostic
// register to be printed at end
setRegister(MSC_CTRL,HIGH,(byte*)2,(byte*)1,1);
delay(2000);
// Set gyro offsets to 0
byte allBits[] = {7,6,5,4,3,2,1,0};
byte allZeros[] = {0,0,0,0,0,0,0,0};
setRegister(XGYRO_OFF,HIGH,allBits,allZeros,8);
setRegister(XGYRO_OFF,LOW,allBits,allZeros,8);
setRegister(YGYRO_OFF,HIGH,allBits,allZeros,8);
setRegister(YGYRO_OFF,LOW,allBits,allZeros,8);
setRegister(ZGYRO_OFF,HIGH,allBits,allZeros,8);
setRegister(ZGYRO_OFF,LOW,allBits,allZeros,8);

// Set up file to write
char filename[] = "DATA00.TXT";
byte filenum = 0;
while (file.open(&root, filename, O_READ)) {
  file.close();
  filenum++;
  filename[4] = (filenum/10)+’0’;
  filename[5] = (filenum%10)+’0’;
}
```

```
// Create a contiguous file
if (!file.createContiguous(&root, filename, 512UL*block_count)) {
  error("createContiguous failed");
}
// Get the location of the file's blocks
if (!file.contiguousRange(&bgnBlock, &endBlock)) {
  error("contiguousRange failed");
}
//*********************NOTE**************************************
// NO SdFile calls are allowed while cache is used for raw writes
//***************************************************************

// Clear the cache and use it as a 512 byte buffer
pCache = volume.cacheClear();

// Fill cache with samples_per_block lines
for(int i = 0 ; i<512 ; i++) {
  dataString[i] = ' ';
  pCache[i] = ' ';
}
// Loop through each sample
for (int i = 0 ; i < 512-sample_size; i += sample_size) {
  // And put newline/carriage return at end of line
  dataString[i] = '\r';
  dataString[i+1] = '\n';
}


// tell card to setup for multiple block write with pre-erase
if (!card.erase(bgnBlock, endBlock)) error("card.erase");
if (!card.writeStart(bgnBlock, block_count)) {
  error("writeStart");
}

// Indicate with LEDS recoding is beginning
digitalWrite(powerLEDpin, LOW);
digitalWrite(statusLEDpin, HIGH);

// Init stats
unsigned int maxWriteTime = 0;
unsigned long start = millis();
unsigned long writeTime = 0;
```

```
// Start Timer1 at fs samples/sec
Timer1.initialize((1000000UL/fs));
// Attach timer interrupt
Timer1.attachInterrupt(isr);

// Stay in loop until we reach desired time
while( (millis()-start) < (time*1000) ) {
  // If we are ready to write
  if (samplecounter==samples_per_block && readyForNextBlock) {
    // Reset sample counter
    samplecounter = 0;
    // Warn that we are not ready for a new block yet
    readyForNextBlock = false;
    // Write a 512 byte block
    writeTime = micros();
    if (!card.writeData(pCache)) {
      error("writeData");
      while(1);
    }
    // Set new block flag
    readyForNextBlock = true;
    // Calculate time it took to write block
    writeTime = micros() - writeTime;
    // Check for max write time
    if (writeTime > maxWriteTime) {
      maxWriteTime = writeTime;
    }
  }
}

// Stop the interrupt
Timer1.detachInterrupt();

// This can't be the best way to do the following,
//   but it works and it won't affect performace.

// Print how many blocks were skipped
for(int i = 0 ; i<512 ; i++) {
  pCache[i] = ' ';
}
pCache[0] = '\r';
pCache[1] = '\n';
pCache[20] = pCache[0];
pCache[21] = pCache[1];
pCache[40] = pCache[0];
```

```
pCache[41] = pCache[1];
pCache[60] = pCache[0];
pCache[61] = pCache[1];
pCache[80] = pCache[0];
pCache[81] = pCache[1];
pCache[7] = 'E';
pCache[8] = 'N';
pCache[9] = 'D';
pCache[10] = '!';
pCache[11] = pCache[10];

// Print skipped blocks
unsigned int digit = 0;
unsigned int temp = skippedBlocks;
for(int ii=0 ; ii<5 ; ii++) {
  digit = floor(temp/word(pow(10,4-ii)));
  pCache[ii+22] = digit+'0';
  temp = temp-digit*pow(10,4-ii);
}
char label[8] = "skipped";
for(int ii=0 ; ii<7 ; ii++) {
  pCache[ii+22+6] = label[ii];
}

// Print max write time in microseconds
digit = 0;
temp = maxWriteTime;
for(int ii=0 ; ii<5 ; ii++) {
  digit = floor(temp/word(pow(10,4-ii)));
  pCache[ii+42] = digit+'0';
  temp = temp-digit*pow(10,4-ii);
}
char label2[11] = "t write us";
for(int ii=0 ; ii<10 ; ii++) {
  pCache[ii+42+6] = label2[ii];
}

// Print sampling frequency
digit = 0;
temp = fs;
for(int ii=0 ; ii<5 ; ii++) {
  digit = floor(temp/word(pow(10,4-ii)));
  pCache[ii+62] = digit+'0';
  temp = temp-digit*pow(10,4-ii);
}
```

```
  char label3[3] = "fs";
  for(int ii=0 ; ii<2 ; ii++) {
    pCache[ii+62+6] = label3[ii];
  }

  // Print flags
  digit = 0;
  temp = readRegister(DIAG_STAT);
  for(int ii=0 ; ii<15 ; ii++) {
    pCache[ii+82] = (temp>>(14-ii))+’0’;
  }
  char label4[40] = "flags: see ADIS16367 datasheet Table 26";
  for(int ii=0 ; ii<39 ; ii++) {
    pCache[ii+82+16] = label4[ii];
  }


  pCache[511] = ’\0’;
  if (!card.writeData(pCache)) {
    error("writeData");
    while(1);
  }

  // End multiple block write mode
  if (!card.writeStop()) error("writeStop");

  // Close files for next pass of loop
  root.close();
  file.close();
  Serial.println();

  // Indicate with LEDS recoding is complete
  digitalWrite(powerLEDpin, HIGH);
  digitalWrite(statusLEDpin, HIGH);

  // Put sensor to sleep to save power
  setRegister(SLP_CNT,HIGH,(byte*)1,(byte*)1,1);

  // All done, stop here and put MCU to sleep
  go_to_sleep();
  while(1);
}
```

```
// The ISR is triggered by Timer1 at intervals defined
// by the sampling frequency
void isr() {
  // Read all the data to a string
  readAllToString();
  // Increment the sample counter
  samplecounter++;
  // If that is enough to fill a block,
  if(samplecounter==samples_per_block){
    // And if we are ready for the next block
    if(readyForNextBlock) {
      // Copy it to the cache so it can be written in
      //    the main loop
      strlcpy((char*)pCache,(char*)dataString,...
      samples_per_block*sample_size+1);
    }
    else {
      // Otherwise, skip it
      skippedBlocks++;
      // And reset the sample counter
      samplecounter = 0;
    }
  }
}
```

```
//Sends a read command to the ADIS16367:
int readRegister(byte thisRegister ) {
  int result = 0;   // result to return
  // take the chip select low to select the device:
  bitClear(PORTD,chipSelectPin);
  // ADIS16367 expects the register address in the lower 7 bits
  // now combine the register address and the command into one byte:
  byte dataToSend = thisRegister | READ;
  // send the device the register you want to read:
  SPItransfer2(dataToSend,0);
  delayMicroseconds(4);
  result = SPItransfer2(0,0);
  bitSet(PORTD,chipSelectPin);
  // return the result:
  return(result);
}
```

```
// Sends write command to ADIS16367
void writeRegister(byte reg, byte data) {
  bitClear(PORTD,chipSelectPin);      // Select device
  SPItransfer2(reg|WRITE,data);       // Writes data to register
  bitSet(PORTD,chipSelectPin);        // Deselect device
}


// Transfer 2 bytes via SPI and combine the results
// into a 16 bit int
int SPItransfer2(byte data1, byte data2) {
  int result = SPItransfer(data1)<<8;
  result |= SPItransfer(data2);
  return(result);
}

// Transfer a single byte via SPI
//    This deals with individual bits
byte SPItransfer(byte data) {
  // SCK begins high
  for(byte bit=0 ; bit<8 ; bit++) {
    bitClear(PORTD,sclkPin); // SCK fall low
    bitWrite(PORTD,mosiPin,(data>>7) & 0x01);
     // Write data to MOSI pin
    data = data << 1; // Shift left 1
    bitSet(PORTD,sclkPin); // SCK rise high
    data |= bitRead(PIND,misoPin);
     // read data from MISO pin
  }
  return(data);
}



// Read all registers and write to string
// String begins with new line (0-1)
// Time takes 6 hex chars (2-7)
// Each of gyro/acc takes 4 hex chars,
// 14 bits (8-11,12-15,16-19...8+i*5-8+i*5+3)
// Analog read takes 3 hex chars, 10 bits (39-41,...)
void readAllToString(void) {
  // take the chip select low to select the device
  bitClear(PORTD,chipSelectPin);
  addhexchar(dataString, millis(), 2 +...
    (samplecounter*sample_size),6);
         // Get time and write to string
  SPItransfer2(XGYRO_OUT,0);
    // Ask for X Gyro
  int data = SPItransfer2(YGYRO_OUT,0);
    // Get X Gyro and ask for Y Gyro
  addhexchar(dataString, data&0x3FFF, 9+(0*5) +...
    (samplecounter*sample_size),4);
         // Save to text string
  data = SPItransfer2(ZGYRO_OUT,0);
    // Get Y Gyro and ask for Z Gyro
  addhexchar(dataString, data&0x3FFF, 9+(1*5) +...
    (samplecounter*sample_size),4);
         // Save to text string
  data = SPItransfer2(XACCL_OUT,0);
    // Get Z Gyro and ask for X Acc
  addhexchar(dataString, data&0x3FFF, 9+(2*5) +...
    (samplecounter*sample_size),4);
          // Save to text string
  data = SPItransfer2(YACCL_OUT,0);
   // Get X Acc and ask for Y Acc
  addhexchar(dataString, data&0x3FFF, 9+(3*5) +...
    (samplecounter*sample_size),4);
         // Save to text string
  data = SPItransfer2(ZACCL_OUT,0);
    // Get Y Acc and ask for Z Acc
  addhexchar(dataString, data&0x3FFF, 9+(4*5) +...
    (samplecounter*sample_size),4);
         // Save to text string
  data = SPItransfer2(0,0);
    // Get Z Acc
  addhexchar(dataString, data&0x3FFF, 9+(5*5) +...
    (samplecounter*sample_size),4);
         // Save to text string
  addhexchar(dataString, analogReadFast(0)&0xFFF,...
    39+(0*4) + (samplecounter*sample_size),3);
         // Save analog P1 to text string
  addhexchar(dataString, analogReadFast(2)&0xFFF,...
    39+(1*4) + (samplecounter*sample_size),3);
         // Save analog P2 to text string
  // Pull chip select HIGH to delect
  bitSet(PORTD,chipSelectPin);
}
```

```
// Initializes software SPI bus for sensor
void SPIbegin(byte CSpin) {
  // Set correct pin modes
  pinMode(dataReadyPin, INPUT);
  pinMode(CSpin, OUTPUT);
  pinMode(mosiPin,OUTPUT);
  pinMode(misoPin,INPUT);
  pinMode(sclkPin,OUTPUT);
  //Clock polarity in mode 3 is high
  digitalWrite(sclkPin,HIGH);
  //Set chip select low to activate device
  digitalWrite(CSpin, HIGH);
}



// creates char[] from int in hex with digits chars
void addhexchar(volatile char* c, unsigned long n,...
 int start, int digits)
{
  byte nextchar;
  // Loop through the hex chars to make
  for(int ii=0; ii<digits ; ii++) {
    // Calculate what the next character should be
    nextchar = (n >> (digits-ii-1)*4) & 0xf;
    // Add it to the end of the string
    if(nextchar<=9) {
      c[start+ii] = '0' + nextchar;
    }
    else {
      c[start+ii] = 'A' + nextchar-10;
    }
  }
}



// Sends commands to set a given register
void setRegister(int reg, boolean highOrLow,...
 byte* whichBits, byte* setTo, byte n) {
  // Not sure why we need the delay, but we do...
  delay(500);
  // Read in data from register
  int regdata;
  if(highOrLow==HIGH) {
    regdata = highByte(readRegister(reg));
  }
  else {
    regdata = lowByte(readRegister(reg));
  }

  // Set or clear bits as needed
  byte working = regdata;
  for (int ii=0 ; ii<n ; ii++) {
    if(setTo[ii]==1) {
      bitSet(working,whichBits[ii]);
    }
    else {
      bitClear(working,whichBits[ii]);
    }
  }

  //Write to register
  if(working!=regdata) {
    writeRegister(reg+highOrLow,working);
  }
  delay(100);
}


// Initialize ADC with different prescaler
// to make it faster
void initADC(void) {
  // Set to A0 with Vcc as reference
  ADMUX = (1<<6)|0;
  // Set prescaler to 011 = division by 8
  ADCSRA &= 0b11111000;
  ADCSRA |= 0b011;
  // Begin first dummy conversion
  ADCSRA |= 0b11000000;
  // Wait for it to compelete (while flag is low)
  while(!(ADCSRA&(1<<ADIF)));
  // Clear flag (set it high)
  ADCSRA |= (1<<ADIF);
  // Ready to go!
}

// Read from ADC faster
```

```
unsigned int analogReadFast(int pin) {
  // Set mux with Vcc as reference
  ADMUX = (1<<6)|pin;
  // Begin conversion
  ADCSRA |= 0b01000000;
  // Wait for it to compelete (while flag is low)
  while(!(ADCSRA&(1<<ADIF)));
  // Clear flag (set it high)
  ADCSRA |= (1<<ADIF);
  // Return data
  return ADCL|(ADCH<<8);
}

// Clear sleep bit to make sure it doesnt fall
//  back asleep mid-run
void clear_sleep(void) {
  // Clear the Sleep Mode Control Register
  SMCR &= 0;
}

// Set sleep bits to enable standby mode, saving power
void go_to_sleep(void) {
  SMCR |= 0b01; // Enable sleep mode SE = 1
  SMCR |= 0b110 << 1; // Enter Standby Mode
}
```

Appendix C

PCB Design

C.1 Introduction

The design of the PCB was undertaken with a few general goals in mind.

1. Keep it as small as possible in a shape that could fit inside of a rock-like shell
2. Limit noise and crosstalk as much as possible
3. Create a layout that will allow for a user-friendly interface

In this appendix, we discuss the components used, the interfaces used between them, as well as how the design relates to these goals.

C.2 Powering

The components used required two separate voltages, $+5V$ and $+3.3V$. First a battery must be selected. A standard $9V$ battery has the required voltage and capacity to allow the Smart Rock to run for several hours, thus meeting the requirements. The battery ground and positive connections are connected to $POWER - 1$ and $POWER - 2$ as shown in figure C.1.

To provide the required voltages, the TLE4476 dual, low-drop voltage regulator is used. This takes the $9V$ and provides clean $5V$ and $3.3V$ supplies. Filtering capacitors are used at the input and both outputs to ensure the supplies are clean. Additionally, several filtering capacitors are used closer to the power input pins of other components to further clean the supplies.

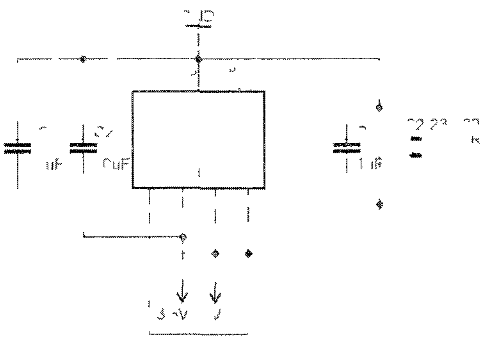


Figure C.1: PCB Powering

C.3 IMU Interface

With the proper power supplies provided, the components can now be connected together. The ADIS16367 IMU, shown on the left of figure C.2, is powered with $5V$ and has another filtering capacitor This interfaces with the Arduino Pro Mini microcontroller via an SPI bus, connected to pins 2 through 7 on the Arduino. Also shown here, are two LEDs to show the user the current status of the Smart Rock. These are driven with two GPIO pins from the microcontroller. The two resistors are used to limit the current through LEDs.

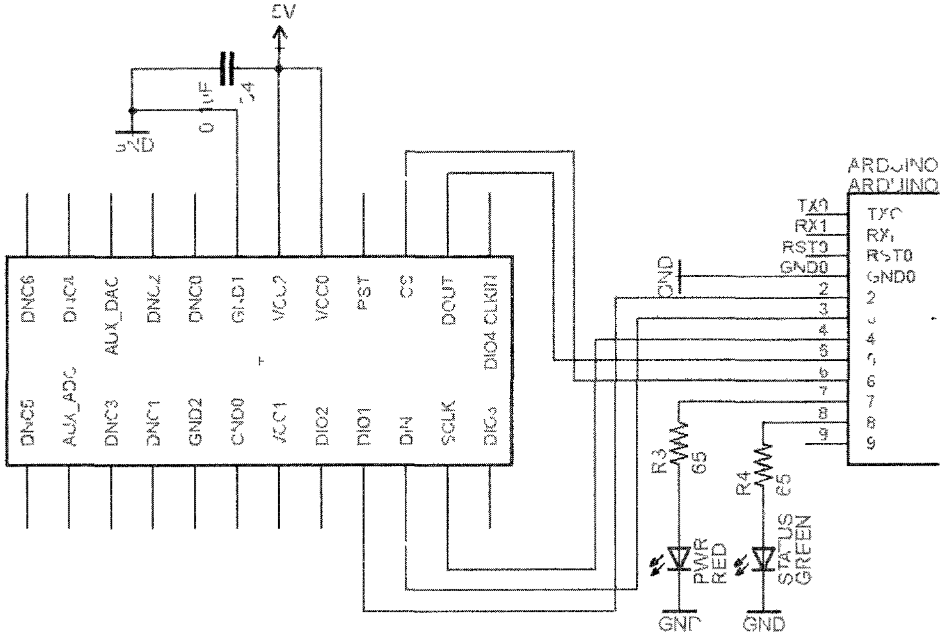


Figure C.2: IMU interface

C.4 Pressure Sensor Interface

In addition to the inertial sensors in the IMU, two pressure sensors are used, one of which is shown in figure C.3. These are also powered by the $5V$ supply. These 26PC05SMT sensors provide differential analog output on pins 2 and 4. For each sensor, these are fed into an AD620 instrumentation amplifier. The amplifier is also powered at $5V$, taking the differential output and providing a single ended output relative to ground. The gain of the amplifier is set by a simple resistor between two pins.

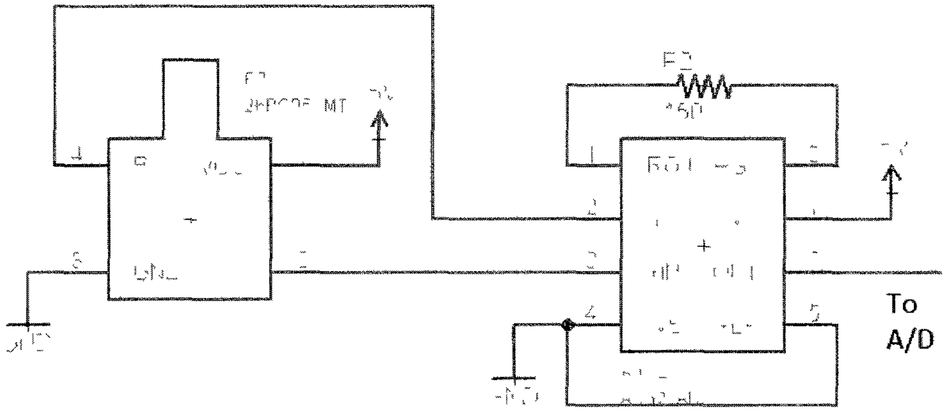


Figure C.3: Pressure sensor interface

C.5 microSD Interface

Now that the sensors can be read by the microcontroller, this data must be written to a μSD card. In figure C.4, the standard SPI bus is shown. This also shows that both the card and the microcontroller are powered with $3.3V$. It can be noted that there is a floating connection on the card socket. There are two standard communication methods with which μSD cards are compatible. These are with the SPI bus, used here, as well as with the SD bus, which uses one additional communication line.

C.6 PCB Layout

The board layout, designed in Eagle (Easily Applicable Graphical Layout Editor), is shown in figures C.5 and C.6. First, it can be noted that it is circular, in order to fit within a circular rock-like shell. The size was constrained by the battery and μSD card socket. These must both be accessible to the user and are positioned adjacent to one another on the top of the board. To the left of the card socket are the two status LEDs. On the top layer we see that the power comes in on the top right, and

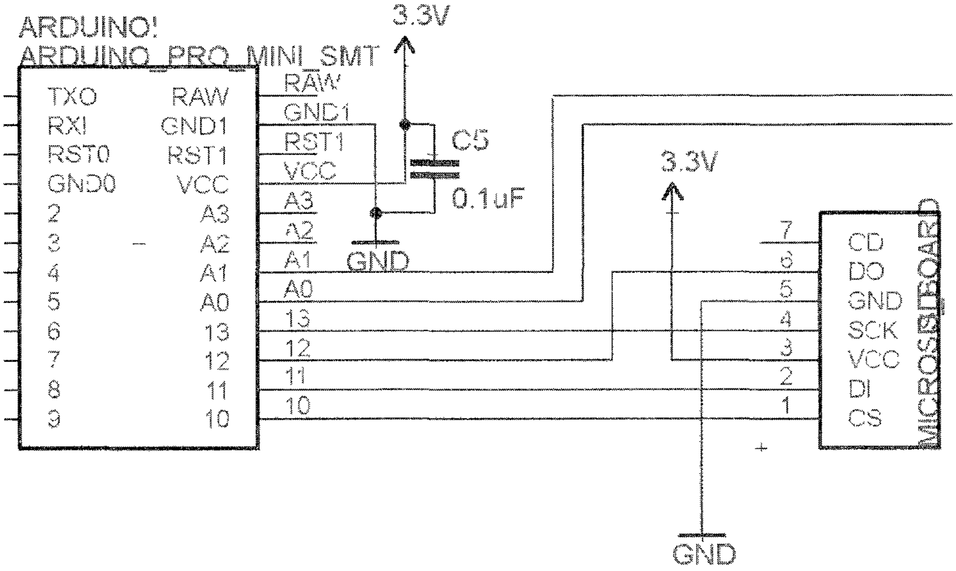


Figure C.4: μSD card interface

is fed to the regulator just next to it. Also included are the pressure sensors, facing outwards to interface with the outside fluid via the shell.

On the bottom layer, the IMU is shown with the header at the top of the board. It is held in place with two screws in the large diameter through holes. The only other components on the bottom of the board are the instrumentation amplifiers and their gain resistors. These are placed just below the pressure sensors.

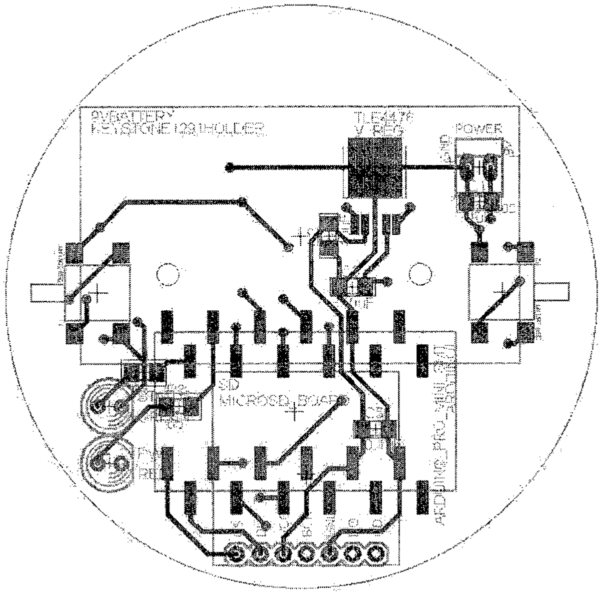


Figure C.5: PCB top layer

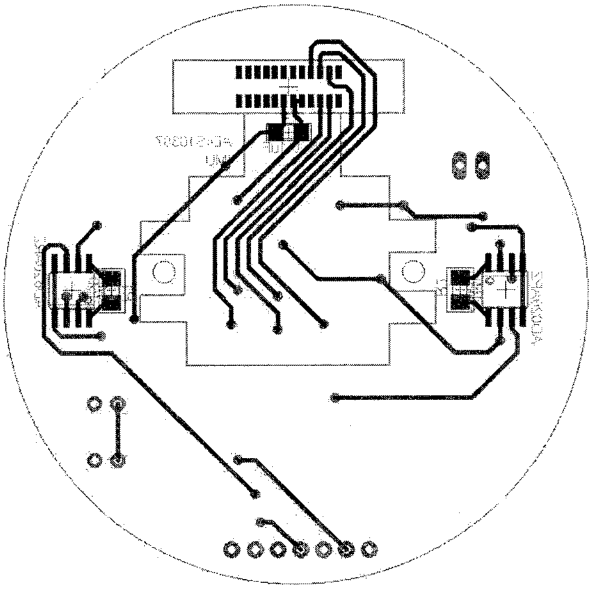


Figure C.6: PCB bottom layer

Appendix D

Shell Design

In this appendix, the design of the Smart Rock shell is discussed. This shell approximates the size and shape of the larger pieces of debris in the flow. The drawings are shown in figures D.1 and D.2.

The bottom half of the shell is a hemisphere with a small cylindrical extrusion. A cavity is created with a stepped bore hole. The PCB assembly is shaped such that the IMU electronics protrude downward. This fits in the smaller, deeper portion of this hole. The PCB then sits on the ledge, aligning the pressure sensors with their interfacing holes.

In the cylindrical section, several holes are drilled. On each side, there is one counterbored through hole, as well as smaller tapped holes beside each. The larger center hole is for the pressure interface. A tube can be inserted from the inside connecting to the pressure sensor. In the counterbore, a brass sinter piece filtration rating of 40 microns can be inserted to protect the sensor from debris while allowing pressure transmission. The tapped holes beside these are for set screws to hold in the brass sinter piece.

On the top there can also be seen a groove for an o-ring and a face with female threads. These features allow it to be assembled with the top section and seals the interface to IP68.

The top half of the shell is a simple cylinder with a filleted edge and a bored hole to hold the electronics and battery that will protrude from the bottom half of the shell. This also has a male threaded face for assembly.

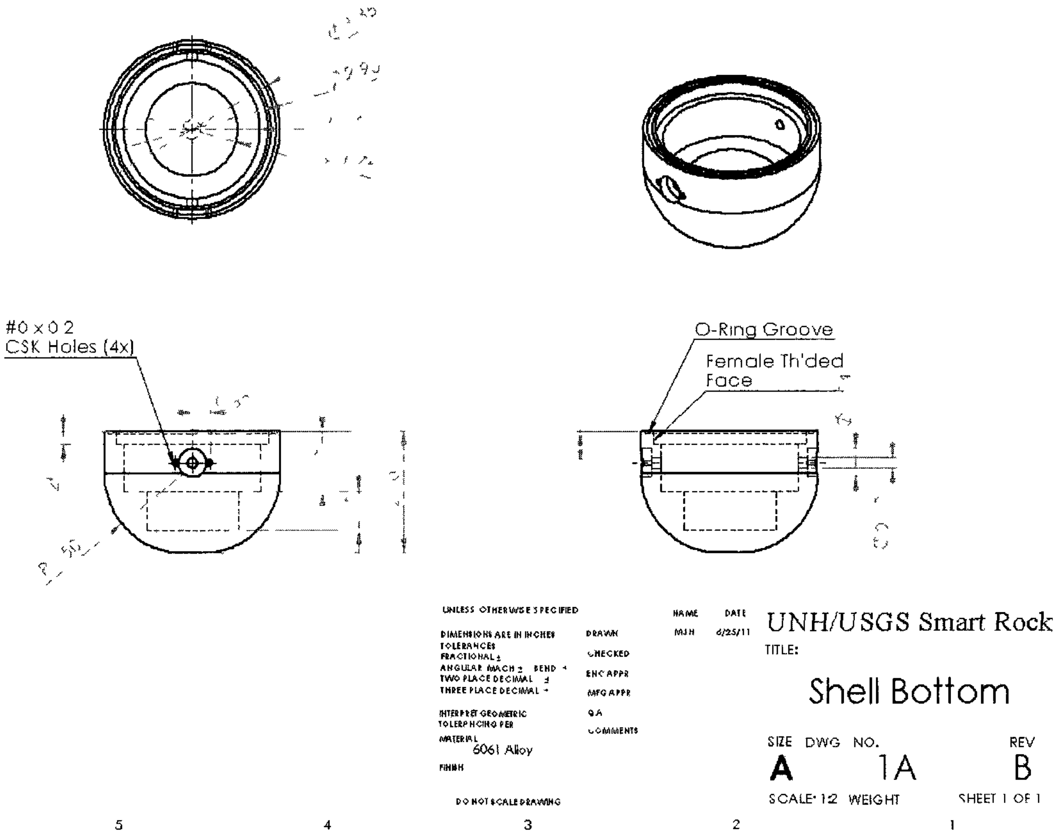


Figure D.1: Smart Rock shell, bottom half

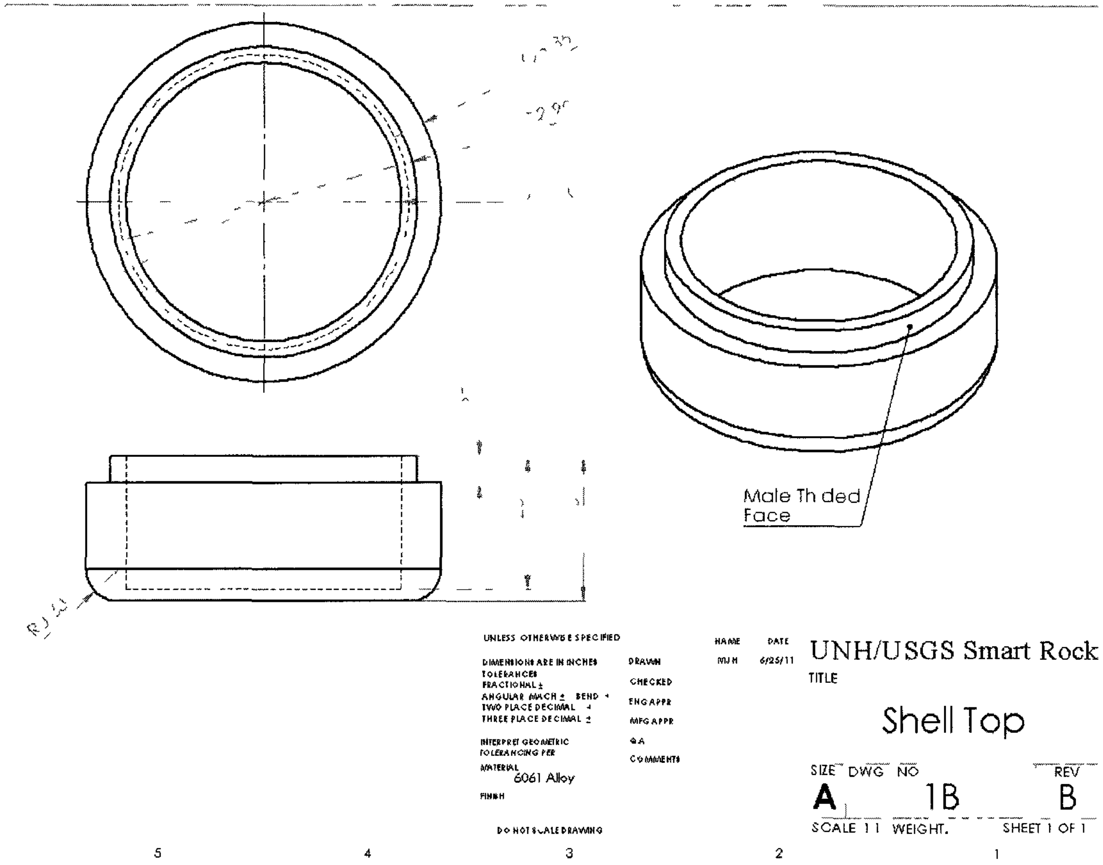


Figure D.2: Smart Rock shell, top half

Appendix E

User Manual

E.1 Introduction

In this appendix, a manual is given to specify how the Smart Rock functions, and how to ensure the most accurate data collection. This covers setup, recording data, extracting the data, and using this gathered information to calculate position.

E.2 Setup

Before using the Smart Rock there are a few preliminary steps. The pressure sensor interface must first be prepared. To do so, remove the set screws and sinter pieces. This reveals a cavity leading directly to the pressure sensor diaphragm. This cavity must be filled with water to eliminate added dynamics of having air in the liquid pressure system. This can be done using a syringe. At this time, the brass sinter piece must also be saturated with water. This also helps to eliminate air bubbles. Once the cavity has been filled and the sinter piece is saturated, both are placed underwater in the same container. The sinter piece can be placed in the counterbored hole and held in place with set screws. This process must be repeated for the second pressure sensor as well. Inserting these underwater further ensures no air is trapped in the system.

Also, the configuration file must be made before using the Smart Rock. This configuration file must be named `CONFIG.TXT` or `config.txt` and reside in the root directory of the μSD card. This simple text file contains only three numbers: delay time, sample rate, and sample duration. For example, if the configuration file contains the characters, `30 400 3600`, there will be a delay of 30 seconds, then the Rock will record data at a rate of 400 samples per second for 3600 seconds, or 1 hour.

E.3 Recording Data

To record data, make sure the μSD card is inserted and the battery is connected properly. Flip the switch, and recording will begin. The power LED will light to signify that it has turned on and will remain lit for the duration of the delay time. Note that even if the delay time is set to 0, there will be a delay of $\approx$ 20 seconds to set up the IMU, etc.

Once the delay is complete, the first LED will turn off and the second LED will light. When this happens, the Rock has begun recording data. This will continue for the specified duration. When recording is complete, both LEDs will light. At this point the power can be turned off and the card removed. If power is lost or the card becomes dislodged while recording, all data that has been logged up to that point will be saved and readable.

When placing the Smart Rock, the orientation must be noted. This will be used when analyzing the data. It may be convenient to align one of the sensor axes, such as the x axis, with the primary axis of motion. This will result in motion in the North direction in global coordinates.

E.4 Extracting Usable Data

The Smart Rock writes data to a text file with 9 columns of hexadecimal numbers. The first column is the time in milliseconds. After this are the x, y, and z gyroscope readings and the x, y, and z accelerometer readings. The final two columns are the pressure sensor readings. The first of these pressure readings is from the sensor on the negative y-axis, and the second reading is from the sensor on the positive y-axis.

At the bottom of the file are a few additional pieces of information. This includes the number of 14-sample blocks of data that were missed, the maximum write time for a block of data in microseconds, the sampling frequency, and the contents of the IMUs flag register. The write time can be used as a performance measure for a given μSD card. This time can vary with size, brand, file system, etc. For a $1GB$ Transcend, FAT16 formatted card, this time is typically 2800 μs. The flag register contents are described in Table 26 of the ADIS16367 data sheet [6]. This contains information about the self-test, supply voltage, communication failures, and other diagnostics. The additional information at the bottom of this file is removed automatically when running the program to read the data.

A MATLAB program has been written called `ReadIMU()` to take this text file and put the data in a usable form. This will create a second text file with the additional information removed as well as a MAT-file containing all the sensor data in engineering units. Alternatively, the sensor data can be converted with other calibrations or other methods if desired. Note that the gyro and accelerometer data are printed in the original text file in 14-bit twos complement format. The inertial data can be converted using the datasheet calibrations, and the pressure sensor readings can be converted using the calibration from section 3.2.3.

E.5 Analysis

With the extracted and usable data, it can now be used to calculate position. This is done using the MATLAB program `AdHocFilter()`. This program takes all of the inertial sensor data and the time vector and performs the filter that was described in chapter 6. This gives the calculated position from the forward filter and backward filter, allowing the user to average these in any desired method.

This is where the original orientation must be known. The inertial navigation process takes data in body coordinates and transforms it into local geographic coordinates. However, the Rock is not able to detect the directions of local geographic North and East, just down as this aligns with gravity. Thus, it takes the initial orientation, and rotates it such that the z axis aligns with the downward direction. The resulting orientations of the x and y axes are the North and East directions respectively. Because the filter takes as arguments known positions in local geographic coordinates, it must be known how the initial orientation relates to the local geographic orientation, and thus how the local geographic coordinates are oriented.

E.6 Example Procedure

To summarize, an example procedure is given here. This includes everything described above in a sequential manner.

1. Create appropriate configuration file on μSD card.
2. Ensure battery is properly inserted.
3. Insert μSD card into slot.
4. Flip power switch to "On". After a slight delay the power LED will illuminate.
5. Tightly close the shell. Hand tightening is typically enough to prevent water leakage.
6. Prepare pressure interface by filling cavity and saturating sinter pieces.
7. Place the Rock in the desired position, noting orientation.
8. The configuration file will be read and the Rock will delay the the prescribed amount of time. At this time, the first LED will turn off and the second LED will illuminate to indicate recording is in progress.
9. The Rock will record for a duration and at a sampling frequency as indicated in the configuration file.
10. When recording is complete, both LEDs will illuminate. At this time, it is safe to flip the power switch off.

11. Remove the μSD card and read data with ReadIMU() MATLAB function.

12. Use the newly created MAT-file to calculate position using the AdHocFilter() MATLAB function.

E.7 Advanced Options: Reprogramming

Next to the μSD socket, there is a 6-pin header. This header allows for reprogramming of the microcontroller. If it is desired, the firmware can be altered. This is done using the FTDI Basic 3.3V board from Sparkfun and a USB-A to USB-mini cable. This must be plugged into the Smart Rock with the top of the FTDI board facing the LEDs. Note that there is nothing preventing the insertion of the plug in the opposite orientation and that doing so may cause permanent damage to the microcontroller.

The appropriate software to upload new code to the Smart Rock can be downloaded for free at www.arduino.cc. With this, a "sketch" can be created and uploaded to the Smart Rock with a single click of a button. For instructions on how to use this IDE, please see the Arduino website.

Appendix F

Navigation Filter

For the interested reader, the full text of the filtering program is given here.

```
function [p, p2, v, v2, q, q2, N, M, t_set, p_set, k_set] = ...
    AdHocTestSmoother(omega, f_b, t, t_set, p_set, twoD, which_v_cor)



%{
Input arguments
    omega      = measure spin rate (3x1xN) [rad/s]
    f_b        = measured body force (3x1xN) [m/s^2]
    t          = time vector (Nx1) [s]
    t_set      = 1xM vector of times at which the position is known
    p_set      = 3xM vector of known positions (x,y,z are rows)
    twoD       = 1xM vector: 1 to restrict to 2D motion plane (in gravity
                     and downhill plane), 2 to restrict to no downward
                     motion, 0 to allow 3D

Output parameters
    p          = position (3x1xN) [m]
    v          = velocity (3x1xN) [m/s]
    q          = quaternion after compensation (4x1xN) []
    q_uncomp   = uncompensated quaternion (4x1xN) []
    g          = calculated gravity vector in local geographic coordinates
                 from average of first 100 points of f_b [ms/^2]
    C          = direction cosine matrix (DCM) attitude representation
    A          = transforms body axes to next time step
    f0         = initial gravity vector in body coordinates from first 100
```

```
                points of f_b [m/s^2]
    f_b_g     = acceleration [m/s^2] in local geographic coordinates
    sigma     = rotation about each axis [rad] over time interval
    dt        = time between data points [s]
    sigma_mag = magnitude of total rotation [rad] over time interval
    L         = latitude [rad]
%}

N=size(f_b,3);  % Number of points in set
if size(t,1)==1;t=t';end
dt = t(2:end)-t(1:end-1); % Create dt
fs = length(t)/max(t);
M = length(t_set);
t_set = [0,t_set];
p_set = [[0;0;0],p_set];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run the Whole Thing Forward           %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Omega = 2*pi/(23*3600+56*60+4.09053); %Rotation rate of earth (rad/s)
          %Period from:
          %http://imagine.gsfc.nasa.gov/docs/ask_astro/answers/970401c.html
f0 = mean(f_b(:,1,1:100),3);
g = [0 0 norm(f0)]';   %Gravity vector, local geographic coordinates (m/s)
            %http://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html
Ro = (6378135+6356750)/2; % (m)
    % https://visualization.hpc.mil/wiki/Radius_of_the_Earth

%% Initialize Variables
sigma = zeros(3,1,N);
q_uncomp = zeros(4,1,N);
q = zeros(4,1,N);
C = zeros(3,3,N);
sigma_mag = zeros(1,1,N);
sigmax = zeros(3,3,N);
A = zeros(3,3,N);
f_b_g = zeros(3,1,N);
L = zeros(1,1,N); L(1) = (43.08)*pi/180;
%Durham, NH 43.08 N Latitude http://www.bcca.org/misc/qiblih/latlong.html
v = zeros(3,1,N); v(:,1,1) = [0 0 0]';
%Velocity in local geographic coordinates (m/s) [v_N v_E v_h]'
p = zeros(3,1,N); p(:,1,1) = [0 0 0]';
%Position in local geographic coordinates (m) [x_N x_E h]'
omega_ie = zeros(3,1,N);
omega_iex = zeros(3,3,N);
```

```
omega_en = zeros(3,1,N);
omega_enx = zeros(3,3,N);
theta = zeros(N,1); % Angle between velocity vector and downhill
downhill = p_set(:,1)/norm(p_set(:,1));
transverse = cross([0;0;1],downhill)/...
    norm(cross([0;0;1],downhill));% What is the third direction
correct_factor = zeros(1,length(t_set));

%% Attitude Computations
% Set rotation rate initially to 0
omega = omega - repmat(mean(omega(:,1,1:100),3),[1,1,N]);
% Find initial quaternion from gravity vector
sigma0 = cross( [0;0;1] , f0 ) ...
    / norm( cross( [0;0;1] , f0 ) ) ...
    * acos( dot( [0;0;1] , f0/norm(f0) ) );
q(:,1,1) = vec2quat(sigma0);
    % Equivalent to q(:,1,1) = expm(skewsymW(sigma0)/2) * [1;0;0;0];
if isnan(mean(q(:,1,1)))
    q(:,1,1) = [1;0;0;0];
end
q_uncomp(:,1,1) = q(:,1,1);
sigma(:,:,1:end-1) = (-omega(:,:,1:end-1)-omega(:,:,2:end))/2.*...
    repmat(reshape(dt,1,1,N-1),3,1); %NOTE: negative was found late in
                                     % the process here, changes from
                                     % rotating local to body, to
                                     % rotating body to local.
%% Navigation Algorithm
v_corrections = 0;
a_corrections = 0;
marked = 0;
t_crit = .5; % [s] time to allow acceleration over a_crit
a_crit = 0.5*norm(g); % [m/s^2] max allowable acceleration
phi_crit = atan(a_crit/norm(g)); % [rad]
v_crit = 2; % [m/s] minimum velocity to attempt velocity filter
v_rot_per = 1.1; %Percentage of the way to rotate velocity vector
d_crit = 1.5; %[m] deviation from nominal at which correction is applied
theta_crit = 30*pi/180; % [rad]
k_crit = 20; %minimum steps to backtrack when deviation>d_crit
k_last = 1;
skip2 = false; % Skips 2nd stage if correction is made from 1st stage

%% Begin loop for each section with known endpoints
k_set = zeros(size(t_set));
for ii=1:length(t_set)
    k_set(ii) = find(t>=t_set(ii),1,'first');
```

```
end
for m = 2:M+1
    downhill = (p_set(:,m)-p_set(:,m-1))/...
        norm(p_set(:,m)-p_set(:,m-1));
    transverse = cross([0;0;1],downhill)/...
        norm(cross([0;0;1],downhill));% What is the third direction


k = k_set(m-1);
while(k<k_set(m))
    % Convert to DCM for use in calculating acceleration
    C(:,:,k) = quat2dcm(q(:,:,k));
    % Find magnitude of rotation
    sigma_mag(1,1,k) = sqrt(sum(sigma(:,:,k).^2,1));
    % Make sure it is not < eps
    sigma_mag(1,1,k) = sigma_mag(1,1,k) + eps*(sigma_mag(1,1,k)<1e-16);
    % Skew symmetric form of the sigma vector
    sigmax(:,:,k) = skewsym(sigma(:,:,k));
    % Find A matrix to update body coordinates
    A(:,:,k) = bodytimeupdate(sigma_mag(1,1,k),sigmax(:,:,k));
    % Calculate velocity change in global coordinates
    f_b_g(:,1,k) = C(:,:,k)*A(:,:,k)*f_b(:,1,k);
    if twoD(m-1)==1
        % What portion of f_b_g is in transverse direction?
        transverse_mag = dot(f_b_g(:,1,k), transverse);
        % Remove that transverse portion
        f_b_g(:,1,k) = f_b_g(:,1,k) - transverse_mag*transverse;
        % What portion of v is in transverse direction?
        transverse_mag = dot(v(:,1,k), transverse);
        % Remove that transverse portion
        v(:,1,k) = v(:,1,k) - transverse_mag*transverse;
    elseif twoD(m-1)==2
        % Remove portion in downward direction
        f_b_g(3,1,k) = g(3);
        % Set velocity to 0 in downward direction
        v(3,1,k) = 0;
    end
    % ???
    omega_ie(:,1,k)  = [Omega*cos(L(k));0;-Omega*sin(L(k))];
    % Skew symmetric form
    omega_iex(:,:,k) = skewsym(omega_ie(:,1,k));
    % ???
    omega_en(:,1,k)  = v(:,1,k).*[1/(Ro+p(3,1,k))
        -1/(Ro+p(3,1,k))
        -tan(L(1))/(Ro+p(3,1,k))];
    % Skew symmetric form
    omega_enx(:,:,k) = skewsym(omega_en(:,1,k));
    % Calculate next velocity vector
    v(:,1,k+1) = (eye(3) - 2*omega_iex(:,:,k)*dt(k)...
        - omega_enx(:,:,k)*dt(k))...
        * (v(:,1,k) + (f_b_g(:,1,k) - g)*dt(k));
    % Compute new latitude
    L(k+1) = L(k) + v(1,1,k)/(Ro+p(3,1,k))*dt(k);
    % Integrate to find position
    for ii=1:3
        p(ii,1,k+1) = p(ii,1,k) + trapz(t(k:k+1),v(ii,1,k:k+1));
    end

    %%%%%%%%% Calculations Complete %%%%%%%%%%%%%%%
    %%%%%%%%%    Begin Filter       %%%%%%%%%%%%%%%

  % If we are far enough into it
    if (k>fs*4)
%%%%%%%%%%%%% Filter Stage 1 %%%%%%%%%%%%%
    % Enter the filter if the velocity has at least this magnitude
        if ( norm(v(:,1,k))>v_crit && k>k_crit )
         % Find distance from nominal trajectory
    % (http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html)
         d_nom = norm(cross(p(:,:,k)-p_set(:,m-1),...
                p(:,:,k)-p_set(:,m)))...
                / norm(p_set(:,m-1)-p_set(:,m));
         % If we're more than d_crit meters away
            if (d_nom>d_crit)
%                 fprintf(['V_correct k = ',num2str(k),' -> ']);
         % Save the current location
         marked = k;
         % Calculate angle between velcity and downhill
        theta(k) = acos(dot(downhill,v(:,1,k)/norm(v(:,1,k))));
        % Backtrack to where theta<theta_crit
                while (theta(k) > theta_crit...
                        && norm(v(:,1,k))>v_crit...
                        && k>k_set(m-1))
         k = k-1;
         theta(k) = acos( dot( downhill,...
                        v(:,1,k)/norm(v(:,1,k)) ) );
                end
     % Make sure we went back far enough
                if (marked-k)<k_crit
     k = marked - k_crit;
                end
```

```
             % Make correction to quaternion
                if k<=k_last
                    k = k + k_crit;
                end
                % Save this spot to see if we go back there
                k_last = k;
                if which_v_cor==2
                    v(:,1,k) = correct_vel2(downhill,v(:,1,k));
                else
                    [q(:,1,k),v(:,1,k)] = correct_vel(q(:,1,k),...
                        downhill,v(:,1,k),v_rot_per);
                end
                v_corrections = v_corrections + 1;
%                 fprintf([num2str(k),'\n']);
             skip2 = true;

                % Re-do previous calculations
                C(:,:,k) = quat2dcm(q(:,:,k));
                % Find magnitude of rotation
                sigma_mag(1,1,k) = sqrt(sum(sigma(:,:,k).^2,1));
                % Make sure it is not < eps
                sigma_mag(1,1,k) = sigma_mag(1,1,k)...
                    + eps*(sigma_mag(1,1,k)<1e-16);
                % Skew symmetric form of the sigma vector
                sigmax(:,:,k) = skewsym(sigma(:,:,k));
                % Find A matrix to update body coordinates
                A(:,:,k) = bodytimeupdate(sigma_mag(1,1,k),sigmax(:,:,k));
                % Calculate velocity change in global coordinates
                f_b_g(:,1,k) = C(:,:,k)*A(:,:,k)*f_b(:,1,k);
                if twoD(m-1)==1
                    % What portion of f_b_g is in transverse direction?
                    transverse_mag = dot(f_b_g(:,1,k), transverse);
                    % Remove that transverse portion
                    f_b_g(:,1,k) = f_b_g(:,1,k)...
                        - transverse_mag*transverse;
                    % What portion of v is in transverse direction?
                    transverse_mag = dot(v(:,1,k), transverse);
                    % Remove that transverse portion
                    v(:,1,k) = v(:,1,k) - transverse_mag*transverse;
                elseif twoD(m-1)==2
                    % Remove portion in downward direction
                    f_b_g(3,1,k) = g(3);
                    % Set velocity to 0 in downward direction
                    v(3,1,k) = 0;
                end
```

```
        % ???
        omega_ie(:,1,k)  = [Omega*cos(L(k));0;-Omega*sin(L(k))];
        % Skew symmetric form
        omega_iex(:,:,k) = skewsym(omega_ie(:,1,k));
        % ???
        omega_en(:,1,k)  = v(:,1,k).*[1/(Ro+p(3,1,k))
            -1/(Ro+p(3,1,k))
            -tan(L(1))/(Ro+p(3,1,k))];
        % Skew symmetric form
        omega_enx(:,:,k) = skewsym(omega_en(:,1,k));
        % Calculate next velocity vector
        v(:,1,k+1) = (eye(3) - 2*omega_iex(:,:,k)*dt(k)...
            - omega_enx(:,:,k)*dt(k))...
            * (v(:,1,k) + (f_b_g(:,1,k) - g)*dt(k));
        % Compute new latitude
        L(k+1) = L(k) + v(1,1,k)/(Ro+p(3,1,k))*dt(k);
        % Integrate to find position
        for ii=1:3
            p(ii,1,k+1) = p(ii,1,k)...
                + trapz(t(k:k+1),v(ii,1,k:k+1));
        end
      else
        skip2 = false;
      end
    else
      skip2 = false;
    end

%%%%%%%%%% Filter Stage 2 %%%%%%%%%%%%
    if 0%skip2==false
      % Start a new counter from current spot
      kk=k;
      % Step back until the angle between the sensed acceleration
      %  and down is less than phi_crit
      while ( acos(dot(f_b_g(:,1,kk),g)...
            /norm(f_b_g(:,:,kk))/norm(g)) > phi_crit )
        kk=kk-1;
      end
      % If sensed angle > phi_crit for at least t_crit seconds
      if ( kk <= k-floor(fs*t_crit)+1 && kk>k_set(m-1) )
%                fprintf(['a_correct k = ',num2str(k),' -> ']);
        % Bring us back to that time
        k = kk;
        % If we went back too far
        if k<=k_last
```

```
                    k = k + k_crit;
                end
                k_last = k;
                % Make correction to quaternion
                q(:,1,k) = correct_acc(q(:,1,k),f_b_g(:,1,k), g);
                a_corrections = a_corrections + 1;
%                 fprintf([num2str(k),'\n']);

                % Re-do previous calculations
                C(:,:,k) = quat2dcm(q(:,:,k));
                % Find magnitude of rotation
                sigma_mag(1,1,k) = sqrt(sum(sigma(:,:,k).^2,1));
                % Make sure it is not < eps
                sigma_mag(1,1,k) = sigma_mag(1,1,k)...
                    + eps*(sigma_mag(1,1,k)<1e-16);
                % Skew symmetric form of the sigma vector
                sigmax(:,:,k) = skewsym(sigma(:,:,k));
                % Find A matrix to update body coordinates
                A(:,:,k) = bodytimeupdate(sigma_mag(1,1,k),sigmax(:,:,k));
                % Calculate velocity change in global coordinates
                f_b_g(:,1,k) = C(:,:,k)*A(:,:,k)*f_b(:,1,k);
                if twoD(m-1)==1
                    % What portion of f_b_g is in transverse direction?
                    transverse_mag = dot(f_b_g(:,1,k), transverse);
                    % Remove that transverse portion
                    f_b_g(:,1,k) = f_b_g(:,1,k)...
                        - transverse_mag*transverse;
                    % What portion of v is in transverse direction?
                    transverse_mag = dot(v(:,1,k), transverse);
                    % Remove that transverse portion
                    v(:,1,k) = v(:,1,k) - transverse_mag*transverse;
                elseif twoD(m-1)==2
                    % Remove portion in downward direction
                    f_b_g(3,1,k) = g(3);
                    % Set velocity to 0 in downward direction
                    v(3,1,k) = 0;
                end
                % ???
                omega_ie(:,1,k)  = [Omega*cos(L(k));0;-Omega*sin(L(k))];
                % Skew symmetric form
                omega_iex(:,:,k) = skewsym(omega_ie(:,1,k));
                % ???
                omega_en(:,1,k)  = v(:,1,k).*[1/(Ro+p(3,1,k))
                    -1/(Ro+p(3,1,k))
                    -tan(L(1))/(Ro+p(3,1,k))]';
                % Skew symmetric form
                omega_enx(:,:,k) = skewsym(omega_en(:,1,k));
                % Calculate next velocity vector
                v(:,1,k+1) = (eye(3) - 2*omega_iex(:,:,k)*dt(k)...
                    - omega_enx(:,:,k)*dt(k))...
                    * (v(:,1,k) + (f_b_g(:,1,k) - g)*dt(k));
                % Compute new latitude
                L(k+1) = L(k) + v(1,1,k)/(Ro+p(3,1,k))*dt(k);
                % Integrate to find position
                for ii=1:3
                    p(ii,1,k+1) = p(ii,1,k)...
                        + trapz(t(k:k+1),v(ii,1,k:k+1));
                end
            end
        end
    end

    %%%%%%%%%%%% Filter Complete %%%%%%%%%%%%%

    % Calculate new quaternion
    q_uncomp(:,1,k+1) = expm(skewsymW(sigma(:,1,k))/2) * q(:,1,k);
    % Normalize
    q(:,1,k+1) = q_uncomp(:,1,k+1)/norm(q_uncomp(:,1,k+1));
    % Increment counter, k
    k = k+1;
    if ~rem(k,1000); fprintf(['k = ',num2str(k),'\n']); end
end



%Finished calculating for the m'th section, normalize position, velocity
correct_factor(m-1) = 1/norm(p(:,:,k_set(m))-p(:,:,k_set(m-1)))...
    * norm(p_set(:,m)-p_set(:,m-1));
if dot(p(:,:,k_set(m)),downhill)<0
    correct_factor(m-1) = -correct_factor(m-1);
end

p(:,:,k_set(m-1):k_set(m)) = ( (p(:,:,k_set(m-1):k_set(m))...
    -repmat(p(:,:,k_set(m-1)),[1,1,k_set(m)-k_set(m-1)+1]))...
    * correct_factor(m-1) )...
    + repmat(p(:,:,k_set(m-1)),[1,1,k_set(m)-k_set(m-1)+1]);
v(:,:,k_set(m-1):k_set(m)) = ( (v(:,:,k_set(m-1):k_set(m))...
    -repmat(v(:,:,k_set(m-1)),[1,1,k_set(m)-k_set(m-1)+1]))...
    * correct_factor(m-1) )...
    + repmat(v(:,:,k_set(m-1)),[1,1,k_set(m)-k_set(m-1)+1]);
```

```
end

fprintf(['velocity corrections : ',num2str(v_corrections),'\n']);
fprintf(['acc corrections : ',num2str(a_corrections),'\n']);




%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Run the Whole Thing Backward          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
f02 = mean(f_b(:,1,end-100:end),3);
g2 = [0 0 norm(f02)]';%Gravity vector, local geographic coordinates (m/s)
          %http://nssdc.gsfc.nasa.gov/planetary/factsheet/earthfact.html
%% Initialize Variables
v_corrections = 0;
a_corrections = 0;
sigma2 = zeros(3,1,N);
q_uncomp2 = zeros(4,1,N);
q2 = zeros(4,1,N);
C2 = zeros(3,3,N);
sigma_mag2 = zeros(1,1,N);
sigmax2 = zeros(3,3,N);
A2 = zeros(3,3,N);
f_b_g2 = zeros(3,1,N);
L2 = zeros(1,1,N); L(1) = (43.08)*pi/180;
%Durham, NH 43.08 N Latitude http://www.bcca.org/misc/qiblih/latlong.html
v2 = zeros(3,1,N); v2(:,1,k_set(end)) = [0 0 0]';
    %Velocity in local geographic coordinates (m/s) [v_N v_E v_h]'
p2 = zeros(3,1,N); p2(:,1,k_set(end)) = p_set(:,end);
    %Position in local geographic coordinates (m) [x_N x_E h]'
omega_ie2 = zeros(3,1,N);
omega_iex2 = zeros(3,3,N);
omega_en2 = zeros(3,1,N);
omega_enx2 = zeros(3,3,N);
theta2 = zeros(N,1); % Angle between velocity vector and downhill
downhill2 = -(p_set(:,end)-p_set(:,end-1))...
    /norm(p_set(:,end)-p_set(:,end-1));
transverse2 = cross([0;0;1],downhill2)...
    /norm(cross([0;0;1],downhill2));% What is the third direction
correct_factor2 = zeros(1,length(t_set));
```

```
%% Attitude Computations
% Set rotation rate initially to 0
omega2 = omega - repmat(mean(omega(:,1,end-100:end),3),[1,1,N]);
% Find initial quaternion from gravity vector
sigma02 = cross( [0;0;1] , f0 ) ...
    / norm( cross( [0;0;1] , f0 ) ) ...
    * acos( dot( [0;0;1] , f0/norm(f0) ) );
q2(:,1,k_set(end)) = vec2quat(sigma02);
    % Equivalent to q(:,1,1) = expm(skewsymW(sigma0)/2) * [1;0;0;0];
if isnan(mean(q2(:,1,k_set(end))))
    q2(:,1,k_set(end)) = [1;0;0;0];
end
q_uncomp2(:,1,end) = q2(:,1,end);
sigma2(:,:,2:end) = -(-omega2(:,:,1:end-1)-omega2(:,:,2:end))/2.*...
    repmat(reshape(dt,1,1,N-1),3,1); %NOTE: negative was found late in
                                     % the process here, changes from
                                     % rotating local to body, to
                                     % rotating body to local.

%% Begin loop for each section with known endpoints
for m = M:-1:1
    downhill2 = -(p_set(:,m+1)-p_set(:,m))...
        /norm(p_set(:,m+1)-p_set(:,m));
    transverse2 = cross([0;0;1],downhill2)...
        /norm(cross([0;0;1],downhill2));% What is the third direction


k = k_set(m+1);
while(k>k_set(m))
    % Convert to DCM for use in calculating acceleration
    C2(:,:,k) = quat2dcm(q2(:,:,k));
    % Find magnitude of rotation
    sigma_mag2(1,1,k) = sqrt(sum(sigma2(:,:,k).^2,1));
    % Make sure it is not < eps
    sigma_mag2(1,1,k) = sigma_mag2(1,1,k)...
        + eps*(sigma_mag2(1,1,k)<1e-16);
    % Skew symmetric form of the sigma vector
    sigmax2(:,:,k) = skewsym(sigma2(:,:,k));
    % Find A matrix to update body coordinates
    A2(:,:,k) = bodytimeupdate(sigma_mag2(1,1,k),sigmax2(:,:,k));
    % Calculate velocity change in global coordinates
    f_b_g2(:,1,k) = - C2(:,:,k)*A2(:,:,k)*f_b(:,1,k);
    if twoD(m)==1
        % What portion of f_b_g is in transverse direction?
        transverse_mag2 = dot(f_b_g2(:,1,k), transverse2);
        % Remove that transverse portion
        f_b_g2(:,1,k) = f_b_g2(:,1,k) - transverse_mag2*transverse2;
        % What portion of v is in transverse direction?
        transverse_mag2 = dot(v2(:,1,k), transverse2);
        % Remove that transverse portion
        v2(:,1,k) = v2(:,1,k) - transverse_mag2*transverse2;
    elseif twoD(m)==2
        % Remove portion in downward direction
        f_b_g2(3,1,k) = g2(3);
        % Set velocity to 0 in downward direction
        v2(3,1,k) = 0;
    end
    % ???
    omega_ie2(:,1,k)  = [Omega*cos(L2(k));0;-Omega*sin(L2(k))];
    % Skew symmetric form
    omega_iex2(:,:,k) = skewsym(omega_ie2(:,1,k));
    % ???
    omega_en2(:,1,k)  = v2(:,1,k).*[1/(Ro+p2(3,1,k))
        -1/(Ro+p2(3,1,k))
        -tan(L2(1))/(Ro+p2(3,1,k))]';
    % Skew symmetric form
    omega_enx2(:,:,k) = skewsym(omega_en2(:,1,k));
    % Calculate next velocity vector
    v2(:,1,k-1) = (eye(3) - 2*omega_iex2(:,:,k)*dt(k)...
        - omega_enx2(:,:,k)*dt(k))...
        * (v2(:,1,k) + (f_b_g2(:,1,k) - g2)*dt(k));
    % Compute new latitude
    L2(k-1) = L2(k) + v2(1,1,k)/(Ro+p2(3,1,k))*dt(k);
    % Integrate to find position
    for ii=1:3
        p2(ii,1,k-1) = p2(ii,1,k) - trapz(t(k-1:k),v2(ii,1,k-1:k));
    end

    %%%%%%%%% Calculations Complete %%%%%%%%%%%%%%%
    %%%%%%%%%    Begin Filter       %%%%%%%%%%%%%%%

  % If we are far enough into it
    if (k< k_set(end)-fs*2)
%%%%%%%%%%%%% Filter Stage 1 %%%%%%%%%%%%%
    % Enter the filter if the velocity has at least this magnitude
        if ( norm(v2(:,1,k))>v_crit && k>k_crit )
         % Find distance from nominal trajectory
   % (http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html)
         d_nom2 = norm(cross(p2(:,:,k)-p_set(:,m+1)...
                ,p2(:,:,k)+p_set(:,m)))...
                / norm(p_set(:,m+1)-p_set(:,m));
         % If we're more than d_crit meters away
            if (d_nom2>d_crit)
%                 fprintf(['V_correct k = ',num2str(k),' -> ']);
         % Save the current location
         marked = k;
         % Calculate angle between velcity and downhill
        theta2(k) = acos( dot( downhill2,...
                    v2(:,1,k)/norm(v2(:,1,k)) ) );
        % Backtrack to where theta<theta_crit
                while (theta2(k) > theta_crit...
                        && norm(v2(:,1,k))>v_crit...
                        && k<k_set(m))
         k = k+1;
         theta2(k) = acos( dot( downhill2,...
                        v2(:,1,k)/norm(v2(:,1,k)) ) );
                end
     % Make sure we went back far enough
                if (k-marked)<k_crit
     k = marked + k_crit;
                end
             % Make correction to quaternion
                if k>=k_last
                    k = k - k_crit;
                end
                % Save this spot to see if we go back there
                k_last = k;
                if which_v_cor==2
                    v2(:,1,k) = correct_vel2(downhill2,v2(:,1,k));
                else
                    [q2(:,1,k),v2(:,1,k)] = correct_vel(q2(:,1,k),...
                        downhill2,v2(:,1,k),v_rot_per);
                end
                v_corrections = v_corrections + 1;
%                 fprintf([num2str(k),'\n']);
             skip2 = true;

                % Re-do previous calculations
                C2(:,:,k) = quat2dcm(q2(:,:,k));
                % Find magnitude of rotation
                sigma_mag2(1,1,k) = sqrt(sum(sigma2(:,:,k).^2,1));
                % Make sure it is not < eps
                sigma_mag2(1,1,k) = sigma_mag2(1,1,k)...
                    + eps*(sigma_mag2(1,1,k)<1e-16);
                % Skew symmetric form of the sigma vector
```

```
sigmax2(:,:,k) = skewsym(sigma2(:,:,k));
% Find A matrix to update body coordinates
A2(:,:,k) = bodytimeupdate(sigma_mag2(1,1,k)...
    ,sigmax2(:,:,k));
% Calculate velocity change in global coordinates
f_b_g2(:,1,k) = C2(:,:,k)*A2(:,:,k)*f_b(:,1,k);
if twoD(m)==1
    % What portion of f_b_g is in transverse direction?
    transverse_mag2 = dot(f_b_g2(:,1,k), transverse2);
    % Remove that transverse portion
    f_b_g2(:,1,k) = f_b_g2(:,1,k)...
        - transverse_mag2*transverse2;
    % What portion of v is in transverse direction?
    transverse_mag2 = dot(v2(:,1,k), transverse2);
    % Remove that transverse portion
    v2(:,1,k) = v2(:,1,k) - transverse_mag2*transverse2;
elseif twoD(m)==2
    % Remove portion in downward direction
    f_b_g2(3,1,k) = g2(3);
    % Set velocity to 0 in downward direction
    v2(3,1,k) = 0;
end
% ???
omega_ie2(:,1,k)  = [Omega*cos(L2(k))
    0
    -Omega*sin(L2(k))];
% Skew symmetric form
omega_iex2(:,:,k) = skewsym(omega_ie2(:,1,k));
% ???
omega_en2(:,1,k)  = v2(:,1,k).*[1/(Ro+p2(3,1,k))
    -1/(Ro+p2(3,1,k))
    -tan(L2(1))/(Ro+p2(3,1,k))];
% Skew symmetric form
omega_enx2(:,:,k) = skewsym(omega_en2(:,1,k));
% Calculate next velocity vector
v2(:,1,k-1) = (eye(3) - 2*omega_iex2(:,:,k)*dt(k)...
    - omega_enx2(:,:,k)*dt(k))...
    * (v2(:,1,k) + (f_b_g2(:,1,k) - g2)*dt(k));
% Compute new latitude
L2(k-1) = L2(k) + v2(1,1,k)/(Ro+p2(3,1,k))*dt(k);
% Integrate to find position
for ii=1:3
    p2(ii,1,k+1) = p2(ii,1,k)...
        - trapz(t(k-1:k),v2(ii,1,k-1:k));
end
```

```
            else
                skip2 = false;
            end
        else
            skip2 = false;
        end

%%%%%%%%%%% Filter Stage 2 %%%%%%%%%%%%%
        if 0%skip2==false
            % Start a new counter from current spot
            kk=k;
            % Step back until the angle between the sensed acceleration
            %   and down is less than phi_crit
            while ( acos(dot(f_b_g(:,1,kk),g)/...
                    norm(f_b_g(:,:,kk))/norm(g)) > phi_crit )
                kk=kk-1;
            end
            % If sensed angle > phi_crit for at least t_crit seconds
            if ( kk <= k-floor(fs*t_crit)+1 && kk>k_set(m-1) )
%                 fprintf(['a_correct k = ',num2str(k),' -> ']);
                % Bring us back to that time
                k = kk;
                % If we went back too far
                if k<=k_last
                    k = k + k_crit;
                end
                k_last = k;
                % Make correction to quaternion
                q(:,1,k) = correct_acc(q(:,1,k),f_b_g(:,1,k), g);
                a_corrections = a_corrections + 1;
%                 fprintf([num2str(k),'\n']);

                % Re-do previous calculations
                C(:,:,k) = quat2dcm(q(:,:,k));
                % Find magnitude of rotation
                sigma_mag(1,1,k) = sqrt(sum(sigma(:,:,k).^2,1));
                % Make sure it is not < eps
                sigma_mag(1,1,k) = sigma_mag(1,1,k)...
                    + eps*(sigma_mag(1,1,k)<1e-16);
                % Skew symmetric form of the sigma vector
                sigmax(:,:,k) = skewsym(sigma(:,:,k));
                % Find A matrix to update body coordinates
                A(:,:,k) = bodytimeupdate(sigma_mag(1,1,k),sigmax(:,:,k));
                % Calculate velocity change in global coordinates
                f_b_g(:,1,k) = C(:,:,k)*A(:,:,k)*f_b(:,1,k);
            if twoD(m)==1
                % What portion of f_b_g is in transverse direction?
                transverse_mag = dot(f_b_g(:,1,k), transverse);
                % Remove that transverse portion
                f_b_g(:,1,k) = f_b_g(:,1,k)...
                    - transverse_mag*transverse;
                % What portion of v is in transverse direction?
                transverse_mag = dot(v(:,1,k), transverse);
                % Remove that transverse portion
                v(:,1,k) = v(:,1,k) - transverse_mag*transverse;
            elseif twoD(m)==2
                % Remove portion in downward direction
                f_b_g(3,1,k) = g(3);
                % Set velocity to 0 in downward direction
                v(3,1,k) = 0;
            end
            % ???
            omega_ie(:,1,k)  = [Omega*cos(L(k));0;-Omega*sin(L(k))];
            % Skew symmetric form
            omega_iex(:,:,k) = skewsym(omega_ie(:,1,k));
            % ???
            omega_en(:,1,k)  = v(:,1,k).*[1/(Ro+p(3,1,k))
                -1/(Ro+p(3,1,k))
                -tan(L(1))/(Ro+p(3,1,k))]';
            % Skew symmetric form
            omega_enx(:,:,k) = skewsym(omega_en(:,1,k));
            % Calculate next velocity vector
            v(:,1,k+1) = (eye(3) - 2*omega_iex(:,:,k)*dt(k)...
                - omega_enx(:,:,k)*dt(k))...
                * (v(:,1,k) + (f_b_g(:,1,k) - g)*dt(k));
            % Compute new latitude
            L(k+1) = L(k) + v(1,1,k)/(Ro+p(3,1,k))*dt(k);
            % Integrate to find position
            for ii=1:3
                p(ii,1,k+1) = p(ii,1,k)...
                    - trapz(t(k:k+1),v(ii,1,k:k+1));
            end
        end
    end
end

%%%%%%%%%%%% Filter Complete %%%%%%%%%%%%%

% Calculate new quaternion
q_uncomp2(:,1,k-1) = expm(skewsymW(sigma2(:,1,k))/2) * q2(:,1,k);
```

```
    % Normalize
    q2(:,1,k-1) = q_uncomp2(:,1,k-1)/norm(q_uncomp2(:,1,k-1));
    % Increment counter, k
    k = k-1;
    if ~rem(k,1000); fprintf(['k = ',num2str(k),'\n']); end
end



%Finished calculating for the m'th section, normalize position, velocity
correct_factor2(m) = 1/norm(p2(:,:,k_set(m+1))-p2(:,:,k_set(m)))...
    * norm(p_set(:,m+1)-p_set(:,m));
if dot(p2(:,:,k_set(m))-p2(:,:,k_set(m+1)),downhill2)<0
    correct_factor2(m) = -correct_factor2(m);
end

p2(:,:,k_set(m):k_set(m+1)) = ( (p2(:,:,k_set(m):k_set(m+1))...
    -repmat(p2(:,:,k_set(m+1)),[1,1,k_set(m+1)-k_set(m)+1]))...
    * correct_factor2(m) )...
    + repmat(p2(:,:,k_set(m+1)),[1,1,k_set(m+1)-k_set(m)+1]);
v2(:,:,k_set(m):k_set(m+1)) = ( (v2(:,:,k_set(m):k_set(m+1))...
    -repmat(v2(:,:,k_set(m+1)),[1,1,k_set(m+1)-k_set(m)+1]))...
    * correct_factor(m) )...
    + repmat(v2(:,:,k_set(m+1)),[1,1,k_set(m+1)-k_set(m)+1]);


end




fprintf(['velocity corrections : ',num2str(v_corrections),'\n']);
fprintf(['acc corrections : ',num2str(a_corrections),'\n']);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%




function A = bodytimeupdate(sigma_mag, sigmax)
% Computes matrix to update body coordinates over time interval
% sigma_mag = vector of magnitudes of sigma vectors
% sigmax    = vector of skew symetric matrix forms of sigma vectors
```

```
A = repmat(eye(3),[1,1,size(sigmax,3)]) + ...
    repmat(sin(sigma_mag)./sigma_mag,3,3).*sigmax + ...
    repmat((1-cos(sigma_mag))./sigma_mag,3,3).*squared(sigmax);
```

```
function [q, v] = correct_vel(quat, downhill, velocity, percent)
% Find unit vector about which to rotate
rot_vec = cross(downhill, velocity)/norm(cross(downhill, velocity));
% Rotate percent% of the way towards downhill
rot_vec = rot_vec*percent*acos(dot( downhill,...
        velocity/norm(velocity) ));
% Calculate new quaternion
q = expm(skewsymW(rot_vec/2)) * quat;
    % Calculate new rotated velocity
    v = quat2dcm(vec2quat(rot_vec)) * velocity;
```

```
function v = correct_vel2(downhill, velocity)
    % Normalize downhill just in case
    downhill = downhill/norm(downhill);
    % Find magnitude of v in downhill direction
    d_mag = dot(velocity,downhill);
    % Make velocity just in the downhill direction
    v = d_mag*downhill;
```

```
function C=quat2dcm(q)
q0 = q(1,:,:);
q1 = q(2,:,:);
q2 = q(3,:,:);
q3 = q(4,:,:);
% http://www.mathworks.com/help/toolbox/aeroblks/
% directioncosinematrixtoquaternions.html
C = [q0.^2+q1.^2-q2.^2-q3.^2, 2*(q1.*q2+q0.*q3), 2*(q1.*q3-q0.*q2)
    2*(q1.*q2-q0.*q3), q0.^2-q1.^2+q2.^2-q3.^2, 2*(q2.*q3+q0.*q1)
    2*(q1.*q3+q0.*q2), 2*(q2.*q3-q0.*q1), q0.^2-q1.^2-q2.^2+q3.^2];
```

```
function mx = skewsym( m )
```

```
%This function takes a 3 x 1 x N matrix and creates the skew symmetric
%matrix 3 x 3 x N

mx=zeros(size(m,1), size(m,1), size(m,3));
mx(2,1,:) = m(3,1,:);
mx(1,2,:) = -m(3,1,:);

mx(3,1,:) = -m(2,1,:);
mx(1,3,:) = m(2,1,:);

mx(3,2,:) = m(1,1,:);
mx(2,3,:) = -m(1,1,:);
```

```
function mx = skewsymW( m )

% This function takes a 3 x 1 x N matrix and creates the skew symmetric
% matrix 4 x 4 x N

mx=zeros(size(m,1), size(m,1), size(m,3));
mx(2,1,:) = m(1,1,:);
mx(1,2,:) = -m(1,1,:);

mx(3,1,:) = m(2,1,:);
mx(1,3,:) = -m(2,1,:);

mx(4,1,:) = m(3,1,:);
mx(1,4,:) = -m(3,1,:);

mx(3,2,:) = -m(3,1,:);
mx(2,3,:) = m(3,1,:);

mx(4,2,:) = m(2,1,:);
mx(2,4,:) = -m(2,1,:);

mx(4,3,:) = -m(1,1,:);
mx(3,4,:) = m(1,1,:);
```

```
function m2 = squared( m )

m2=zeros(size(m));
for ii=1:size(m,3)
    m2(:,:,ii) = m(:,:,ii)^2;
end
```

```
function q=vec2quat(v)

q = [cos(norm(v)/2)
    v(1)/norm(v)*sin(norm(v)/2)
    v(2)/norm(v)*sin(norm(v)/2)
    v(3)/norm(v)*sin(norm(v)/2)];
```

Appendix G

Data Sheets

This appendix serves to provide supplementary information regarding the instrumentation and electronic components used in creation of the Smart Rock. Attached are the data sheets for each of the sensors and components used.

Six Degrees of Freedom Inertial Sensor

ADIS16367

FEATURES

Tri-axis digital gyroscope with digital range scaling
- **±300°/sec, ±600°/sec, ±1200°/sec settings**
- **Tight orthogonal alignment: 0.05°**

Tri-axis digital accelerometer: ±18 *g*

Autonomous operation and data collection
- **No external configuration commands required**
- **Start-up time: 180 ms**
- **Sleep mode recovery time: 4 ms**

Factory-calibrated sensitivity, bias, and axial alignment
- **Calibration temperature range: −40°C to +85°C**

SPI-compatible serial interface

Wide bandwidth: 330 Hz

Embedded temperature sensor

Programmable operation and control
- **Automatic and manual bias correction controls**
- **Bartlett window, FIR filter length, number of taps**
- **Digital I/O: data ready, alarm indicator, general-purpose**
- **Alarms for condition monitoring**
- **Sleep mode for power management**
- **DAC output voltage**
- **Enable external sample clock input: up to 1.2 kHz**
- **Single-command self-test**

Single-supply operation: 4.75 V to 5.25 V

2000 *g* shock survivability

Operating temperature range: −40°C to +105°C

APPLICATIONS

Medical instrumentation

Robotics

Platform controls

Navigation

GENERAL DESCRIPTION

The ADIS16367 *i*Sensor® is a complete inertial system that includes a tri-axis gyroscope and tri-axis accelerometer. Each sensor in the ADIS16367 combines industry-leading *i*MEMS® technology with signal conditioning that optimizes dynamic performance. The factory calibration characterizes each sensor for sensitivity, bias, alignment, and linear acceleration (gyro bias). As a result, each sensor has its own dynamic compensation formulas that provide accurate sensor measurements over a temperature range of −40°C to +85°C.

The ADIS16367 provides a simple, cost-effective method for integrating accurate, multiaxis inertial sensing into industrial systems, especially when compared with the complexity and investment associated with discrete designs. All necessary motion testing and calibration are part of the production process at the factory, greatly reducing system integration time. Tight orthogonal alignment simplifies inertial frame alignment in navigation systems. An improved SPI interface and register structure provide faster data collection and configuration control.

The ADIS16367 uses a compatible pinout and the same package as the ADIS1635x family. Therefore, systems that currently use the ADIS1635x family can upgrade their performance with minor firmware adjustments in their processor designs.

This compact module is approximately 23 mm × 23 mm × 23 mm and provides a flexible connector interface that enables multiple mounting orientation options.

FUNCTIONAL BLOCK DIAGRAM

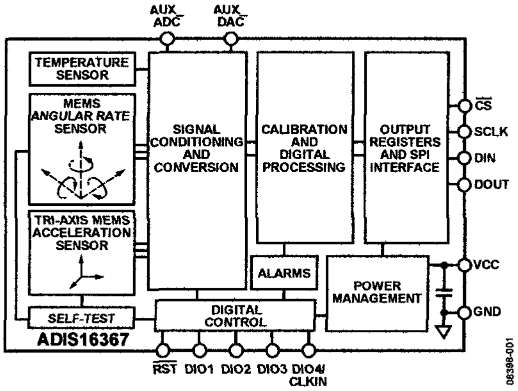


Figure 1.

Rev. A

Information furnished by Analog Devices is believed to be accurate and reliable. However, no responsibility is assumed by Analog Devices for its use, nor for any infringements of patents or other rights of third parties that may result from its use. Specifications subject to change without notice. No license is granted by implication or otherwise under any patent or patent rights of Analog Devices. Trademarks and registered trademarks are the property of their respective owners.

One Technology Way, P.O. Box 9106, Norwood, MA 02062-9106, U.S.A.
Tel: 781.329.4700
Fax: 781.461.3113

SPECIFICATIONS

T_A = 25°C, VCC = 5.0 V, angular rate = 0°/sec, dynamic range = ±300°/sec ± 1 *g*, unless otherwise noted.

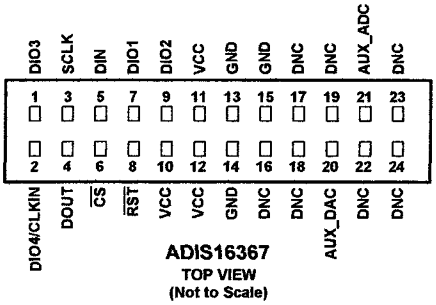
Table 1.

Parameter	Test Conditions/Comments	Min	Typ	Max	Unit
GYROSCOPES					
Dynamic Range		±1200	±1400		°/sec
Initial Sensitivity	Dynamic range = ±1200°/sec	0.198	0.2	0.202	°/sec/LSB
	Dynamic range = ±600°/sec		0.1		°/sec/LSB
	Dynamic range = ±300°/sec		0.05		°/sec/LSB
Sensitivity Temperature Coefficient	−40°C ≤ T_A ≤ +85°C		±40		ppm/°C
Misalignment	Axis-to-axis		±0.05		Degrees
	Axis-to-frame (package)		±0.5		Degrees
Nonlinearity	Best-fit straight line		±0.1		% of FS
Initial Bias Error	±1 σ		±3		°/sec
In-Run Bias Stability	1 σ, SMPL_PRD = 0x0001		0.013		°/sec
Angular Random Walk	1 σ, SMPL_PRD = 0x0001		2.0		°/√hr
Bias Temperature Coefficient	−40°C ≤ T_A ≤ +85°C		±0.01		°/sec/°C
Linear Acceleration Effect on Bias	Any axis, 1 σ (MSC_CTRL[7] = 1)		0.075		°/sec/*g*
Bias Voltage Sensitivity	VCC = 4.75 V to 5.25 V		±0.3		°/sec/V
Output Noise	±1200°/sec range, no filtering		0.8		°/sec rms
Rate Noise Density	f = 25 Hz, ±1200°/sec range, no filtering		0.044		°/sec/√Hz rms
3 dB Bandwidth			330		Hz
Sensor Resonant Frequency			14.5		kHz
Self-Test Change in Output Response	±1200°/sec range setting	±170	±350	±625	LSB
ACCELEROMETERS	Each axis				
Dynamic Range		±18			*g*
Initial Sensitivity		3.285	3.33	3.38	m*g*/LSB
Sensitivity Temperature Coefficient	−40°C ≤ T_A ≤ +85°C		±50		ppm/°C
Misalignment	Axis-to-axis		0.2		Degrees
	Axis-to-frame (package)		±0.5		Degrees
Nonlinearity	Best-fit straight line		0.1		% of FS
Initial Bias Error	±1 σ		±50		m*g*
In-Run Bias Stability	1 σ		0.2		m*g*
Velocity Random Walk	1 σ		0.2		m/sec/√hr
Bias Temperature Coefficient	−40°C ≤ T_A ≤ +85°C		±0.3		m*g*/°C
Bias Voltage Sensitivity	VCC = 4.75 V to 5.25 V		2.5		m*g*/V
Output Noise	No filtering		9		m*g* rms
Noise Density	No filtering		0.5		m*g*/√Hz rms
3 dB Bandwidth			330		Hz
Sensor Resonant Frequency			5.5		kHz
Self-Test Change in Output Response	X-axis and y-axis	59		151	LSB
TEMPERATURE SENSOR					
Scale Factor	Output = 0x0000 at 25°C (±5°C)		0.136		°C/LSB
ADC INPUT					
Resolution			12		Bits
Integral Nonlinearity			±2		LSB
Differential Nonlinearity			±1		LSB
Offset Error			±4		LSB
Gain Error			±2		LSB
Input Range		0		3.3	V
Input Capacitance	During acquisition		20		pF

Parameter	Test Conditions/Comments	Min	Typ	Max	Unit
DAC OUTPUT	5 kΩ/100 pF to GND				
Resolution			12		Bits
Relative Accuracy	101 LSB ≤ input code ≤ 4095 LSB		±4		LSB
Differential Nonlinearity			±1		LSB
Offset Error			±5		mV
Gain Error			±0.5		%
Output Range		0		3.3	V
Output Impedance			2		Ω
Output Settling Time			10		μs
LOGIC INPUTS[1]					
Input High Voltage, V_{IH}		2.0			V
Input Low Voltage, V_{IL}				0.8	V
	$\overline{\text{CS}}$ signal to wake up from sleep mode			0.55	V
$\overline{\text{CS}}$ Wake-Up Pulse Width		20			μs
Logic 1 Input Current, I_{IH}	$V_{IH} = 3.3$ V		±0.2	±10	μA
Logic 0 Input Current, I_{IL}	$V_{IL} = 0$ V				
All Pins Except $\overline{\text{RST}}$			40	60	μA
$\overline{\text{RST}}$ Pin			1		mA
Input Capacitance, C_{IN}			10		pF
DIGITAL OUTPUTS[1]					
Output High Voltage, V_{OH}	$I_{SOURCE} = 1.6$ mA	2.4			V
Output Low Voltage, V_{OL}	$I_{SINK} = 1.6$ mA			0.4	V
FLASH MEMORY	Endurance[2]	10,000			Cycles
Data Retention[3]	$T_J = 85°C$	20			Years
FUNCTIONAL TIMES[4]	Time until data is available				
Power-On, Start-Up Time	Normal mode, SMPL_PRD ≤ 0x09		180		ms
	Low power mode, SMPL_PRD ≥ 0x0A		250		ms
Reset Recovery Time	Normal mode, SMPL_PRD ≤ 0x09		60		ms
	Low power mode, SMPL_PRD ≥ 0x0A		130		ms
Sleep Mode Recovery Time	Normal mode, SMPL_PRD ≤ 0x09		4		ms
	Low power mode, SMPL_PRD ≥ 0x0A		9		ms
Flash Memory Test Time	Normal mode, SMPL_PRD ≤ 0x09		17		ms
	Low power mode, SMPL_PRD ≥ 0x0A		90		ms
Automatic Self-Test Time	SMPL_PRD = 0x0001		12		ms
CONVERSION RATE	SMPL_PRD = 0x0001 to 0x00FF	0.413		819.2	SPS
Clock Accuracy				±3	%
Sync Input Clock[5]		0.8		1.2	kHz
POWER SUPPLY	Operating voltage range, VCC	4.75	5.0	5.25	V
Power Supply Current	Low power mode		24		mA
	Normal mode		49		mA
	Sleep mode		500		μA

[1] The digital I/O signals are driven by an internal 3.3 V supply, and the inputs are 5 V tolerant.
[2] Endurance is qualified as per JEDEC Standard 22, Method A117, and measured at −40°C, +25°C, +85°C, and +125°C.
[3] The data retention lifetime equivalent is at a junction temperature (T_J) of 85°C as per JEDEC Standard 22, Method A117. Data retention lifetime decreases with junction temperature.
[4] These times do not include thermal settling and internal filter response times (330 Hz bandwidth), which may affect overall accuracy.
[5] The sync input clock functions below the specified minimum value, at reduced performance levels.

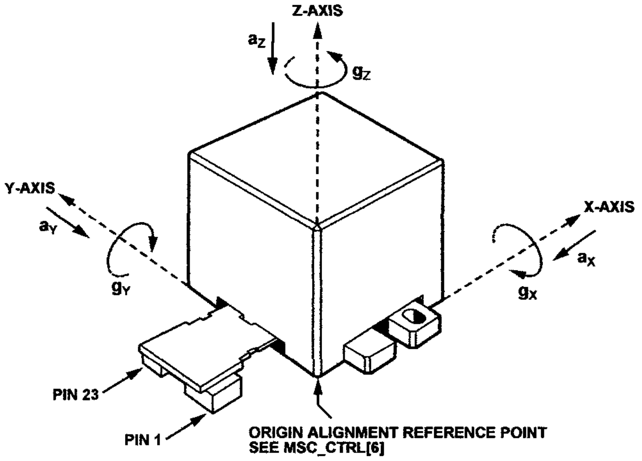
PIN CONFIGURATION AND FUNCTION DESCRIPTIONS



NOTES
1 THIS REPRESENTATION DISPLAYS THE TOP VIEW PINOUT
FOR THE MATING SOCKET CONNECTOR.
2. THE ACTUAL CONNECTOR PINS ARE NOT VISIBLE FROM THE TOP VIEW
3 MATING CONNECTOR SAMTEC CLM-112-02 OR EQUIVALENT
4. DNC = DO NOT CONNECT

08398-005

Figure 5 Pin Configuration



NOTES
1 ACCELERATION (a_X, a_Y, a_Z) AND ROTATIONAL (g_X, g_Y, g_Z) ARROWS
INDICATE THE DIRECTION OF MOTION THAT PRODUCES
A POSITIVE OUTPUT

08398-006

Figure 6 Axial Orientation

Table 5. Pin Function Descriptions

Pin No.	Mnemonic	Type[1]	Description
1	DIO3	I/O	Configurable Digital Input/Output
2	DIO4/CLKIN	I/O	Configurable Digital Input/Output or Sync Clock Input
3	SCLK	I	SPI Serial Clock
4	DOUT	O	SPI Data Output Clocks output on SCLK falling edge
5	DIN	I	SPI Data Input Clocks input on SCLK rising edge
6	$\overline{\text{CS}}$	I	SPI Chip Select
7, 9	DIO1, DIO2	I/O	Configurable Digital Input/Output
8	$\overline{\text{RST}}$	I	Reset
10, 11, 12	VCC	S	Power Supply
13, 14, 15	GND	S	Power Ground
16, 17, 18, 19, 22, 23, 24	DNC	N/A	Do Not Connect
20	AUX_DAC	O	Auxiliary, 12 Bit DAC Output
21	AUX_ADC	I	Auxiliary, 12 Bit ADC Input

[1] I/O is input/output, I is input O is output, S is supply, and N/A is not applicable

Honeywell

Microstructure Pressure Sensors
26PC SMT (1 psi, 5 psi, 15 psi)

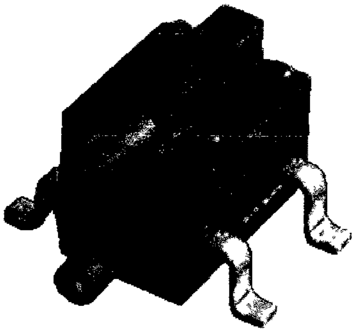
26PC SMT Series

FEATURES

- Alignment pins for position accuracy
- Small package size (less than one half the size of the 26PC) and compact surface mount profile
- 3,18 mm [0.125 in] diameter pick-up feature for use in pick and place machines
- Max peak reflow temperature of 260 °C [500 °F]
- Gage, vacuum gage, differential, wet/wet differential sensing available in one package
- True wet/wet differential sensing
- Proven elastomeric interconnections of the 20PC family
- Temperature compensation
- End point calibration
- Sensor consists of only five components
- Elastomeric construction
- Wide operating temperature range of -40 °C to 85 °C [-40 °F to 185 °F]

TYPICAL APPLICATIONS

- Blood glucose monitors
- Oxygen conservers
- Infusion pumps
- Ventilators
- CPAP (Continuous Positive Airway Pressure) equipment
- Residential fuel cells



The 26PC SMT (Surface Mount Technology) Series pressure sensor, the first offering in the 20PC SMT family of pressure sensors, is a small, low cost, high value, pressure sensing solution for use with printed circuit boards (PCBs). Based on the long established reliability and accuracy of the 26PC pressure sensor, the 26PC SMT offers reduced size with true SMT capability. The smaller size reduces the sensor's footprint on the PCB, thereby reducing the size of the PCB. The 26PC SMT is the first pressure sensor capable of being used with other SMT components on the PCB, helping to lower installation costs and eliminate secondary operations.

The sensor features Wheatstone bridge construction, silicon piezoresistive technology, and ratiometric output for proven application flexibility, design simplicity and ease of manufacture.

Although primarily designed for the medical industry, the 20PC SMT pressure sensor may be applied in any industry that requires a surface mount pressure sensor.

⚠WARNING

PERSONAL INJURY

- DO NOT USE these products as safety or emergency stop devices, or in any other application where failure of the product could result in personal injury.

Failure to comply with these instructions could result in death or serious injury.

⚠WARNING

MISUSE OF DOCUMENTATION

- The information presented in this product sheet is for reference only. Do not use this document as system installation information.
- Complete installation, operation, and maintenance information is provided in the instructions supplied with each product.

Failure to comply with these instructions could result in death or serious injury.

26PC SMT PERFORMANCE CHARACTERISTICS (AT 10 VDC ±0.01 VDC EXCITATION, 25 °C)

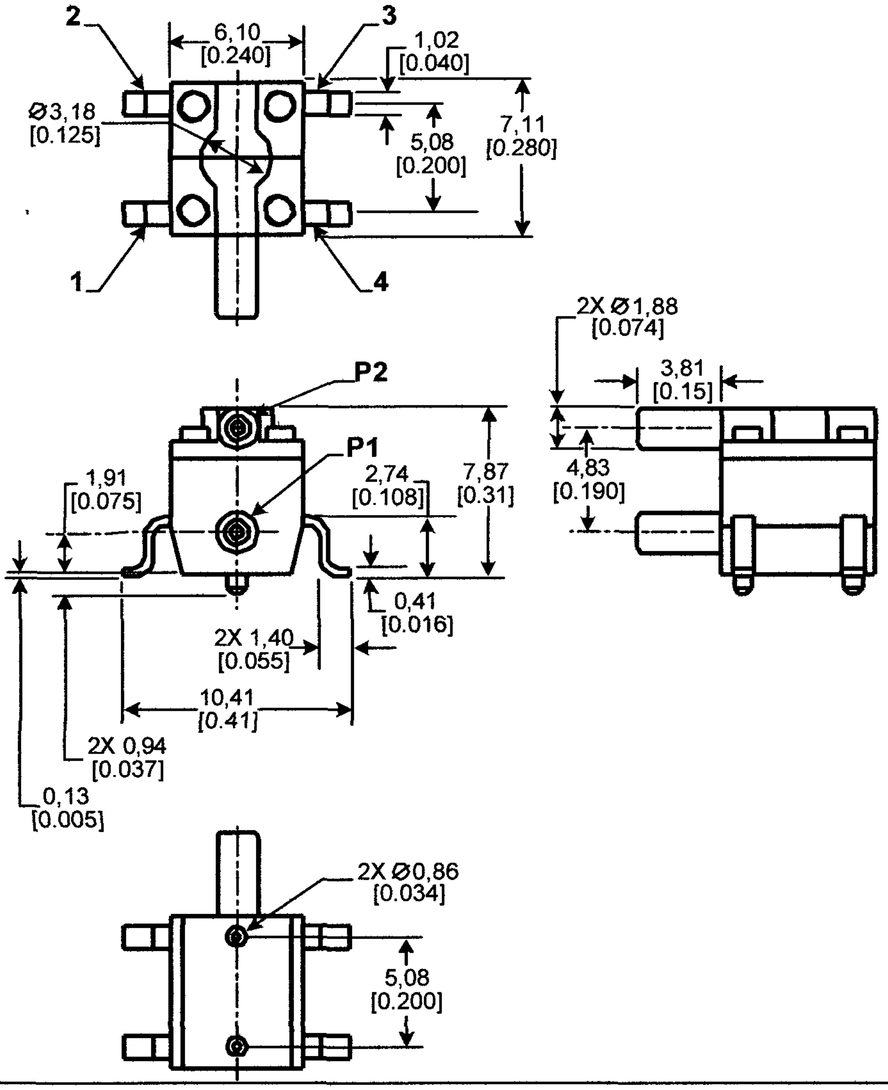
	Min.	Typ.	Max.	Units
Excitation Voltage	—	10.0	16.0	Vdc
Response Time	—	—	1.0	ms
Input Resistance	5.5 k	7.5 k	11.5 k	Ohm
Output Resistance	1.5 k	2.5 k	3.0 k	Ohm
Span P2>P1[1]	**Min.**	**Typ.**	**Max.**	
0 to 1	14.7	16.7	18.7	mV
0 to 5	47	50	53	mV
0 to 15	96	100	104	mV
Null Offset	**Min.**	**Typ.**	**Max.**	
0 to 1	-2.0	0	+2.0	mV
0 to 5	-2.0	0	+2.0	mV
0 to 15	-2.0	0	+2.0	mV
Linearity (BFSL P2>P1)		**Typ.**	**Max.**	
0 to 1	—	±0.50	±1.75	% span
0 to 5	—	±0.50	±1.5	% span
0 to 15	—	±0.50	±1.0	% span
Null Shift 25 °C to 0 °C, 25 °C to 50°C[2]		**Typ.**	**Max.**	
0 to 1	—	—	±1.0	mV
0 to 5	—	—	±1.0	mV
0 to 15	—	—	±1.0	mV
Span Shift 25 °C to 0 °C, 25 °C to 50°C[2]		**Typ.**	**Max.**	
0 to 1	—	±1.5	±4.5	% span
0 to 5	—	±1.0	±1.7	% span
0 to 15	—	±0.75	±1.5	% span
Repeatability and Hysteresis		**Typ.**	**Max.**	
0 to 1	—	±0.2	—	% span
0 to 5	—	±0.2	—	% span
0 to 15	—	±0.2	—	% span
Overpressure P2>P1; P1>P2		**Typ.**	**Max.**	
0 to 1	—	—	20	psi
0 to 5	—	—	20	psi
0 to 15	—	—	45	psi

Notes:
1. Span is the algebraic difference between output at maximum rated operating pressures and output at 0 psi.
2. Temperature error is calculated with respect to 25 °C.

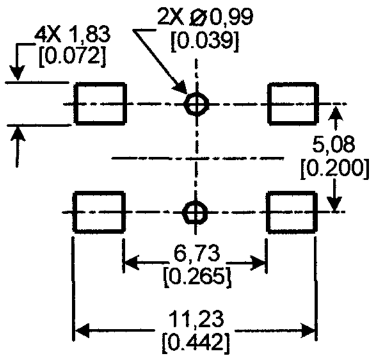
SPECIFICATIONS

Characteristic	Description
Storage Temperature	-55 °C to 100 °C [-67 °F to 212 °F]
Operating Temperature	-40 °C to 85 °C [-40 °F to 185 °F]
Compensated Temperature	0 °C to 50 °C [32 °F to 122 °F]
Alignment Pins	0,86 mm [0.034 in] diameter pins extend through PCB
Port Diameter	1,88 mm [0.074 in] diameter uses standard 0,59 mm [0.0625 in] ID tubing
Port Orientation	Parallel to PCB (low profile on board)
Pick Up Feature	3,18 mm [0.125 in] feature on port cover
SMT Solder	• Sn 96.5 Ag 3.5 No Clean Flux • Sn 63 Pb 37 No Clean Flux
SMT Reflow Profile	Max peak temperature of 260 °C [500 °F] for 10 seconds
Media Compatibility	Both ports are limited to media that are compatible with polyphthalamide, fluorosilicone and silicon.
Shock	Qualification tested to 150 g
Vibration	MIL-STD-202. Method 213 (150 g half sine 11 ms)
Weight	0.5 grams [0.0176 oz]

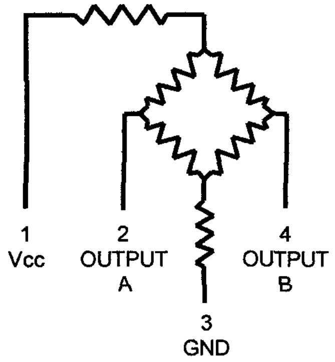
MOUNTING DIMENSIONS mm[in] (for reference only)



SUGGESTED LAND PATTERN



CIRCUIT DIAGRAM



OUTPUT VOLTAGE

Output A increases as P2 pressure increases.

Output B decreases as P2 pressure increases.

Microstructure Pressure Sensors
26PC SMT (1 psi, 5 psi,15 psi)

26PC SMT Series

DATE CODE

_ _ _ _ _ _

Week: Two Digits

Year: Four Digits

CATALOG LISTING NOMENCLATURE

26 PC __ SMT

Package Style: SMT (Surface Mount Technology)

Pressure Range:
01 = 1 psi
05 = 5 psi
15 = 15 psi

Family: PC

Type: Compensated

BRANDING SCHEME

6 __ F

Seal: Fluorosilicone

Pressure Range:
01 = 1 psi
05 = 5 psi
15 = 15 psi

Type: Compensated

TECHNICAL NOTES

Technical Notes that provide further application information on the 26PC SMT are available on the Honeywell web site at: ***http://www.honeywell.com/sensing/prodinfo/pressure/20pc***

WARRANTY/REMEDY

Honeywell warrants goods of its manufacture as being free of defective materials and faulty workmanship. Contact your local sales office for warranty information. If warranted goods are returned to Honeywell during the period of coverage, Honeywell will repair or replace without charge those items it finds defective. The foregoing is Buyer's sole remedy and is **in lieu of all other warranties, expressed or implied, including those of merchantability and fitness for a particular purpose.**

Specifications may change without notice. The information we supply is believed to be accurate and reliable as of this printing. However, we assume no responsibility for its use.

While we provide application assistance personally, through our literature and the Honeywell web site, it is up to the customer to determine the suitability of the product in the application.

For application assistance, current specifications, or name of the nearest Authorized Distributor, check the Honeywell web site or call:
1-800-537-6945 USA
1-800-737-3360 Canada
1-815-235-6847 International
FAX
1-815-235-6545 USA
INTERNET
www.honeywell.com/sensing
info.sc@honeywell.com

Honeywell

Sensing and Control
Honeywell
11 West Spring Street
Freeport, Illinois 61032

www.honeywell.com/sensing

Printed with Soy Ink
on 50% Recycled Paper

008065-1-EN IL50 GLO 0601 Printed in USA

Guide to the Arduino Mini

To get started with the Arduino Mini, follow the directions for the regular Arduino on your operating system (Windows, Mac OS X, Linux), with the following modifications.

- Connecting the Arduino Mini is a bit more complicated than a regular Arduino board (see below for instructions and photos).
- You need to select **Arduino Mini** from the **Tools | Board** menu of the Arduino environment.
- To upload a new sketch to the Arduino Mini, you need to press the reset button on the board immediately before pressing the upload button in the Arduino environment.

Information about the Arduino Mini

The microcontroller (an ATmega168) on the Arduino Mini is a physically smaller version of the chip on the USB Arduino boards, with the following small difference·

- There are two extra analog inputs on the Mini (8 total). Four of these, however, are not connected to the legs that come on the Arduino Mini, requiring you to solder wires to their holes to use them. Two of these unconnected pins are also used by the Wire library (I2C), meaning that its use will require soldering as well.

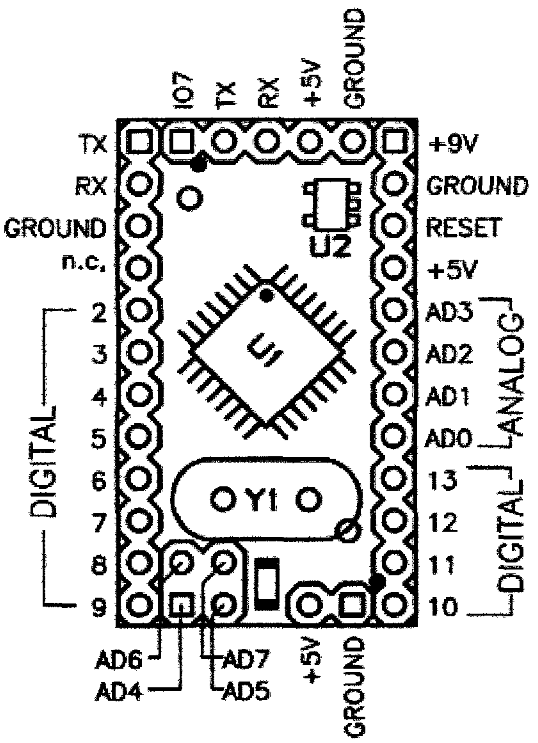
- Also, the Arduino Mini is more **fragile and easy to break** than a regular Arduino board.

- Don't connect more than 9 volts to the +9V pin or reverse the power and ground pins of your power supply, or you might kill the ATmega168 on the Arduino Mini.

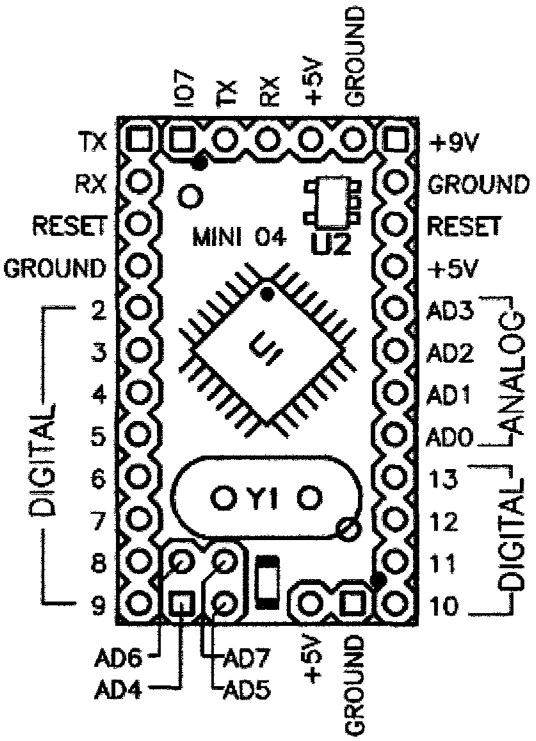
- You can't remove the ATmega168, so if you kill it, you need a new Mini.

Connecting the Arduino Mini

Here's a diagram of the pin layout of the Arduino Mini



Mini 03 pinout (compatible with earlier revisions)



Mini 04 pinout (the ground on the left has moved down one pin)

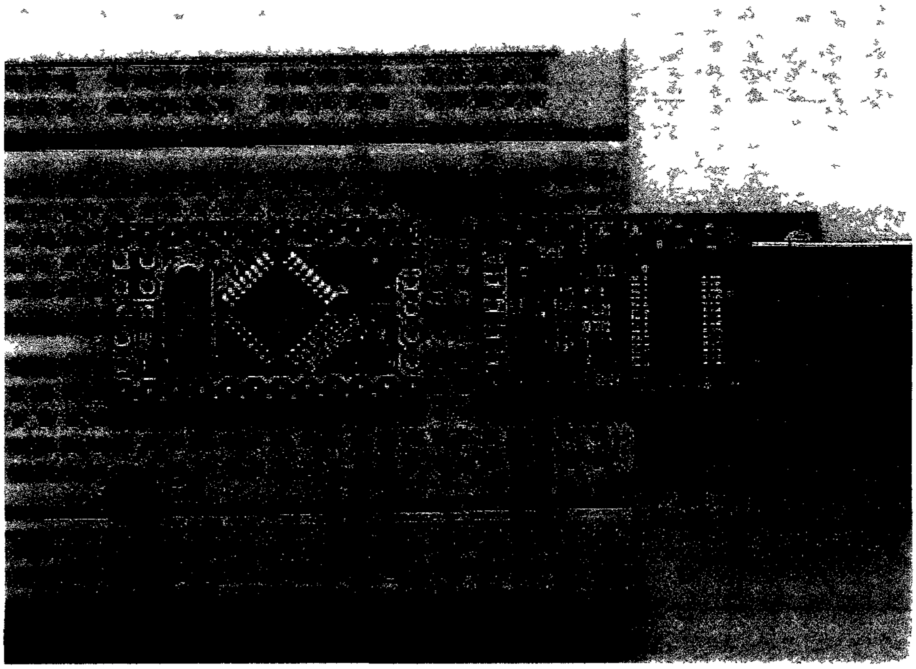
To use the Arduino Mini, you need to connect

- Power This can be a regulated +5V power source (e.g. from the +5V pin of the Mini USB Adapter or an Arduino NG) connected to the +5V pin of the Arduino Mini. Or, a +9V power source (e.g. a 9 volt battery) connected to the +9V pin of the Arduino Mini.
- Ground. One of the ground pins on the Arduino Mini must be connected to ground of the power source.
- TX/RX. These pins are used both for uploading new sketches to the board and communicating with a computer or other device
- Reset. Whenever this pin is connected to ground, the Arduino Mini resets. You can wire it to a pushbutton, or connect it to +5V to prevent the Arduino Mini from resetting (except when it loses power) If you leave the reset pin unconnected, the Arduino Mini will reset randomly.
- An LED. While not technically necessary, connecting an LED to the Arduino Mini makes it easier to check if it's working. Pin 13 has a 1 KB resistor on it, so you can connect an LED to it directly between it and ground When using another pin, you will need an external resistor.

You have a few options for connecting the board the Mini USB Adapter, a regular Arduino board, or your own power supply and USB/Serial adapter.

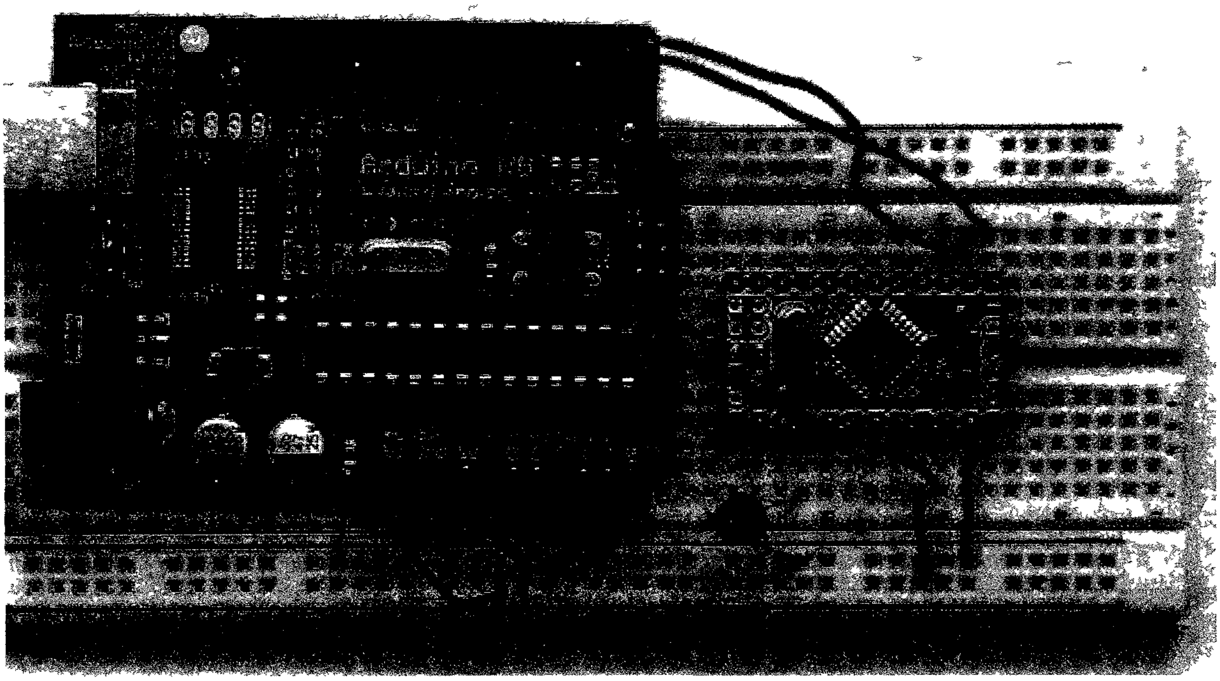
Connecting the Arduino Mini and Mini USB Adapter

Here is a photo showing the Arduino Mini connected to the Mini USB adapter Notice that the reset pin is connected directly to +5V (the orange wire), without a pushbutton. Thus, to reset the Arduino Mini, you will need to unplug and reconnect the USB cable to the Mini USB Adapter, or manually move the orange wire connected to the reset pin from +5V to ground and back



Connecting the Arduino Mini and a regular Arduino

Here's a photo of the Arduino Mini connected to an Arduino NG. The NG has its ATmega8 removed and is being used for its USB connection, power source, and reset button. Thus, you can reset the Arduino Mini just by pressing the button on the NG



Arduino NG