# A NAVIGATION SYSTEM FOR LOW-COST AUTONOMOUS ALL-TERRAIN-VEHICLES

by

Shantanu Mhapankar

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2019

Approved by:

_____
Dr. James M. Conrad

_____
Dr. Hamed Tabkhi

_____
Dr. Aidan Browne

# ABSTRACT

SHANTANU MHAPANKAR. A navigation system for low-cost autonomous all-terrain-vehicles. (Under the direction of DR. JAMES M. CONRAD)

In the last decade, there has been significant research work in the field of autonomous on-road vehicle, but, research in automating off-road vehicles has been largely untouched. The main aim of this thesis was to propose a robust framework for an off-road All-Terrain Vehicle (ATV) that can be used for navigation across all types of terrain. The ATV would navigate to the destination by following various way-points. This framework consists of a Global Positioning System (GPS) and an Inertial Measurement Unit (IMU). The GPS and Inertial Navigation System (INS) are two basic navigation systems. Due to their complementary characteristics in many aspects, a GPS/INS integrated navigation system is more accurate and dependable than having just either one of them. A sensor fusion was implemented for GPS and accelerometer to predict position and velocity using Kalman filtering. After the data was received from the Kalman filtering, and the main controller, i.e. the brain, could compare this data with the position of the GPS way-point and makes decisions regarding which direction the ATV was to be steered. All the actuators of the ATV are controlled by a micro-controller and the brain sends appropriate commands to the microcontroller controlling the actuators. This microcontroller then generates signals for either braking, changing the speed or moving the steering wheel depending on the actuator to which it is connected. All the sensors and actuator controllers are plug-and-play modules that are connected to a single Controller Area Network (CAN) bus, and thus can easily be removed, added, or upgraded. A library was built such that the controllers could only call Application Program Interface (API) functions, thus simplifying debugging the code, adding modularity to the program, and improving readability.

# ACKNOWLEDGEMENTS

I would like to first thank Dr. James M. Conrad for his constant guidance and advice offered throughout this research work. I would like to thank Dr. Hamed Tabkhi and Dr. Aidan Browne for serving on my committee.

I would also like to thank my colleague, Karim Erian for his valuable inputs. Also, I am thankful to Sushil Sundaram for the help he provided with understanding Kalman filtering.

I also want to thank my family without whose constant financial and mental support, this wouldn't have been possible.

TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| API | Application Programming Interface. |
| ATV | All Terrain Vehicle. |
| BSL | Basic Software Layer. |
| CAN | Controller Area Network. |
| CEP | Circular Error Probability. |
| DGPS | Differential Global Positing System. |
| ECE | Electrical and Computer Engineering. |
| EPS | Electronic Power Steering. |
| ETCS | Electronic Throttle Control System. |
| GPGGA | Global Positioning System Fix Data |
| GPGLL | Geographic position, latitude / longitude. |
| GPGRS | GPS range residuals for each satellite. |
| GPGSA | GPS DOP and active satellites. |
| GPGST | Pseudorange measurement noise statistics. |
| GPGSV | GPS Satellites in view. |
| GPIO | General Purpose Input Output. |
| GPRMC | Recommended minimum specific GPS/Transit data. |
| GPS | Global Positing System. |
| GPVTG | Track made good and ground speed. |

| | |
|---|---|
| HWI | Hardware Interface. |
| I/O | Input/Output. |
| I2C | Inter-Integrated Circuit. |
| IMU | Inertial Measurement Unit. |
| INS | Inertial Navigation System. |
| NDOF | Nine Degrees of Freedom. |
| NMEA | National Marine Electronics Association. |
| PPS | Pulse Per Second. |
| PWM | Pulse Width Modulation. |
| SoC | System on Chip. |
| SPI | Serial Peripheral Interface. |
| UART | Universal Asynchronous Receiver Transmitter. |
| USCI | Universal Serial Communication Interface. |
| UTC | Coordinated Universal Time. |
| WDT | Watchdog Timer. |

# CHAPTER 1: INTRODUCTION

Autonomous vehicles have been a subject of substantial research and development in the last few years. Technology giants like Google, Microsoft, Tesla, Argo, Lyft, Uber, Samsung, and Apple have been working towards making these vehicles deployable on roads [1]. Not only has the use case been to drive humans around but Toyota has come up with self-driving boxes to make deliveries easier [2]. These vehicles are called "e-Palette", and after recent announcement of deals with Amazon, Pizza Hut, and Uber, one can expect them to be seen everywhere. Making vehicles autonomous has abundant use. Instead of wasting time driving to work, something productive can be accomplished in that period of time. Disabled people who have to use public transportation or ask help from family or friends, could eventually use these autonomous vehicles and get to work with ease [3].

## 1.1 Motivation

For the aforementioned autonomous vehicles, the path is decided via detailed road maps, lane following and a group of sensor information [4, 5]. However, an All Terrain Vehicle (ATV) is designed for use on rough grounds where normal cars would have trouble navigating. The lower pressure and deep threads on the ATV tires make them the perfect candidate to maneuver off-road. Although the ATV was earlier sold as a recreational vehicle for use in races, their usefulness has soon been realized. Border patrol agents use ATV to make their way through rugged roads so that the inaccessible areas of the border are covered and kept safe [6]. They are also used by emergency medical rescue teams that operate in remote areas on rough terrains. Trailers pulled by an ATV are used to carry medical equipment and transport injured people in a

safer manner through the bumpy terrains [7]. An ATV is useful in agriculture to inspect crops, carry materials and perform various activities [8]. My motivation for this thesis stems from the potential advantages in making the ATV run autonomously. An autonomous ATV can be very useful in wildfire control by deploying them to the forest areas to either aid firefighters or simply apply pesticides across a field. The autonomous ATV can be used to pull a trailer carrying sensing equipment through a pre-defined path and survey topography of nearby areas [9]. Currently, since map data is unavailable for the ATV to choose its own path by, there needs to be another solution so that it can go from point A to point B without a driver seated inside and controlling it or by using a remote control.

## 1.2 Objective of this work

The objective of this thesis is to build a functioning autonomous ATV that can traverse from one point to the other without any human assistance. As an ATV is used in unpaved regions, the major challenge is to work without a definite road map and lanes. The idea is to use GPS and compass sensor data to help the vehicle compensate for the lack of a detailed road map [10]. The first step is to control the ATV by making its actuators electronically driven rather than manually. Furthermore, a communication protocol is to be used that will reduce the number of I/O ports used and the number of interconnections. The standard for an in-vehicle network is the Controller Area Network (CAN) protocol [11]. Once the vehicle can be controlled electronically with an efficient communication protocol, the next step is to use a robust system that the ATV can use to navigate on the ground. As there is a lack of map data, GPS way-points are added across various points between the starting point and the destination [12, 13]. This provides a path that the ATV can use for navigating. The ATV itself uses a GPS to locate its own position; however, the GPS position alone is not sufficient. The ATV still needs to know which direction to move in to pass the way-points and reach the destination. For this purpose, a

magnetometer sensor on the IMU is used. The magnetometer sensor provides axis data which can be used to calculate the angle from the North direction [14, 13]. This robust system of GPS and an IMU sensor provides data to the main controller. The main controller compares this data with GPS way-point data and makes decisions on whether to brake, increase or decrease the speed, or just change the direction in which the ATV is moving.

## 1.3 Contribution

The work on the topic began with studying the work done by earlier teams and finding ways to improve them [15, 16, 17]. After going through the available options, a low power and low cost embedded micro-controller MSP430 was decided to be used to control the actuators on the ATV. An Hardware Interface (HWI) layer was implemented for GPIO, PWM, and WDT on MSP430 to control the actuators. The library was arranged in a way that each module had separate files such that even digital input and digital output had different source files. This made it easier to debug. To implement the CAN protocol, Microchip's MCP2515 was selected as the CAN module. An SPI driver was built using of the HWI layer mentioned above as MCP2515 uses SPI protocol for communication with the MSP430 board. The next step was to work on GPS and IMU modules. The GPS module transmits data to the controller that it is connected to by using UART protocol while the IMU module uses I2C protocol. Drivers for both the protocols were written and the library was updated with them in it. The APIs for GPS and IMU were built on top of these communication drivers[18]. They were tested and results were generated to check their accuracy. The results obtained were very noisy and the GPS data took a second to update. The IMU provides acceleration and this data was used to predicted positions. Thus, a sensor fusion using Kalman filter was implemented to fuse these error-prone measurements and get a more accurate estimate of the system at a much higher frequency. Used the Kalman filtered position and orientation to help the ATV navigate.

## 1.4 Thesis Organization

This thesis is divided into 6 chapters. Chapter 1 introduces the topic, states the motivation behind it, the objective and the contribution towards it. Chapter 2 covers the hardware overview of the ATV and the sensors used. Chapter 3 looks at API library development. Chapter 4 looks at the GPS and IMU sensors in-depth and how they can be fused using Kalman filters. Chapter 5 shows the results obtained in different scenarios. The thesis ends with Chapter 6 showing stating the conclusion.

# CHAPTER 2: HARDWARE OVERVIEW

## 2.1 The ATV

Figure 2.1: The ATV [16].

The ATV used in this project was a Honda "Four Trax Rancher EPS" vehicle provided by Zapata Engineering as can be seen in Figure 2.1. A good amount of work was already completed by previous teams to control the ATV's three main actuators - the throttle, the steering, and the brake. The ATV utilizes a 'drive by wire' Electronic Throttle Control System (ETCS) that replaces the mechanical linkage between the accelerator pedal and the throttle [19]. ETCS uses an all-electronic system equipped

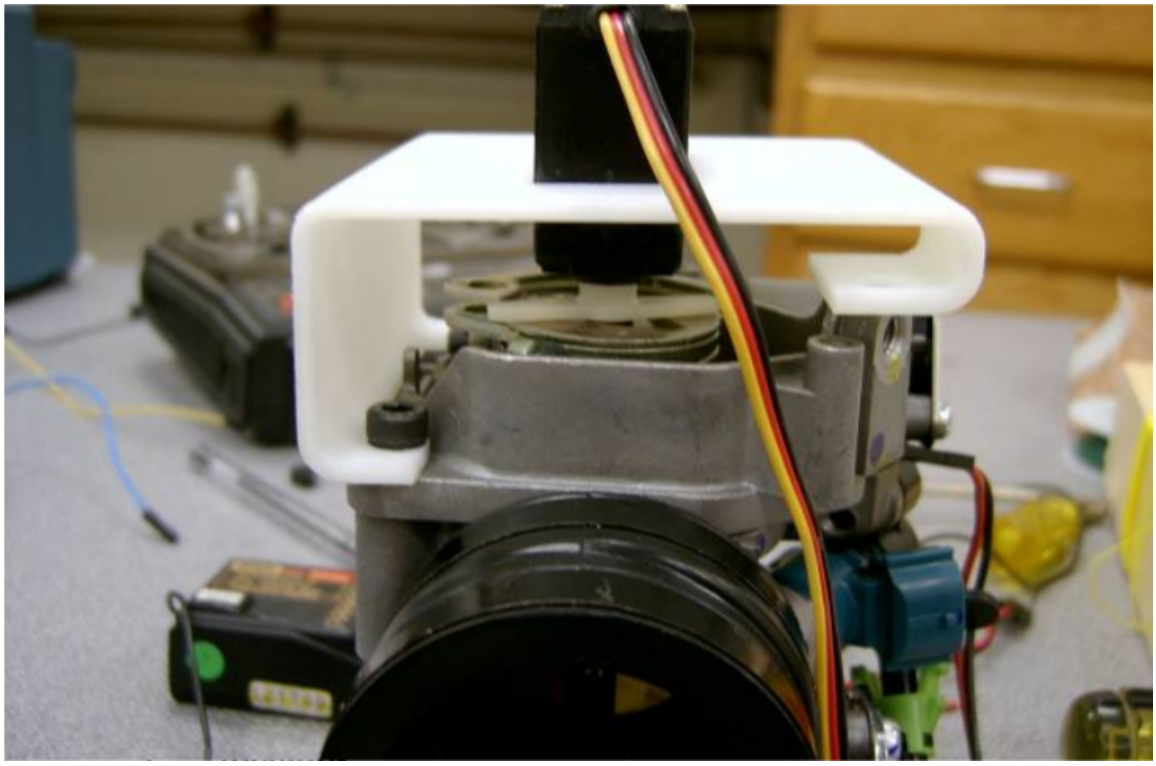

Figure 2.2: Throttle with servo attached [15].

with various sensors to detect throttle position. This position information is then passed to a computer which moves the throttle with a DC motor. For the purpose of controlling the throttle, the ETCS behavior is represented by an electrical servo motor controlled by an embedded board. Another advantage of this throttle system is that when the throttle is not engaged, the vehicle slowes to a stop, which reduces the use of brakes. This throttle control was already implemented [15], and the only thing changed was the controller board. This ATV is also equipped with a steering assist and electronic power steering (EPS) module. A torque sensor is attached to the steering. When the vehicle handle is moved, torque is applied to the steering shaft. The EPS powers the motor in response to this torque. The steering controller went through many iterations. A single channel H-bridge that had three FETs in each leg of the H-bridge was originally used to control the steering [15]. However, it had overheating issues and became less responsive to control commands [16]. A closed

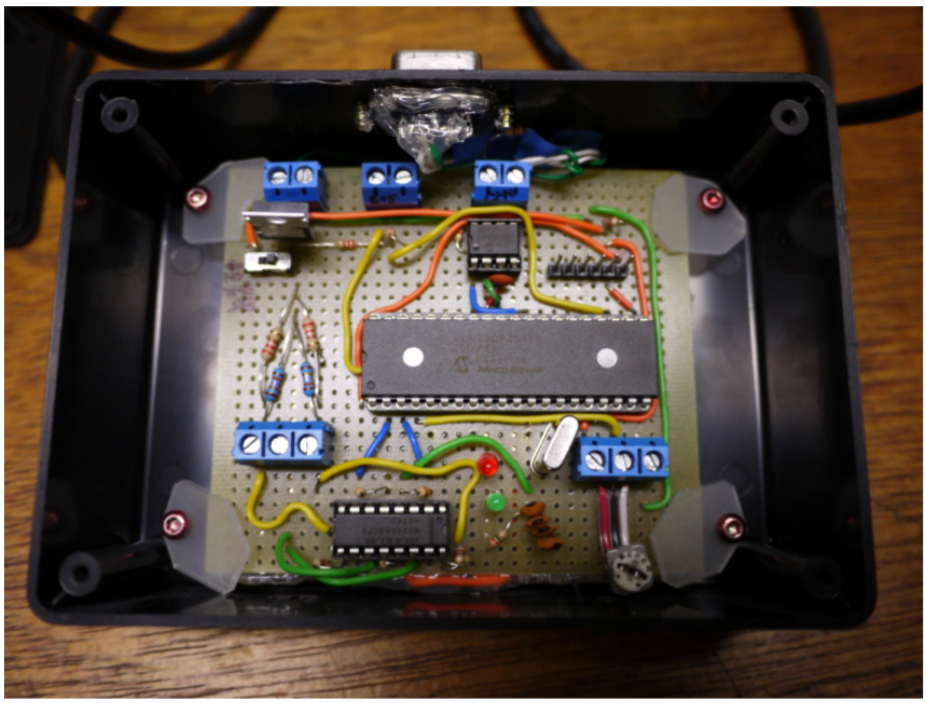

Figure 2.3: Steering controller with CAN Bus [17].

loop system for error feedback was developed with signal conditioning based on the motor error feedback [16]. However, a change in resistance at the torque sensor was noticed whenever the steering was moved [17]. The change was noted and emulated using a circuit that switched between the resistances for moving left and right. The circuit is shown in Figure 2.3. The current systems talks to the steering controller via CAN Bus. The CAN Bus sends PWM signals and based on the value of this PWM signal, the resistance is switched to move either left or right. The braking system was previously implemented as a linear actuator to the foot brake which was controlled by an H-bridge [15] as shown in Figure 2.4. Overheating issues were noticed again, and this H-bridge was replaced with a Pololu motor controller. Currently, there's work ongoing to design a better implementation of the braking system.

## 2.2 Controller Board

The earlier team worked with Renesas board RX63N to move the vehicle using a remote control [17]. For this project, however, it was decided to go with Texas

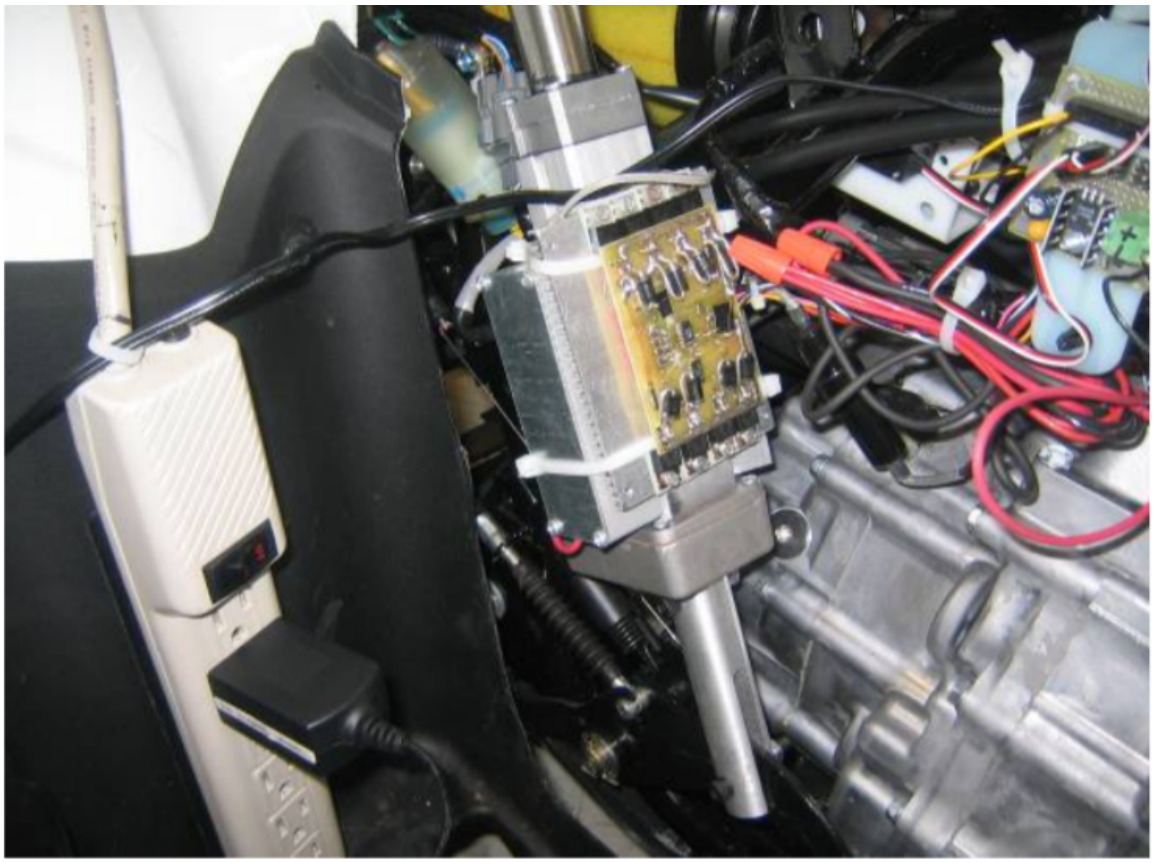

Figure 2.4: Braking System [15].

Instruments low-power microcontroller family - MSP430 Launchpad [20]. Initially, the project started by choosing MSP430G2553 as the controller board to be used. MSP430G2553 has a flash memory of 16KB and RAM of 512B. MSP430 uses the Universal Serial Communication Interface (USCI) for its serial communication modules. This includes I2C, SPI, and UART. MSP430G2553 uses two blocks for USCI - A and B. This limits the number of communication protocols that can be used at the same time to two. USCI_A provides support for SPI and UART while USCI_B provides support for SPI and I2C. The number of I/O pins available out on the launchpad is only 14. On reviewing the options, a more powerful MSP430 Launchpad, MSP430F5529 as shown in Figure 2.5, was selected. This Launchpad has a total of 35 usable I/O pins. Moreover, it consists of four USCI blocks - USCI_A0 and USCI_A1 each supporting UART, and SPI while USCI_B0, and USCI_B1 each

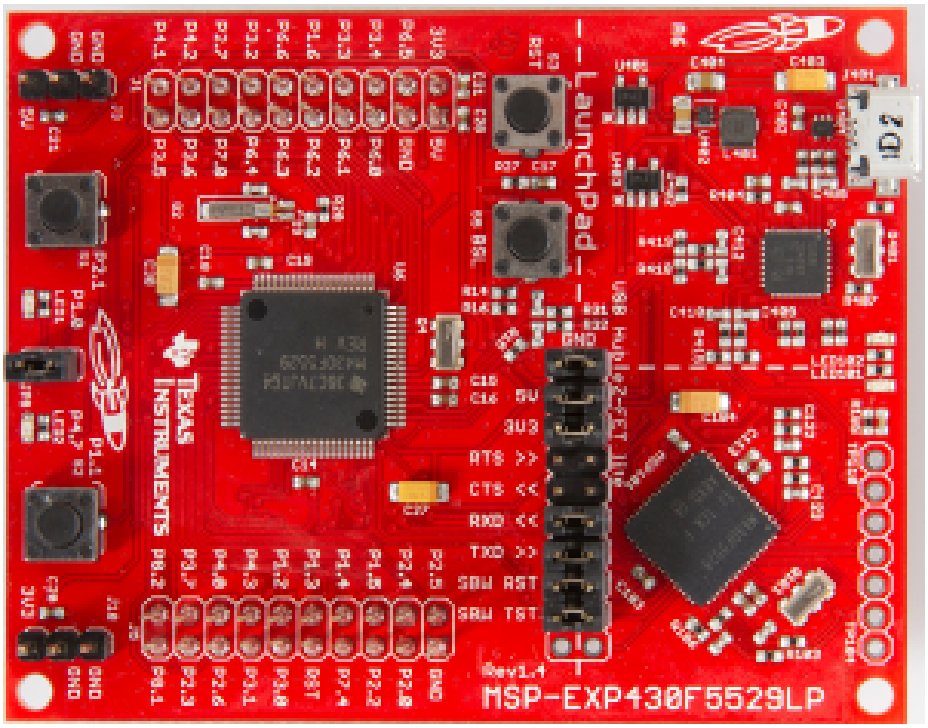


Figure 2.5: TI MSP430F5529

support I2C, and SPI. It also features a flash memory of 128KB and a RAM of 8KB + 2KB which is considerably higher than that of the MSP430G2553. The board also features both 3.3V and 5V on its Launchpad. This eliminates the need for a voltage converter, in-case any sensor module needs it. This board was the most used component in this ATV project, since the sensor modules were interfaced to these boards. Moreover, even the actuators are controlled via signals from this board. All these boards were connected to the main embedded controller via a CAN bus.

## 2.3 GPS

For absolute positioning of the vehicle, a GPS module is needed. GPS is a satellite-based navigation system that provides location and time information. Satellites circle the Earth twice a day in a precise orbit and transmit a unique signal and parameters that allow for precise location decoding. On the other hand, GPS receivers, apply trilateration to calculate the exact location as shown in Figure 2.6. The amount of time taken to receive a transmitted signal is used to measure the distance [22].

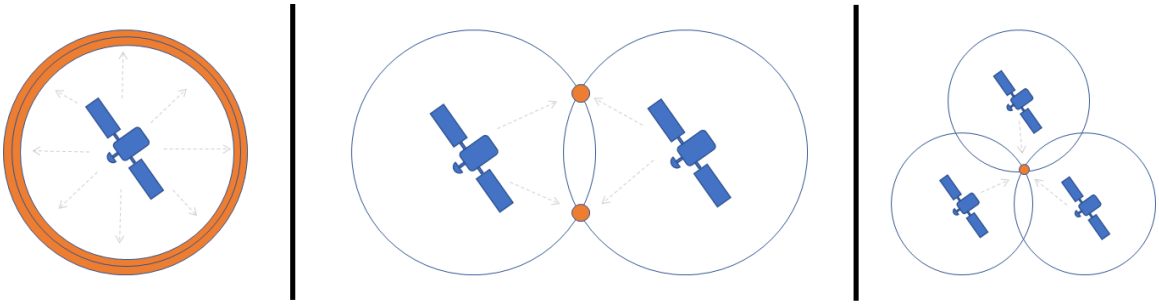

Figure 2.6: GPS position calculation [21].

The module chosen was Adafruit's Ultimate GPS as shown in Figure 2.7. Adafruit has built this breakout board around the MediaTek's all-in-one GPS system on a chip (SoC) - MTK3339. This module can track up to 22 satellites on 66 channels, and has a high-sensitivity receiver [23]. It has an update rate of 1 to 10 Hz. The position accuracy claimed in the datasheet to be up to 3 meters at 50% Circular Error Probability (CEP). It comes with a multi-tone active interference canceller to reject external RF interference with comes from nearby components [24]. This helps improve GPS reception without the need for a hardware design change. It can cancel up to twelve independent channels interference. The Time To First Fix (TTFF) is typically 1 second. It also uses an embedded assist system for quick positioning by saving predicted GPS position information into memory. This information will be used for position when there's no position data from the satellites and also improves TTFF in such areas. This module also features a built-in antenna that gives it -165 dB sensitivity however, an external 3V active antenna can be connected via the uFL connector for higher accuracy. It automatically detects the active antenna and makes the switch. A data-logging capability is also present. It uses its own Flash memory for logging GPS data in the format: Coordinated Universal Time (UTC), date, latitude, longitude, and height with a max logging time frame of 16 hours. All the embedded board needs to do is to pass a "Start logging" command and can go to sleep to save power. This data is logged every 15 seconds and when a fix is present. Unfortunately, that time interval is hard-coded into the firmware of the GPS and can't be changed

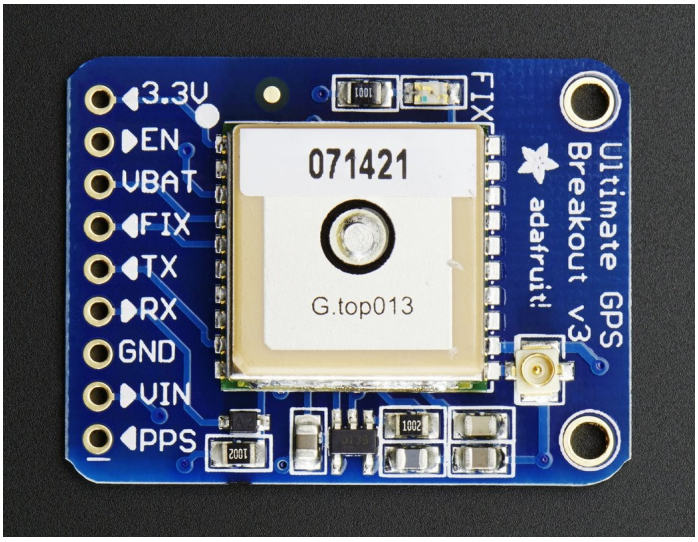


Figure 2.7: Adafruit Ultimate GPS

to 1 second to suit the project. While the earlier version (PA6C) featured this data-logging ability, it lacked an external antenna support. One more advantage of the PA6H version of MTK3339 is that it has pulse per second (PPS) output. It indicates the start of a second. The PA6H module provides highly accurate 1PPS timing on the PPS pin to synchronize to GPS time after 3D-Fix. The module operates at a voltage range of 3.3V to 5VDC. It also has an optional footprint for a coin cell to run the RTC to allow warm start. The GPS outputs data at a default baud rate of 9600. The data output follows the National Marine Electronics Association (NMEA) 0183 protocol. The module communicates with the MSP430 using TTL Serial.

## 2.4 IMU

To get the orientation of the vehicle with respect to magnetic North, a magnetometer is needed. Magnetic North refers to the Earth's magnetic pole position and differs from true North by about 11.5 degrees [25]. The difference between magnetic North and true North can be +/-25 degrees. This difference is known as the declination angle. The X-axis points forward, Y-axis points right and the Z-axis points downwards. To calculate the true heading angle, x and y components of the magnetometer were obtained and heading is calculated from that and then the declination

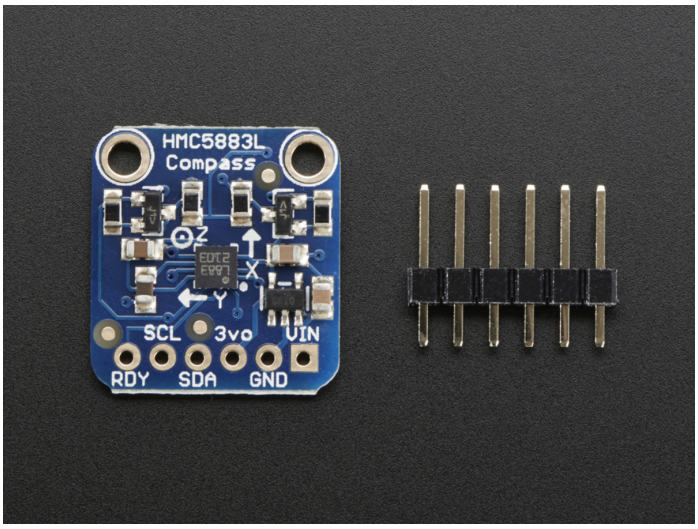


Figure 2.8: Adafruit Triple-Axis Magnetometer HMC5883L

angle is either subtracted or added. The project started with work on Adafruit's HMC5883L as shown in Figure 2.8 which is a triple-axis magnetometer. This module communicates using I2C communication protocol. Once an I2C driver was written for MSP430, the values received from HMC5883L were used to calculate the heading angle. When the module is placed flat on the surface, the heading angle received is correct. However, once there is tilt, the heading angle shows incorrect reading. As the vehicle is expected to move in uneven terrains, tilt-compensation was vital to this project. To compensate for the tilt, an accelerometer needs to be added. So it was decided to move to an IMU BNO055.

The BNO055 consists of an accelerometer, a magnetometer, and a coriolis vibrating gyroscope instead of just a magnetometer in HMC5883L. The acceleration is measured by measuring the change in capacitance. A mass attached to a string moves when acceleration is applied thus changing the capacitance. For a gyroscope, there is displacement of mass when the external angular rate is applied. This displacement changes capacitance which corresponds to angular rate. The magnetometer gives a reading based on changes caused due to magnetic force.Therefore, this IMU module can measure tangential acceleration, rotational acceleration and, strength of the local

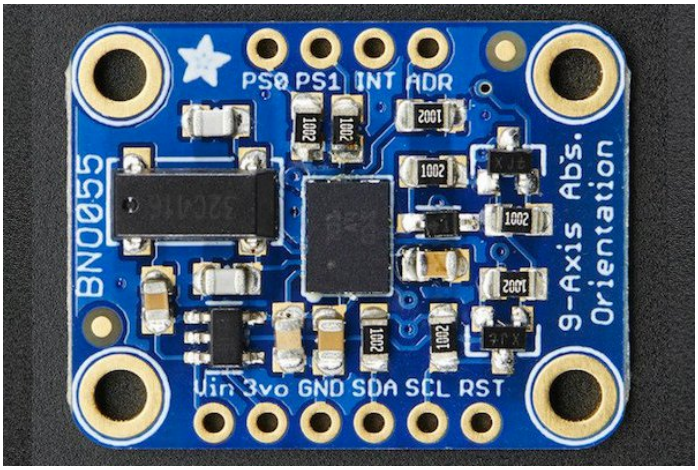


Figure 2.9: Adafruit BNO055 Absolute Orientation Sensor

magnetic field. One of the major reason for choosing Bosch's BNO055 over other IMU modules was that it possesses a high-speed ARM Cortex-M0 processor that does a lot of the math in itself. This reduces the need for doing math on the MSP430 which is a huge plus as the memory is limited. Also, this sensor works on I2C protocol like HMC5883L so there was no need to write another communication driver.

# CHAPTER 3: API LIBRARY IMPLEMENTATION

```
// Set GPIO function for Port 1 & 2
P1SEL = 0;
P2SEL = 0;
// Set P1.0 and P2.0 direction as output
P1DIR |= BIT0;
P2DIR |= BIT0;
// Set P1.0 and P2.0 value as high
P1OUT |= BIT0;
P2OUT |= BIT0;
```

Figure 3.1: Turning on LEDs at P1.0 and P2.0

The ATV can be controlled by controlling its three actuators - a brake actuator, steering actuator, and, throttle actuator. To control these, an MSP430 is used. Programming the MSP430 involves changing registers specific to the module used. To set a pin as high or low, there are certain steps. Since the port pins are multiplexed, the select register (PxSEL) is cleared to specify the pin function as a GPIO. Then, the direction register (PxDIR) is set to assign the pin direction as an output pin. After this, to turn the pin low or high, the out register (PxOUT) is set to the desired value. Configuring a pin as digital input or output is essential to most applications and is used more than a few times. This is where the usage of an application programming interface (API) becomes necessary. An API in simple terms is a well-defined function or a set of such functions that do exactly what is stated. In the library, a simple API call gpioSetOP is used to set a pin as output and gpioSetVal is used to set the value of the pin. Instead of code as seen in Figure 3.1, a much better readable and understandable code is seen in Figure 3.2 where there are APIs that hide the abstraction

```
// Set output
gpioSetOP(GPIO_PORT_P1, GPIO_PIN0);
gpioSetOP(GPIO_PORT_P2, GPIO_PIN0);

// Set to desired value
gpioSetVal(GPIO_PORT_P1, GPIO_PIN0, 1);
gpioSetVal(GPIO_PORT_P2, GPIO_PIN0, 0);
```

Figure 3.2: Using APIs for turning on the LEDs

```
void pinSetOP(unsigned int port, int pin) {
    pinSetOffsetSet(port, pin, GPIO_DIR_REG_OFFSET);
}

void gpioSetOP(unsigned int port, int pin) {
    pinSetOP(port, pin);
    pinClearSEL(port, pin);
}

void gpioSetVal(unsigned int port, int pin, int val) {
    if (val) pinSetOffsetSet(port, pin, GPIO_OUT_REG_OFFSET);
    else pinSetOffsetClear(port, pin, GPIO_OUT_REG_OFFSET);
}
```

Figure 3.3: Digital output implementation in the library

below it. The implementation for these function is done in the library and is shown in Figure 3.3. The aim of building this library was to make it as standardized as possible so other teams could build on it in the future.

A Basic Software Layer (BSL) was created which involved libraries for digital input, digital output, pulse width modulation, and communication protocol like UART, SPI, and I2C. The reason for implementing the drivers for digital input and digital output separately instead of having one GPIO driver file is because it makes bug fixing and making changes to the code much easier. If there is a bug in the digital output driver, just that driver and its dependent files need to be recompiled not other files. On top of this basic software layer, higher level drivers for CAN bus, GPS, and IMU modules were built.

```
void pinSetOffsetSet(unsigned int port, int pin, int offset) {
    int portAddress = GPIO_PORT_ADD_TABLE[port];
    (*((volatile int *)(portAddress + offset))) |= pin;
}

void pinSetOffsetClear(unsigned int port, int pin, int offset) {
    int portAddress = GPIO_PORT_ADD_TABLE[port];
    (*((volatile int *)(portAddress + offset))) &= ~(pin << 8);
}
```

(a) MSP430G2553

```
void pinSetOffsetSet(unsigned int port, int pin, int offset) {
    int portAddress = GPIO_PORT_ADD_TABLE[port];
    if (!(port & 1)) (*((volatile int *)(portAddress + offset))) |= pin;
    else (*((volatile int *)(portAddress + offset))) |= (pin << 8);
}

void pinSetOffsetClear(unsigned int port, int pin, int offset) {
    int portAddress = GPIO_PORT_ADD_TABLE[port];
    if (!(port & 1)) (*((volatile int *)(portAddress + offset)))  &= ~pin;
    else (*((volatile int *)(portAddress + offset))) &= ~(pin << 8);
}
```

(b) MSP430F5529

Figure 3.4: Register Accessing

As mentioned earlier, the change from MSP430G2553 to MSP430F5529 was made. The libraries were already written for MSP430G2553 at that point. The only thing needed to be changed after that was port addresses in a header file and register accessing. This is because for some reason the register offsets in MSP430G2553 merge two ports and access it in such a way that each port has a register one after the other. Whereas in MSP430G2553 the ports aren't merged. The code for register access can be seen in Figure 3.4. The GPS module communicates via UART protocol. A GPS driver was written on top of the HWI. This GPS driver initializes the UART driver to a baud rate of 9600 as the module runs at that speed by default. Since the NMEA messages outputted by the GPS start with a '$', the received character isn't added to any buffer until '$' is received. Once the start of that message is received, the rest of the data is filled into the buffer until a newline character, '\n' is received. This means a full NMEA message has been received. Since the GPS module was set to output just GPGGA (Global Positioning System Fix Data) and GPRMC (Recommended minimum specific GPS/Transit data) messages, the parsing code checks which one is

```
 $GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M,46.9,M,,*47

Where:
     GGA          Global Positioning System Fix Data
     123519       Fix taken at 12:35:19 UTC
     4807.038,N   Latitude 48 deg 07.038' N
     01131.000,E  Longitude 11 deg 31.000' E
     1            Fix quality: 0 = invalid
                               1 = GPS fix (SPS)
                               2 = DGPS fix
                               3 = PPS fix
                               4 = Real Time Kinematic
                               5 = Float RTK
                               6 = estimated (dead reckoning) (2.3 feature)
                               7 = Manual input mode
                               8 = Simulation mode
     08           Number of satellites being tracked
     0.9          Horizontal dilution of position
     545.4,M      Altitude, Meters, above mean sea level
     46.9,M       Height of geoid (mean sea level) above WGS84
                      ellipsoid
     (empty field) time in seconds since last DGPS update
     (empty field) DGPS station ID number
     *47          the checksum data, always begins with *
```

(a) GPGGA

```
$GPRMC,123519,A,4807.038,N,01131.000,E,022.4,084.4,230394,003.1,W*6A

Where:
     RMC          Recommended Minimum sentence C
     123519       Fix taken at 12:35:19 UTC
     A            Status A=active or V=Void.
     4807.038,N   Latitude 48 deg 07.038' N
     01131.000,E  Longitude 11 deg 31.000' E
     022.4        Speed over the ground in knots
     084.4        Track angle in degrees True
     230394       Date - 23rd of March 1994
     003.1,W      Magnetic Variation
     *6A          The checksum data, always begins with *
```

(b) GPRMC

Figure 3.5: GPGGA and GPRMC decoded [26].

received and then extracts latitude, latitude direction, longitude, longitude direction, time and speed from them. The decoding of GPGGA, and GPRMC is shown in Figure 3.5. The latitude and longitude values received were in degrees, minutes, and seconds format. The GPS driver converts them into decimal degrees and then uses these values to get distance and bearing angle between two points. The BNO055 IMU communicates using I2C communication. The BNO055 driver initializes the I2C driver by passing the slave address it wants to communicate to. This slave address is the I2C address of BNO055 which is 0x28. There's a sequence that needs to be

followed during this communication. For writing to a register of the slave via I2C, first, set the start bit and the transmit bit in the control register. Then, send the register to write to. Followed by the value to set the register to. And finally, set the stop bit in the control register. For reading a register via I2C, the start bit and the transmit bit is set. Then, the register address is written. Once this is done, the transmit bit in the control register is cleared to read from the register. Once the register is read, the stop bit is set. While this seemed simple enough, the values received didn't match the ones expected. This was due to the fact that MSP430 modules need the stop bit sent before reading the last byte from the register. The I2C driver was also used for controlling the new braking system which uses T'Rex motor controller. However, for that application, it was simply sending the motor values for the brake and no register writing was required. Lastly, the CAN module MCP2515 uses SPI communication which provides full duplex communication and thus, higher speed.

# CHAPTER 4: NAVIGATION SYSTEM

## 4.1 GPS

### 4.1.1 Initial Settings

```
$GPGLL,3518.5114,N,08044.5192,W,181403.000,A,D*48
$GPGSA,A,3,30,09,07,23,08,11,,,,,,,2.13,1.93,0.89*0C
$GPGSV,4,1,13,08,77,141,21,27,54,054,18,09,52,249,31,07,50,320,22*7D
$GPGSV,4,2,13,23,39,199,24,44,33,231,32,16,29,055,,11,24,172,16*79
$GPGSV,4,3,13,30,18,308,25,18,15,149,,28,05,257,,26,05,070,*72
$GPGSV,4,4,13,01,01,169,*45
$GPRMC,181403.000,A,3518.5114,N,08044.5192,W,0.35,255.81,060419,,,D*78
$GPVTG,255.81,T,,M,0.35,N,0.64,K,D*37
$GPGGA,181404.000,3518.5114,N,08044.5191,W,2,6,1.93,207.6,M,-32.5,M,0000,0000*68
$GPGLL,3518.5114,N,08044.5191,W,181404.000,A,D*4C
$GPGSA,A,3,30,09,07,23,08,11,,,,,,,2.13,1.93,0.89*0C
$GPGSV,4,1,13,08,77,141,19,27,54,054,17,09,52,249,31,07,51,320,21*7B
$GPGSV,4,2,13,23,39,199,24,44,33,231,32,16,29,055,,11,24,172,16*79
$GPGSV,4,3,13,30,18,308,25,18,15,149,,28,05,257,,26,05,070,*72
$GPGSV,4,4,13,01,01,169,*45
$GPRMC,181404.000,A,3518.5114,N,08044.5191,W,0.36,255.81,060419,,,D*7F
$GPVTG,255.81,T,,M,0.36,N,0.66,K,D*36
$GPGGA,181405.000,3518.5115,N,08044.5191,W,2,6,1.93,207.6,M,-32.5,M,0000,0000*68
$GPGLL,3518.5115,N,08044.5191,W,181405.000,A,D*4C
$GPGSA,A,3,30,09,07,23,08,11,,,,,,,2.13,1.93,0.89*0C
$GPGSV,4,1,13,08,77,140,18,27,54,054,17,09,52,249,30,07,51,320,20*7B
$GPGSV,4,2,13,23,39,199,24,44,33,231,32,16,29,055,,11,24,172,16*79
$GPGSV,4,3,13,30,18,308,25,18,15,149,,28,05,257,,26,05,070,*72
$GPGSV,4,4,13,01,01,169,*45
```

Figure 4.1: Raw NMEA strings received from GPS

Adafruit's Ultimate GPS used in this project gives output via NMEA frames. By default, GPS displays NMEA data like GPGLL (Geographic position, latitude / longitude), GPGRS (GPS range residuals for each satellite), GPGGA, GPGSA (GPS DOP and active satellites), GPRMC, GPVTG (Track made good and ground speed), and GPGSV (GPS Satellites in view). You can see the raw output in Figure 4.1. Among all these NMEA sentences, GPGGA provides the essential longitude, and lat-

itude data on current fix and GPRMC gives velocity information. Since these were the two NMEA sentences that position data is extracted from, other NMEA sentences were turned off. This is done by passing sending the packet format shown in 4.1

$$\$PMTK314,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0*28 \tag{4.1}$$

In this equation, '$' is the preamble and PMTK is the talker ID. Packet type 314 was used to set the NMEA output frequency desired of each NMEA message. The next 8 bits were used to set the frequency of GPGLL, GPRMC, GPVTG, GPGGA, GPGSA, GPGSV, GPGRS, and GPGST (Pseudorange measurement noise statistics). The value of frequency can be 0 to 5 where 0 means disabled, 1 means output once every one position fix, 2 means output once every two position fixes, 3 means output once every three position fixes, 4 means output once every four position fixes, and 5 means output once every five position fixes. However, GPGST data even when enabled was not received. For this project, GPGGA and GPRMC were set to output once every position fix as shown in Figure 4.2. It was suggested in the Adafruit Ultimate GPS overview page that for reading just the GPGGA and GPRMC output, the position fix update rate can be increased to 5Hz as opposed to the default 1Hz. However, it is kept to 1Hz in the project to compensate for time being spent in parsing the NMEA sentences to get desired values from them. To change the fix update rate, the command "$PMTK220,val" can be sent to the GPS module via UART where val corresponds to the value of update time in milliseconds. So for updating each second, i.e. at a 1Hz frequency, it is set to "$PMTK220,1000".

### 4.1.2 Distance Calculation

The latitude and longitude values received from the NMEA messages were in the form of degrees, minutes, and seconds. To convert them to decimal degrees, the equation 4.2 was used where d is the degree, m is minutes, s is seconds and dd is

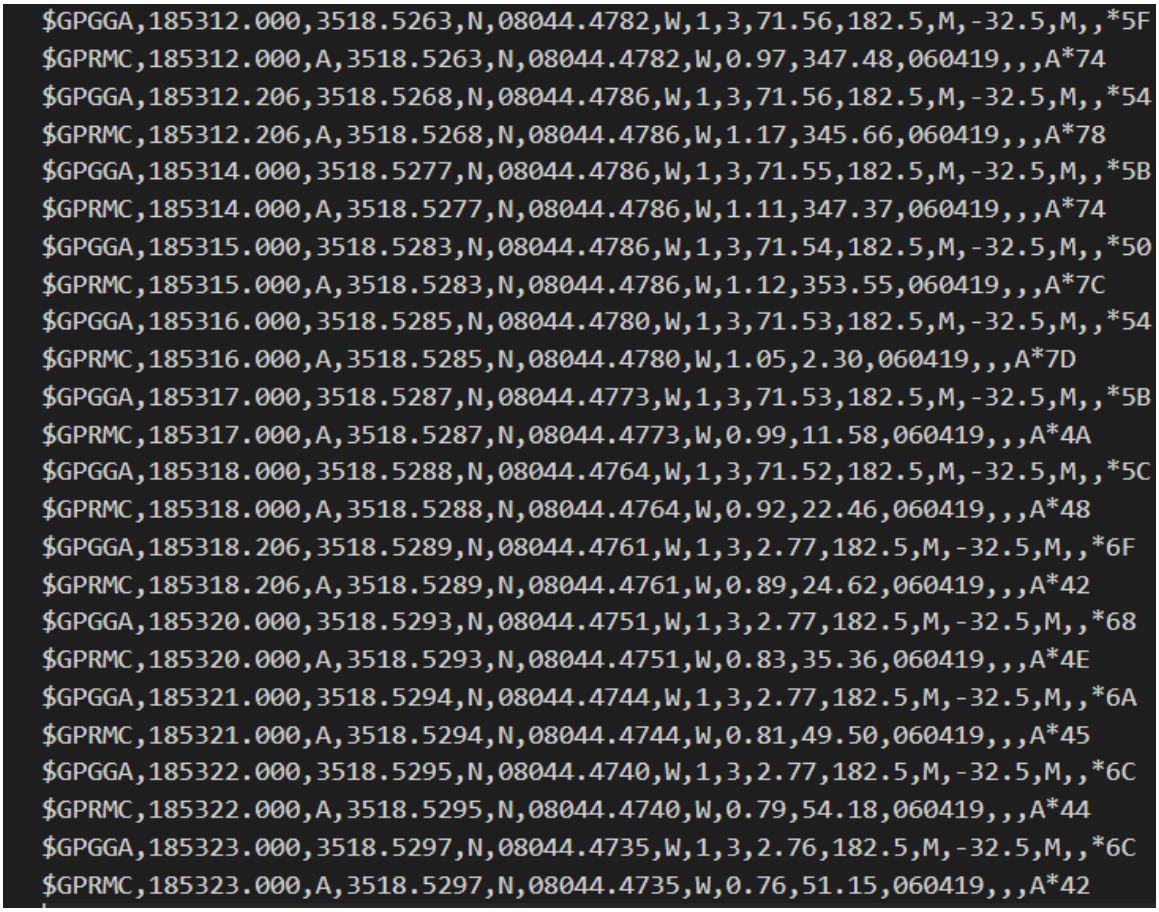


Figure 4.2: NMEA data after only GPGGA and GPRMC were enabled

decimal degree.

$$dd = d + m/60 + s/3600 \tag{4.2}$$

The result received in dd is then multiplied by '-1' if the direction is either South or West. This conversion takes place for both longitude and latitude. Once the conversion is done, the next step is distance calculation. It is assumed that the final destination is known by the vehicle. Work is currently ongoing to build a network that transmits way-points GPS data to the vehicle which will eventually be merged. To calculate the distance between two latitude and longitude points, there are two very famous formulas: Haversine formula and Vincenty formula.

The Haversine formula uses the great-circle distance between two latitude and longitude points [27]. This formula assumes that the Earth is a perfect sphere. It

gives distance by calculating the shortest distance between two points via a line over the sphere. The latitude and longitude points were firstly converted to radians by multiplying them by '0.0174533'. The delta of the latitude and longitude is taken by subtracting the two points as shown in Equation 4.3.

$$\begin{aligned} dlat &= lat1 - lat2, \\ dlon &= lon1 - lon2 \end{aligned} \tag{4.3}$$

Using these delta values and the point values, the Haversine formula listed in Equations 4.4, 4.5, and 4.6 can be applied. As can be seen in Equation 4.6, Earth's radius is multiplied by y. However, Earth's radius depends upon the specific location on Earth. It varies from 6356.75 km at the poles to 6378.13 km at the equator. This introduces small errors when the distance is large. However, for small distances, it seemed to give accurate distance readings.

$$x = \sin(dlat/2) \cdot \sin(dlat/2) + \cos(lat1) \cdot \cos(lat2) \cdot \sin(dlong/2) \cdot \sin(dlong/2) \tag{4.4}$$

$$y = 2 \cdot arctan2(sqrt(x), sqrt(1-x)) \tag{4.5}$$

$$distance = Earthradius \cdot y \tag{4.6}$$

A more accurate formula is Vincenty's formula, which assumes that the Earth is an oblate spheroid. And which more closely matches the Earth's real shape [28]. However, after looking at the huge computation needed for Vincenty formula in Figure 4.3, there were doubts whether a microcontroller could handle such recursive and extensive math and the delay would cause. Also, as stated above, the Haversine formula seemed to give accurate values for short distances. According to [30], the difference in the accuracy is just 0-0.034%. However, the Vincenty formula was found to take twice the amount of time as compared to the Haversine formula. It is assumed

```
const L = λ2 - λ1;
const tanU1 = (1-f) * Math.tan(φ1), cosU1 = 1 / Math.sqrt((1 + tanU1*tanU1)), sinU1 = tanU1 * cosU1;
const tanU2 = (1-f) * Math.tan(φ2), cosU2 = 1 / Math.sqrt((1 + tanU2*tanU2)), sinU2 = tanU2 * cosU2;

const λ = L, λ′, iterationLimit = 100;
do {
    const sinλ = Math.sin(λ), cosλ = Math.cos(λ);
    const sinSqσ = (cosU2*sinλ) * (cosU2*sinλ) + (cosU1*sinU2-sinU1*cosU2*cosλ) * (cosU1*sinU2-sinU1*cosU2*cosλ);
    const sinσ = Math.sqrt(sinSqσ);
    if (sinσ==0) return 0;  // co-incident points
    const cosσ = sinU1*sinU2 + cosU1*cosU2*cosλ;
    const σ = Math.atan2(sinσ, cosσ);
    const sinα = cosU1 * cosU2 * sinλ / sinσ;
    const cosSqα = 1 - sinα*sinα;
    const cos2σM = cosσ - 2*sinU1*sinU2/cosSqα;
    if (isNaN(cos2σM)) cos2σM = 0;  // equatorial line: cosSqα=0 (§6)
    const C = f/16*cosSqα*(4+f*(4-3*cosSqα));
    λ′ = λ;
    λ = L + (1-C) * f * sinα * (σ + C*sinσ*(cos2σM+C*cosσ*(-1+2*cos2σM*cos2σM)));
} while (Math.abs(λ-λ′) > 1e-12 && --iterationLimit>0);
if (iterationLimit==0) throw new Error('Formula failed to converge');

const uSq = cosSqα * (a*a - b*b) / (b*b);
const A = 1 + uSq/16384*(4096+uSq*(-768+uSq*(320-175*uSq)));
const B = uSq/1024 * (256+uSq*(-128+uSq*(74-47*uSq)));
const Δσ = B*sinσ*(cos2σM+B/4*(cosσ*(-1+2*cos2σM*cos2σM)-
    B/6*cos2σM*(-3+4*sinσ*sinσ)*(-3+4*cos2σM*cos2σM)));

const s = b*A*(σ-Δσ);

const fwdAz = Math.atan2(cosU2*sinλ,  cosU1*sinU2-sinU1*cosU2*cosλ);
const revAz = Math.atan2(cosU1*sinλ, -sinU1*cosU2+cosU1*sinU2*cosλ);
```

Figure 4.3: Java Implementation of Vincenty [29].

that the way-points will be at short distances. Hence, because of the simplicity of Haversine formula and the fact that it works well for small distances, it was chosen to calculate distance.

### 4.1.3 Bearing Angle

While Haversine's formula provides distance, there's still direction to be calculated. Bearing angle is used for this purpose. Bearing is calculated clockwise from the true North direction. It gives a value in degrees in the range of 0 to 360. This means that 0° points to the North, 90° points to the East, 180° points to the South and 270° points to the West. Bearing is given by the Equation 4.7. Here, X and Y were given by Equations 4.8 and 4.9. dlat and dlon correspond to the delta values of the two points as shown in Equation 4.3.

$$\beta = arctan2(X, Y) \tag{4.7}$$

$$X = \cos(lat2) * \sin(dLon) \tag{4.8}$$

$$Y = \cos(lat1) * \sin(lat2) - \sin(lat1) * \cos(lat2) * \cos(dlon) \tag{4.9}$$

The bearing now calculated is the initial bearing angle. For final bearing, the initial bearing is calculated from the endpoint to the start point and reversed. However, the ATV's position acts as the start point and each way-point acts as the end point. For this reason, only the initial bearing is of importance. atan2 returns values in the range -180° to +180°, which needs to be normalized to 0° to 360°. This can be achieved by the Equation 4.10. The modulo operator (%) makes sure that when the bearing angle is 0° to 180°, the same value is returned despite adding 360. While, if the bearing angle is in the range -180° to -1°, the modulo operator here will give values in the range of 180° to 159°.

$$\beta = (\beta + 360)\%360; \tag{4.10}$$

## 4.2 IMU

The IMU BNO055 has a powerful microcontroller of its own and can either work in a non-fusion mode or a fusion mode. In non-fusion mode, it outputs raw 3-axis values for accelerometer, gyroscope and magnetometer. To get the heading angle from this module, the magnetometer values can be used. Michael [25] provides the formula for getting the desired angle as shown in Equation 4.11 where $\theta$ is the heading angle.

$$\theta = \arctan(y/x) \tag{4.11}$$

However, this formula fails when x = 0 and gives same angle for when x,y = 1, 1 and when x,y = -1,-1 [31]. This can be fixed by using arctan2.

$$\theta = \arctan 2(y/x) \tag{4.12}$$

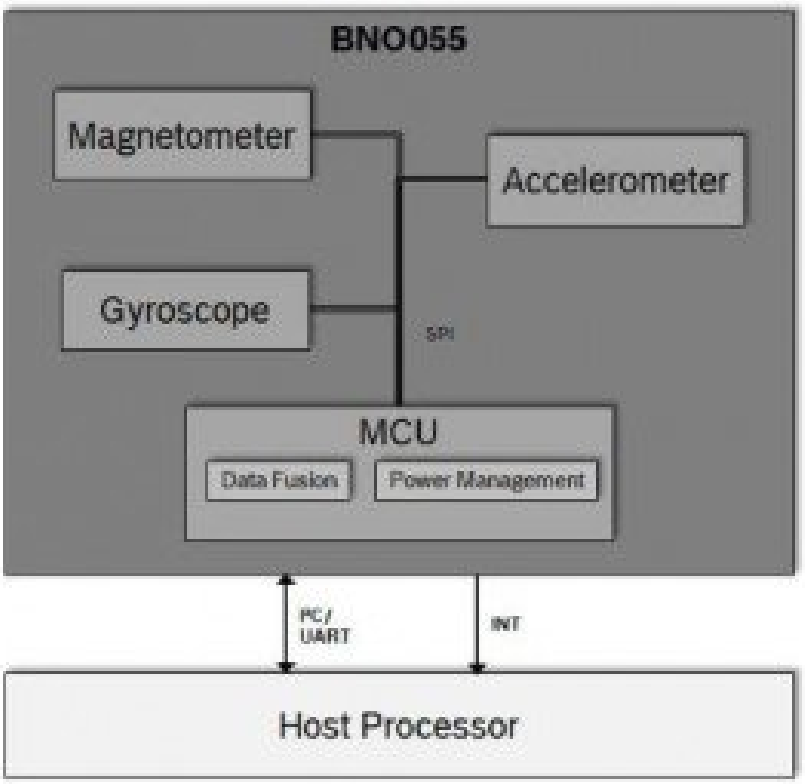


Figure 4.4: System Architecture of BNO055

This module communicates using I2C communication protocol. Once an I2C driver was written for MSP430, the values received from BNO055 were used to calculate heading angle using the Equation 4.12. However, this is still not tilt-compensated. To compensate for the tilt, an accelerometer needs to be added. Equations 4.13, 4.14, 4.15, can be used to get tilt-compensated heading angle. ax, ay, and, az were the x-axis, y-axis and z-axis values of the accelerometer in radians. Similarly, mx, my, and, mz were the x-axis, y-axis and z-axis values of the magnetometer [32]. To convert the accelerometer readings to radians, they were divided by 57.30.

$$xc = mx\cos(ay) + my\sin(ay)\sin(ax) - mz\cos(ax)\sin(ay) \tag{4.13}$$

$$yc = my\cos(ax) + mz\sin(ax) \tag{4.14}$$

$$\theta = \arctan 2(yc/xc) \tag{4.15}$$

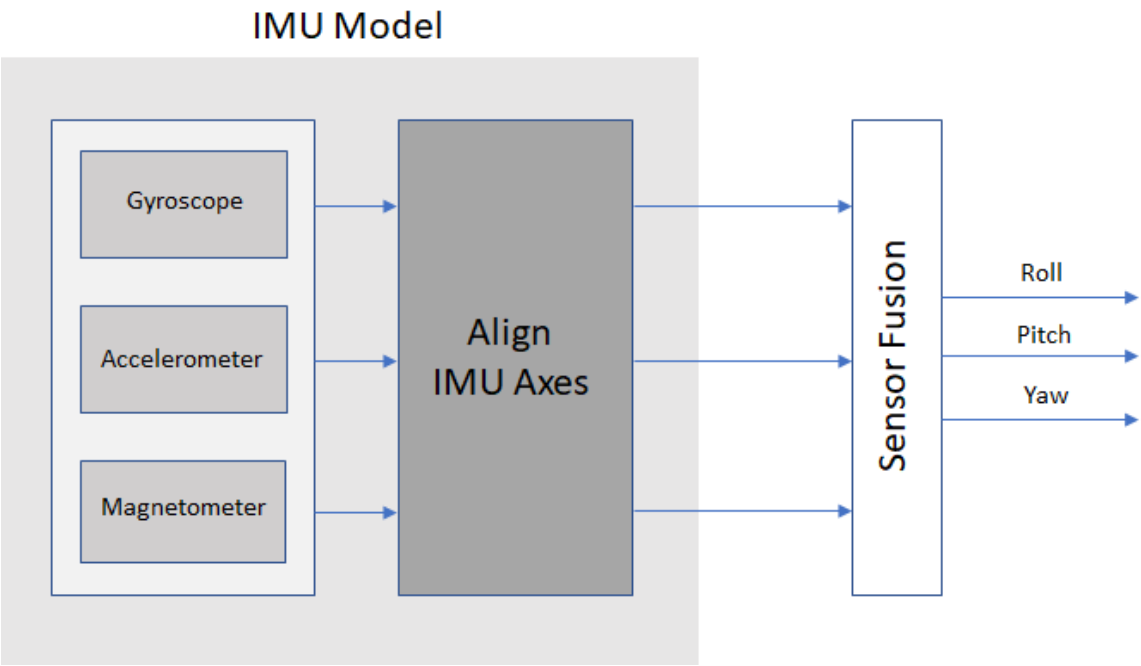


Figure 4.5: Sensor Fusion [21].

However, looking at the architecture of BNO055 in Figure 4.4, it already has a data fusion algorithm for fusing either all three of the sensors together or any two of them as shown in Figure 4.5.

While there are different fusion modes, they provide heading of the sensor in either Quaternion data or in Euler angles. This is used to get an absolute orientation of the module in space. The orientation can be either an absolute orientation or relative orientation in fusion mode. Absolute orientation means the position of the module with respect to the Earth and its magnetic field. This is used to calculate the heading angle to the magnetic North pole. Relative orientation gives out values based on how the module was initially placed. The fusion mode used is Nine Degrees of Freedom (NDOF) which has 9 degrees of freedom and fused absolute orientation data is calculated from accelerometer, gyroscope, and the magnetometer. This mode has the Fast Magnetometer calibration turned on which results in the quick calibration of the magnetometer and increases output data accuracy. This mode also comes with high robustness from magnetic field distortions. All this comes at the cost of slightly higher

current consumption when compared to other fusion modes. NDOF_FMC_OFF mode switches off fast magnetometer calibration and saves current.

### 4.2.1 Calibration

Although the BNO055's sensor fusion firmware runs the calibration algorithm for the accelerometer, gyroscope, and magnetometer to remove the offsets, the data-sheet recommends initial calibration steps to be taken when the device boots up. For the accelerometer, the data-sheet recommends to place the device in 6 different stable positions for a period of few seconds and move slowly between these stable positions to allow the accelerometer to calibrate. While the 6 positions can be in any direction, it is recommended that the device is lying perpendicular to either x-axis, y-axis or z-axis at least once. The gyroscope is the easiest to calibrate. It just needs to be kept stable for a few seconds for it to calibrate. The magnetometers, in general, are susceptible to both hard-iron and soft-iron distortions. The initial calibration is most important for the magnetometer as the accelerometer and the gyroscope are much less susceptible to disturbances. To get the proper heading angle, the magnetometer is calibrated at the start by making a few random moments or just making an '8' in the air. The raw magnetometer readings were taken from the BNO055 after a distortion was observed. The module was spun along the z-axis [33] and the x-axis and y-axis was plotted to see how calibration affects the magnetometer. The x and y reading as seen in Figure 4.6 was not centered but a slight shift towards the left and down was noticed. After this, the sensor was moved around in random 8 shapes until it was calibrated as suggested in the data sheet. The BNO055 stores the calibration status in a register called 'CALIB_STAT'. This is an 8-bit register whose register description is shown is Figure 4.7. As soon as the magnetometer is calibrated, the last two bits should display 1.

Once the magnetometer was calibrated, the same readings were taken again by spinning the module around the z axis. The plotted x and y reading seen in Figure

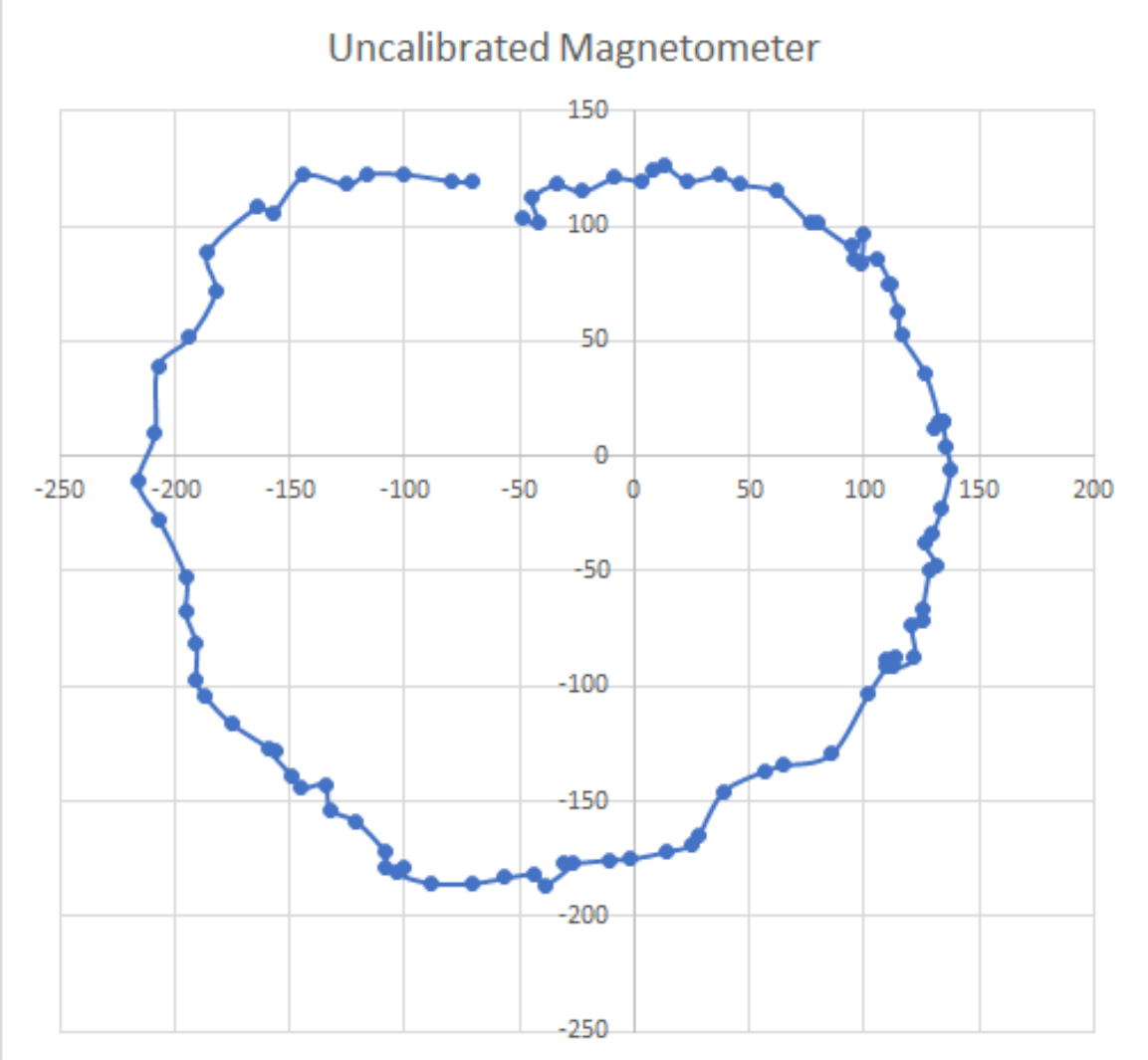


Figure 4.6: BNO055 Un-calibrated Magnetometer Readings

4.8 which were much more centered and good enough to use for calculating the heading angle. The data sheet also mentions that this calibration needs to be performed each time the module boots up. However, for use in an autonomous vehicle, a manual calibration on each power-up cannot be done. Upon reading the datasheet further, it was found that after calibration, the same calibration profile can be reused on each power up. The calibration profile is nothing but the sensor offsets and sensor radius registers. Once all the bits of the calibration status register were read as '1', i.e. not only accelerometer, gyroscope, and magnetometer are set but also the system

### 4.3.54 CALIB_STAT 0x35

| | bit7 | bit6 | bit5 | bit4 | bit3 | bit2 | bit1 | bit0 |
|---|---|---|---|---|---|---|---|---|
| **Access** | r | r | r | r | r | r | r | r |
| **Reset** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Content** | SYS Calib Status <0:1> | | GYR Calib Status <0:1> | | ACC Calib Status <0:1> | | MAG Calib Status <0:1> | |

| DATA | bits | Description |
|---|---|---|
| SYS Calib Status <0:1> | <7:6> | Current system calibration status, depends on status of all sensors, read-only<br>Read: 3 indicates fully calibrated; 0 indicates not calibrated |
| GYR Calib Status <0:1> | <5:4> | Current calibration status of Gyroscope, read-only<br>Read: 3 indicates fully calibrated; 0 indicates not calibrated |
| ACC Calib Status <0:1> | <3:2> | Current calibration status of Accelerometer, read-only<br>Read: 3 indicates fully calibrated; 0 indicates not calibrated |
| MAG Calib Status <0:1> | <1:0> | Current calibration status of Magnetometer, read-only<br>Read: 3 indicates fully calibrated; 0 indicates not calibrated |

Figure 4.7: BNO055 Calibration Status register description [34].

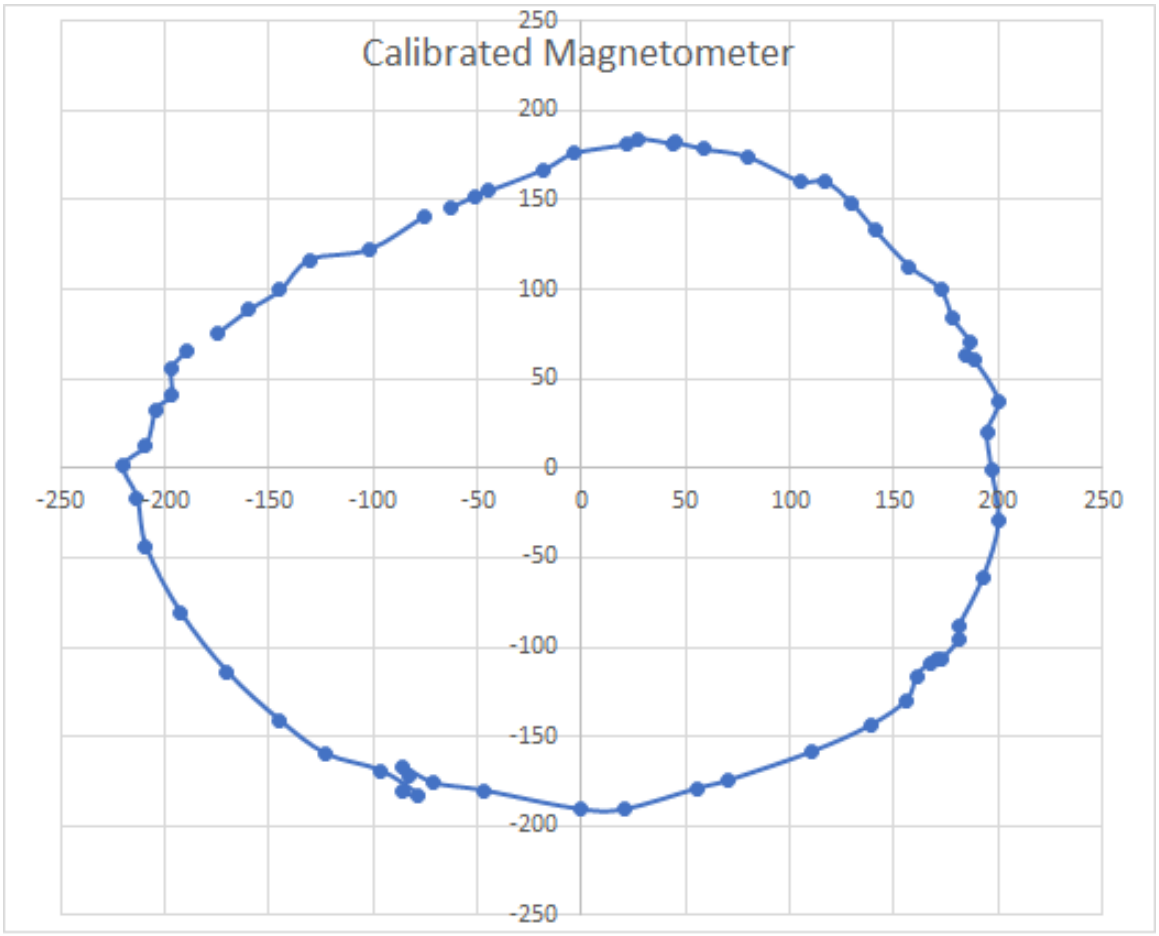


Figure 4.8: BNO055 Calibrated Magnetometer Readings

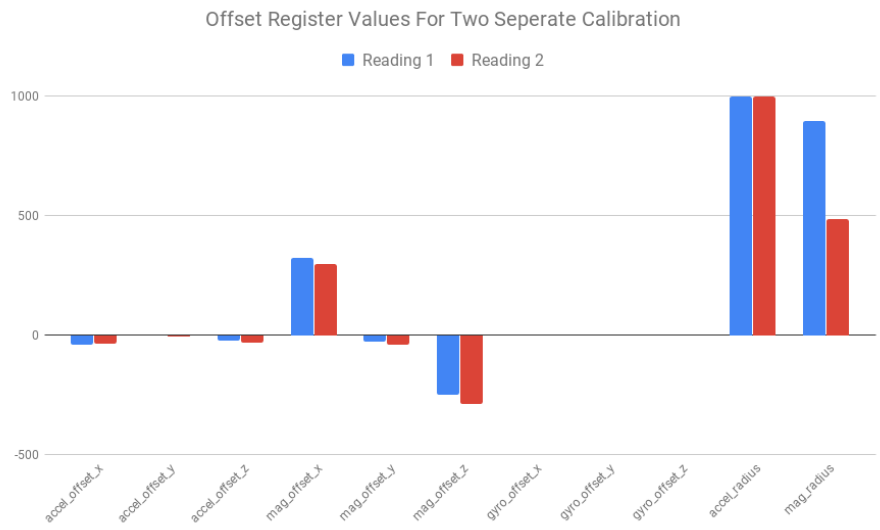


Figure 4.9: Offset And Radius Register Readings After Calibration

was calibrated, the offset registers and radius registers were read. The BNO055 needs to be switched from operation mode to configuration mode before this can be done. The three sensors have their offset registers for each axis. Thus a total of 9 8-bit offset registers and 4 radius registers were read. The values from these registers are stored in the EEPROM or could just be hard coded so that they were set on each power up. The values received after reading these 13 registers stayed nearly the same on each reading when the module was kept in a stable position after calibration except for the magnetometer registers. This can be seen in the Figure 4.9. The accelerometer and gyroscope offset registers remain almost the same. The gyroscope readings were single digit hence not visible in the graph. However, there's a change seen in the magnetometer offset readings and especially the radius reading. This suggests that although saving and reading the offset registers will work fine in a normal power up case, but, when the magnetometer will come across a distortion, the saved magnetometer offset values won't be useful anymore. Luckily though once the BNO055 internal calibration routine is done, it overwrites the offsets and radius. However, the readings before that cannot be trusted.

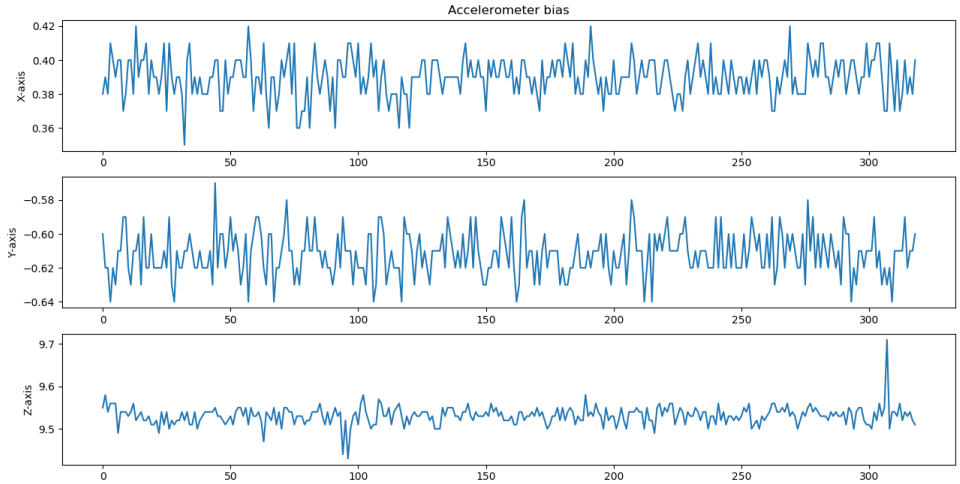


Figure 4.10: Accelerometer Bias Reading.

### 4.2.2 Sensor Bias

The raw readings were taken from the IMU sensor and plotted while it was stationary. When the module is kept stationary on a flat surface, gravity is the only acceleration which is 9.8g, there is no angular velocity and the magnetic field is static. The output can be seen in Figures 4.10, 4.11, 4.12. Small offsets on either

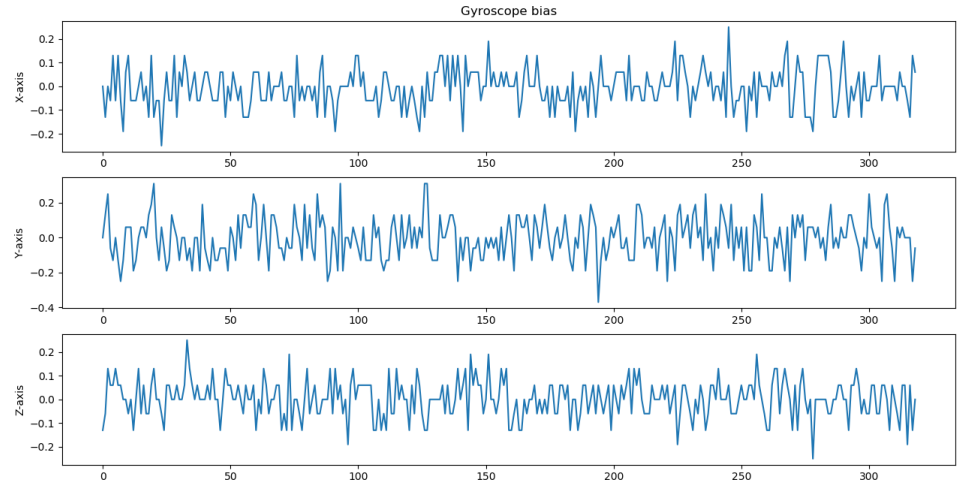


Figure 4.11: Gyroscope Bias Reading.

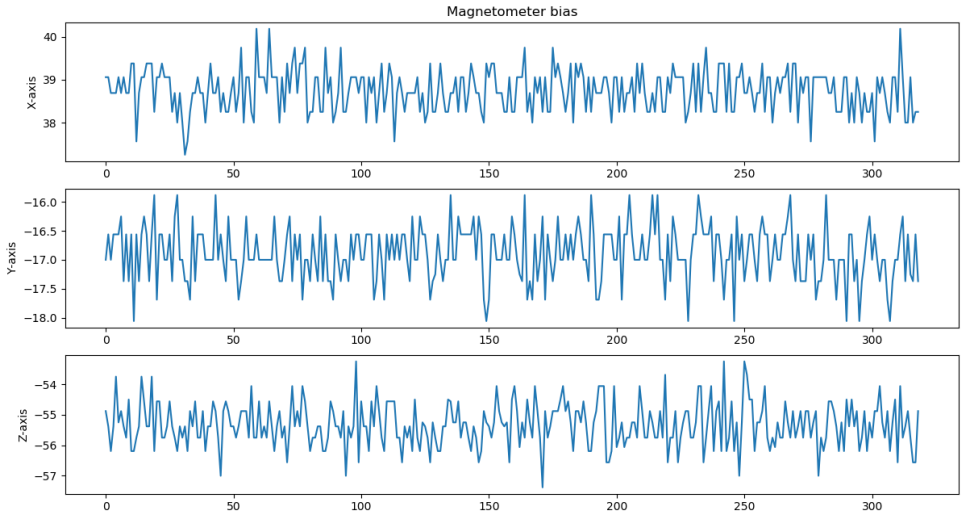


Figure 4.12: Magnetometer Bias Reading.

side on an average were noticed even after calibration. These offsets are called sensor biases.

### 4.2.3 Euler vs Quaternion

Initially, the plan was to use Euler angles that the BNO055 outputs in the NDOF mode it is set in and check for bias. The Euler angle values were read from the MSB and LSB registers of x-axis, y-axis, and z-axis. Where x-axis is the heading angle, y-axis is the roll and z-axis is the pitch. Looking at Figure 4.13, yaw or heading angle is the rotation about z-axis i.e it increases if moved clockwise, pitch represents rotation about the y-axis, and roll is the rotation about the x-axis. It is very important to calibrate before reading these registers. If the IMU is not calibrated, it considers the current position as 0° and thus gives relative heading instead of absolute heading angle which is needed. As can be seen from the bias readings in Figure 4.14, the data received is very stable. The sensor fusion algorithm in the BNO055 makes sure the bias offsets were removed and stable readings were received. The Yaw or the heading angle reading received in the Figure is 242°. This angle was verified with a compass

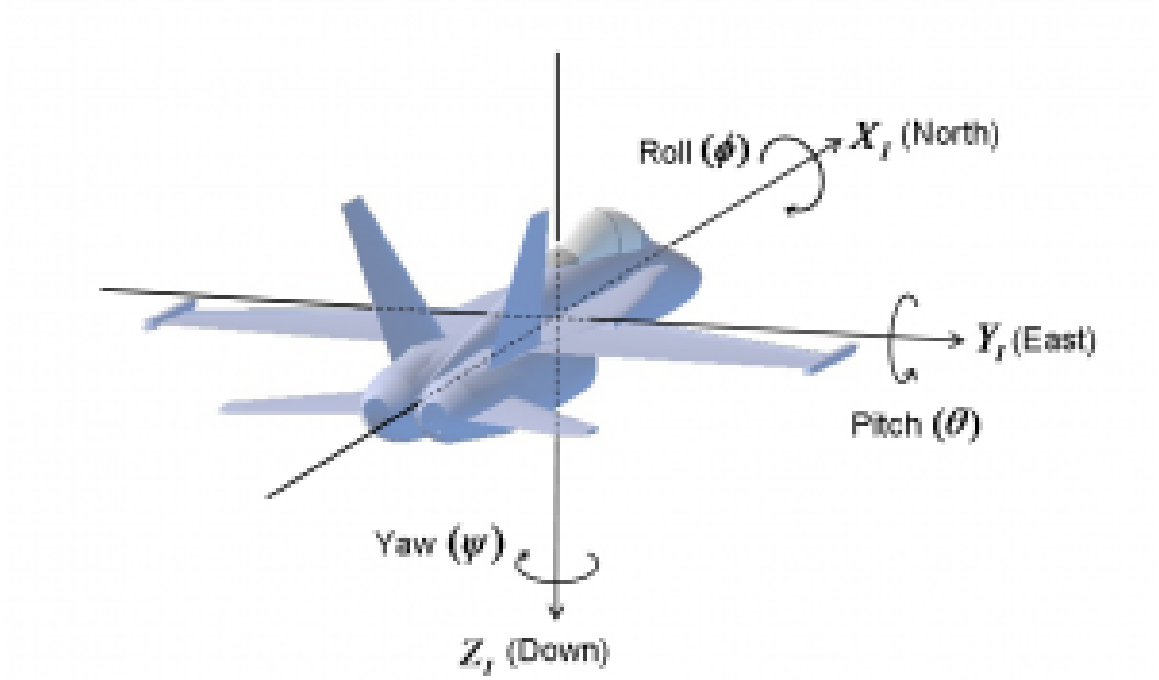


Figure 4.13: Euler Angle Axes [35].

reading on a personal mobile phone which displayed 250°. This is due to magnetic declination. While the GPS NMEA messages ideally have the magnetic declination for the location it measures in the GPRMC packet, it was found that Adafruit's GPS module disables it. The declination angle across the US can be seen in Figure 4.15. However, magnetic declination changes only by a few degrees every year. Thus, it can be used directly for now instead of replacing the GPS to get declination from it. Using the declination calculator on National Oceanic and Atmospheric Organization [37], the declination received was 7.82° W for Charlotte, NC. It was also mentioned that this values change by about 0.03° per year. Since this declination is to the West of true North, it must be compensated by adding the declination to the heading angle. If the declination was to the East of true North, it must be subtracted from the heading angle. Hence, the heading angle becomes 242 + 7.82 = 249.82° which is what the phones compass displays. During testing, the IMU would not display the same angle at the exact same position sometimes. Upon reading more about

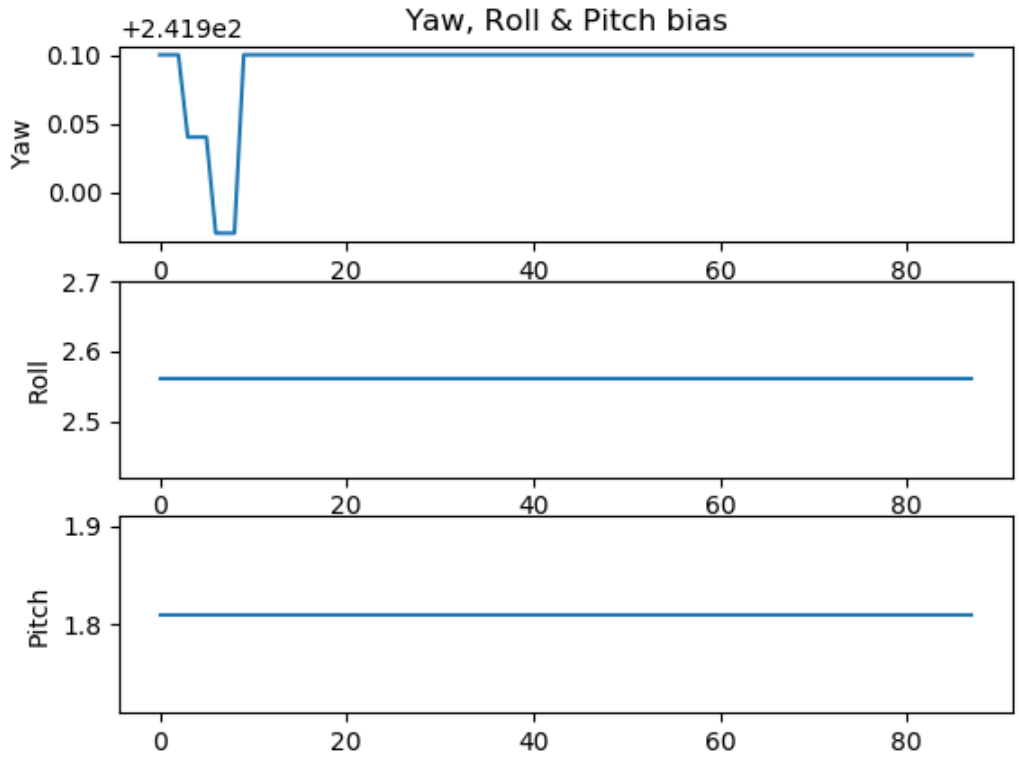


Figure 4.14: Euler Bias Reading.

Euler angles [35], the case of gimbal locking came forward. For Euler angles to give correct representation, they need to be at least partially perpendicular. The Euler angle rotation matrix in 3 dimensions is represented by the Equation 4.16 where R is the rotation matrix, x is the roll, y is the pitch and z is the yaw. However, consider a condition where the pitch is up by an angle of 90°. The yaw angle now becomes parallel to the roll angle. Equation 4.17 shows the rotation matrix when pitch is 90, which converts to Equation 4.18 where the output for changes in x and z results in the same matrix.

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix} * \begin{bmatrix} \cos(y) & 0 & \sin(y) \\ 0 & 1 & 0 \\ -\sin(y) & 0 & \cos(y) \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix} \quad (4.16)$$

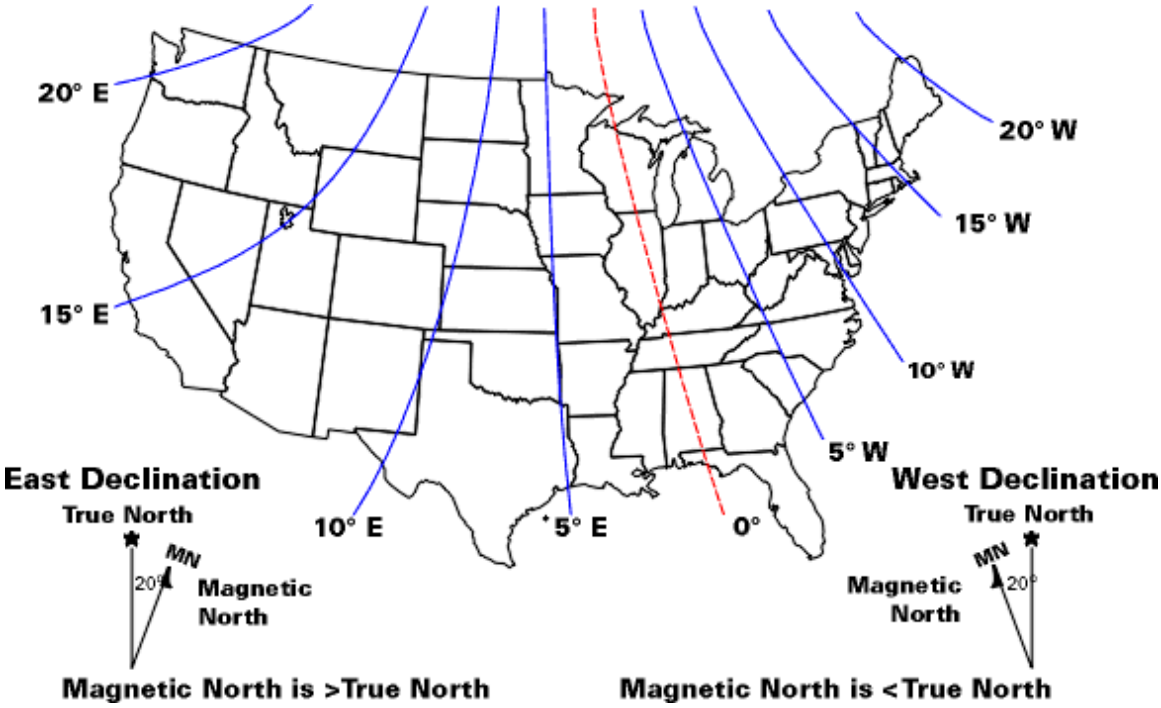


Figure 4.15: Declination Angles Across USA [36].

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix} * \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(x) & -\sin(x) \\ 0 & \sin(x) & \cos(x) \end{bmatrix} \tag{4.17}$$

$$R = \begin{bmatrix} 0 & 0 & 1 \\ \sin(x+z) & \cos(x+z) & 0 \\ -\cos(x+z) & \sin(x+z) & 0 \end{bmatrix} \tag{4.18}$$

Readings for the Euler angles were taken again to make sure this was the issue as seen in Figure 4.16. The Yaw angle drops suddenly after a point. However, the BNO055 does not fail when the pitch angle was 90° but when the angle went to -135°. This is a math bug in the firmware 0x0311 of the BNO055 and Adafruit suggests not to go beyond +/-45°. Although it was initially thought the change in reading was due to gimbal locking in in BNO055, it became apparent that was actually due to a math bug in BNO055's firmware. Due to this, it was decided to use Quaternion angles instead.

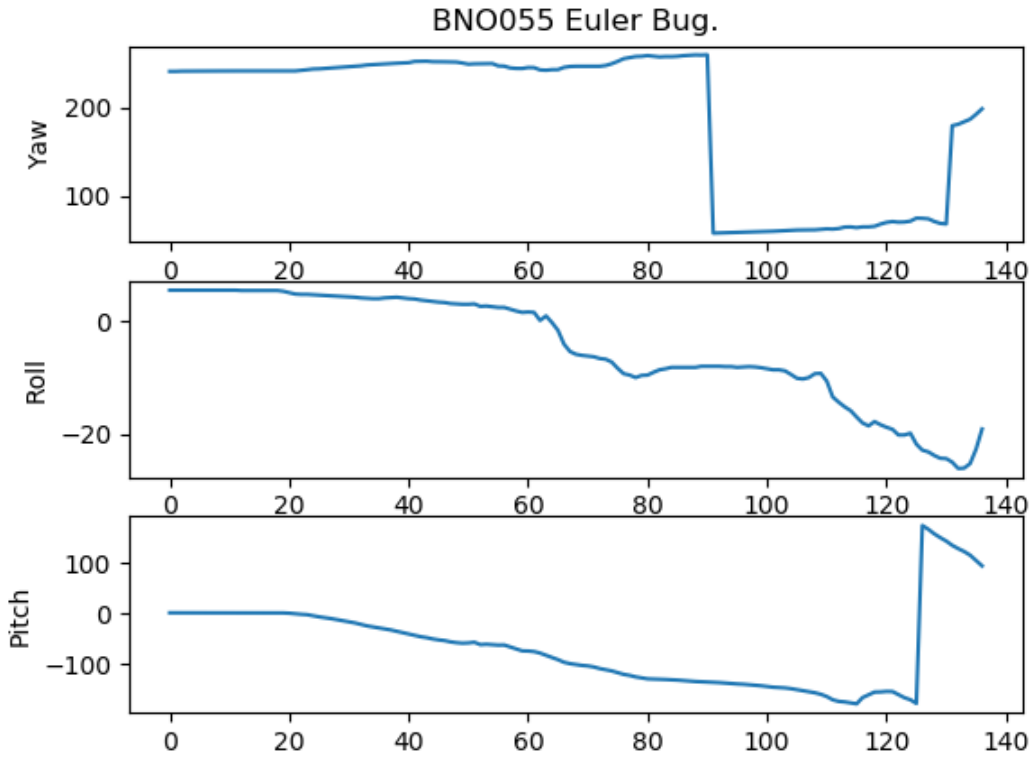


Figure 4.16: Euler Angle BNO055 Bug.

Quaternion is a four-dimensional complex number which consists of one scalar and three vector components. This is shown in Equation 4.19 where w, x, y, and z are real numbers and i, j and k are Quaternion units [38].

$$q = (w, xi, yj, zk) \tag{4.19}$$

The real numbers are normally in the range of -1 to 1. The absolute values of these when added together equal to 1.

$$\sqrt{w^2 + x^2 + y^2 + z^2} = 1 \tag{4.20}$$

The Quaternion matrix can be converted to a rotational matrix using the Equation

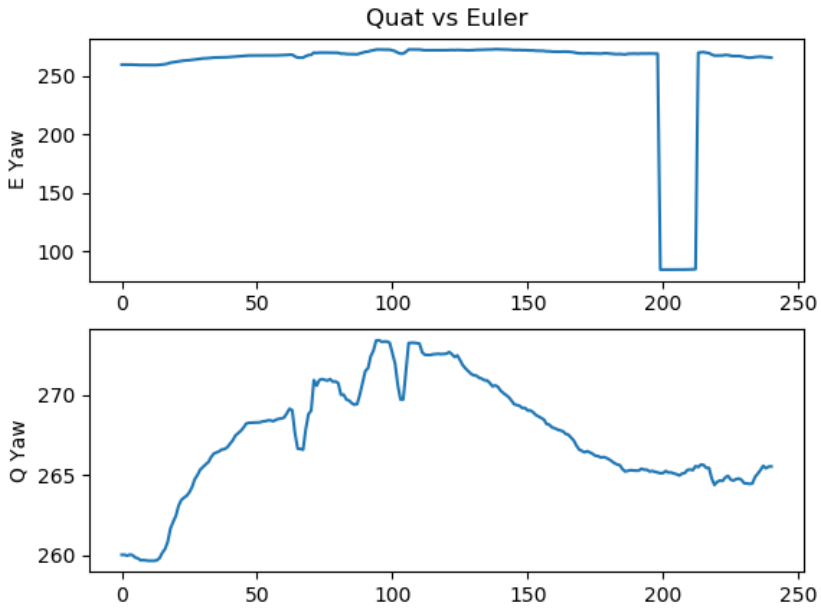


Figure 4.17: Comparing Quaternion with Euler.

4.21 [39].

$$R = \begin{bmatrix} w^2 + x^2 + y^2 + z^2 & 2(xy - wz) & 2(wy + xz) \\ 2(xy + wz) & w^2 - x^2 + y^2 - z^2 & 2(-wx + yz) \\ 2(-wy + xz) & 2(wx + yz) & w^2 - x^2 - y^2 + z^2 \end{bmatrix} \tag{4.21}$$

To get yaw, pitch, and roll from Quaternion data, Equations 4.22, 4.23 and 4.24 were used.

$$roll = \arctan 2(2 * (w * x + y * z), 1 - 2 * (x * x + y * y)) \tag{4.22}$$

$$pitch = \arcsin(2 * w * y - x * z) \tag{4.23}$$

$$yaw = \arctan 2(2 * (w * z + x * y), 1 - 2 * (y * y + z * z)) \tag{4.24}$$

Using these Equations, the values from the Quaternion registers were read and converted to yaw, pitch and roll. To check for the math bug that Euler readings had, the module was again moved by -135° pitch and Quaternion angles still displayed the

expected values as can be seen in Figure 4.17.

## 4.3 GPS and IMU together

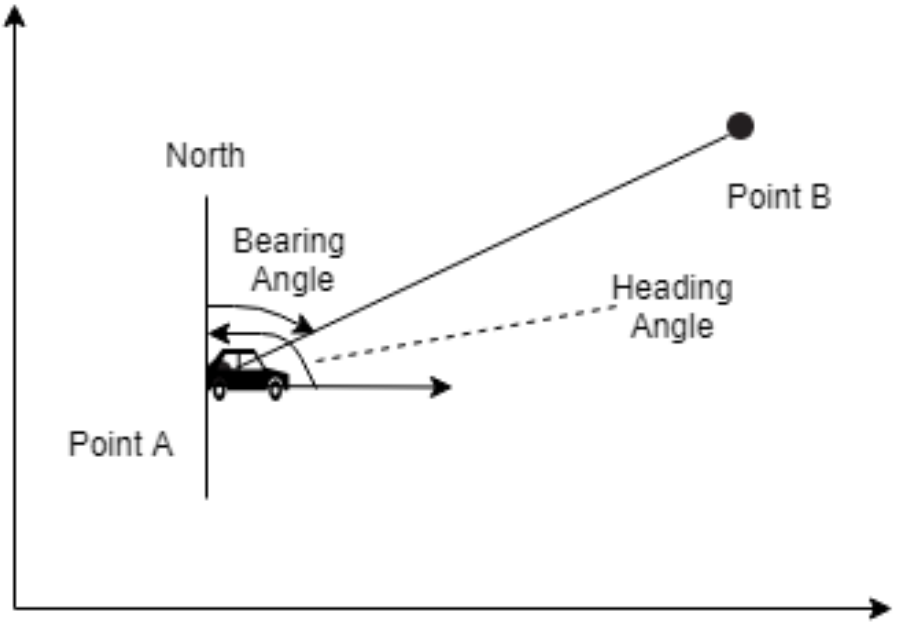


Figure 4.18: Illustration of heading angle and bearing angle.

Once data from GPS and INS is received, the next step is to make sense of it. The GPS provides latitude and longitude points for the ATV. The latitude and longitude points for the destination is also know. Using these two points, distance is calculated using Equation 4.6, and bearing angle using the Equation 4.7. The bearing angle is the angle that the ATV needs to point to, to get to the next way-point. Heading angle is received from the IMU. The heading angle indicates the angle the ATV is pointing relative to true North. From these two angles, the angle the vehicle needs to move to can be calculated. The illustration of this can be seen in Figure 4.18. Point A is the point the ATV is at; i.e. the latitude and longitude from the GPS module in the ATV. Point B is the destination and where the ATV needs to go to. Here, to figure out the angle to steer to, the Equation 4.25.

$$SteeringAngle = BearingAngle - HeadingAngle. \tag{4.25}$$

Negative steering angle indicates that the ATV should steer left and a positive angle indicates that the ATV should turn right.

## 4.4 Kalman Filtering

The data received from the GPS contains noise as seen in Figure 5.3. GPS readings were taken along a trail and the actual path compared to the GPS readings was plot as shown in 4.19. The GPS readings looked a bit noisy and jumpy around the expected path line. Averaging these reading would just add the noise into the readings. Moreover, this would slow down the whole process as the controller would now take several readings and then output the averaged reading. This is undesirable. It also does not solve the problem of not having GPS readings at regular distance intervals on the path as can be seen in the Figure 4.19. To rectify this, IMU readings can be used to estimate the position using the Equation of motion 4.26 where x is the position, v is the velocity in meter per seconds, $\Delta t$ is the time interval, and a is the acceleration per second squared. The Equation 4.27 is used for getting velocity.

$$x' = x + v\Delta t + \frac{1}{2}a\Delta t^2 \tag{4.26}$$

$$v' = v + a\Delta t \tag{4.27}$$

However, the IMU itself consists of a lot of noise as Figures 4.10, 4.10, and 4.10 show. Using just either of the IMU and GPS sensor by itself is not enough to correct the readings. The accelerometer requires initial position and velocity to estimate the next point and correct GPS errors. The output from the accelerometer can be read at a considerably higher frequency than the GPS readings. However, the noise associated with the accelerometer indicates that using just the acceleration would cause the noise to accumulate and this will cause drift. Due to this, after a few seconds, the position estimate will be quite off from the actual position. Thus, GPS readings are required to avoid drift errors and the accelerometer readings are required to correct GPS noise.

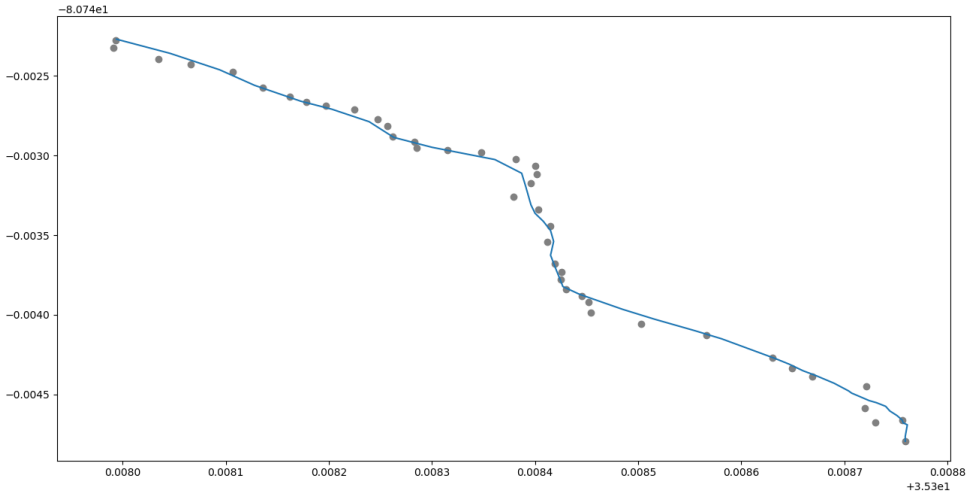


Figure 4.19: Actual path compared to the GPS readings.

This can be done by fusing the two sensor data. Upon reading more about it, the most popular technique for position based sensor fusion was using a Kalman filter [40, 41, 42]. Kalman filter can be used when there is uncertain information about some dynamic system, and a good estimate needs to be made about the next step. Kalman filter uses mathematical models to fuse error-filled real-time measurements from various sensors to get an estimate of the data needed. The future estimate data is based on the current state and changes in the sensor values at the current timestamp. It then compares the estimated state with the data received and adjusts its own estimate [43].

## 4.5 Kalman Model

Initially, the system is represented by matrices A, B, and H. State vector X gives an initial state estimate of the system. Initial error estimate of the system is given by matrix P. Error estimates for process and measurement are given by matrices Q and R respectively [43]. The Kalman filter works recursively in two steps: (1) Prediction and (2) Update shown in Figure 4.20. At each step, control vector 'u' and measurement vector 'z' are passed into the filter. The Kalman then outputs current estimate of the

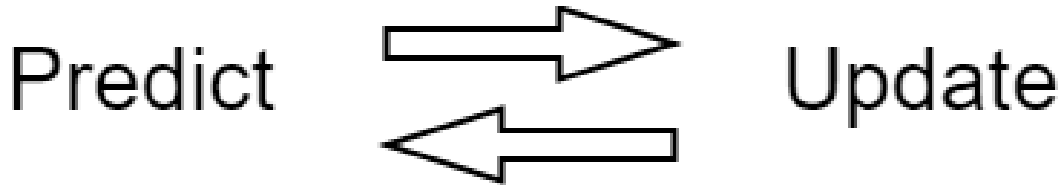


Figure 4.20: Two Step Kalman Process

state and current estimate of the error in the system.

### 4.5.1 Prediction

The first step in prediction is to predict the next state. The state of the Kalman filter is given by the Equation 4.28 where $X_{\text{predicted}}$ is the predicted state, A is the state transition matrix, $X_{\text{n-1}}$ is the previous state at time step n-1, B is the control matrix, and $u_{\text{n}}$ is the control vector at time step n.

$$\mathrm{X}_{\text{predicted}} = \mathrm{A}\mathrm{X}_{n-1} + \mathrm{B}\mathrm{u}_{n} \tag{4.28}$$

The next step is to predict how much error is there in the system. The covariance prediction matrix at n is given by the Equation 4.29 where A is the state transition matrix, $P_{n-1}$ is the covariance matrix at the previous time step n-1, $A^T$ is the transpose of state transition matrix, and Q is the process variance matrix.

$$\mathrm{P}_{\text{predicted}} = \mathrm{A}\mathrm{P}_{n-1}\mathrm{A}^{\mathrm{T}} + \mathrm{Q} \tag{4.29}$$

### 4.5.2 Update

During the update step, the Kalman gain is updated to moderate the prediction. It indicates the amount of trust on a particular sensor. This is given by the Equation 4.30 where K is the Kalman gain, $P_{\text{predicted}}$ is the predicted covariance, $H^T$ is the transpose of the observation matrix, and $S^{-1}$ is the inverse of innovation covariance matrix. This innovation covariance matrix is given by the Equation 4.31 where R is

the estimated measurement error covariance matrix.

$$\mathrm{K} = \mathrm{P}_{\mathrm{predicted}}\mathrm{H}^{\mathrm{T}}\mathrm{S}^{-1} \tag{4.30}$$

$$\mathrm{S} = \mathrm{H}\mathrm{P}_{\mathrm{predicted}}\mathrm{H}^{\mathrm{T}} + \mathrm{R} \tag{4.31}$$

To get new estimate of the state of the system, the state matrix is updated using the Equation 4.32 where $\tilde{y}$ is the innovation matrix and is given by the Equation 4.33. This is Kalman filtered out estimate. Here, $z_n$ is the measurement vector.

$$\mathrm{X}_n = \mathrm{X}_{\mathrm{predicted}} + \mathrm{K}\tilde{y} \tag{4.32}$$

$$\tilde{y} = \mathrm{z}_n - \mathrm{H}\mathrm{X}_{\mathrm{predicted}} \tag{4.33}$$

Finally, the error covariance matrix is updated using the Equation 4.34 to get new error estimate. Where 'I' is an identity matrix.

$$\mathrm{P}_{\mathrm{n}} = (\mathrm{I} - \mathrm{K}\mathrm{H})\,\mathrm{P}_{\mathrm{predicted}} \tag{4.34}$$

The flow of the filtering can be seen in Figure 4.21. To sum it all up,

$u$: Control Vector (Input)

$z$: Measurement Vector (Input)

$X$: State Estimate (Output)

$P$: Error covariance Estimate For State (Output)

$A$: State Transition Matrix

$B$: Control Matrix

$H$: Observation Matrix

$Q$: Process Error covariance Estimate

$R$: Measurement Error covariance Estimate

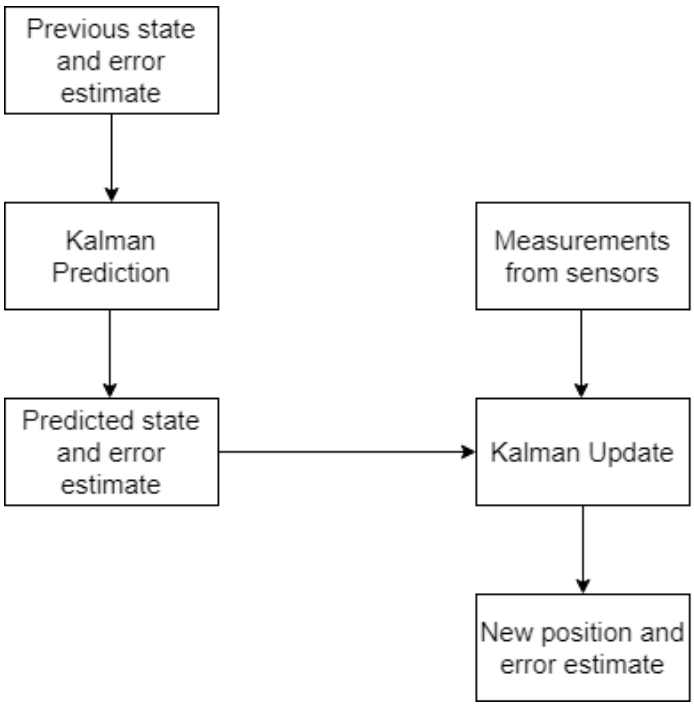


Figure 4.21: Flow of Kalman filtering

$S$: Innovation covariance

$y$: Innovation Matrix

## 4.6 Implementation

Implementing the Kalman filter for this project meant identifying what the various matrices mentioned in the previous section would translate to. Looking at the Kalman Equations again, for Equation 4.28, $X_n$ would denote the predicated state of the ATV, A would be matrix that predicts the new state using Equation of motion as given in 4.26. $X_{n-1}$ denotes the state of the ATV at a previous time step n-1. B is the control matrix that predicts the change in motion. $u_n$ is the acceleration input at time n. To predict the error covariance in 4.29, A is the same as the previous Equation, $P_{n-1}$ the previous state error estimate, $A^T$ is the transposed matrix of A. Q is the error covariance for noise in the accelerometer readings. It is the standard distribution of noise of the accelerometer data. R is the covariance matrix for error estimate in the GPS readings. u is the acceleration input and z is the GPS input. Raw

acceleration data consists of linear acceleration and gravity acceleration. To strip out the acceleration due to gravity the Quaternion output from the BNO055 is used. The gravity component is calculated using the Equations 4.35, 4.36, and 4.37 where gx, gy, and gz are the gravity component.

$$g_x = (2.0 * (q.y * q.w - q.x * q.z)) \tag{4.35}$$

$$g_y = (2.0 * (q.x * q.y + q.y * q.z)) \tag{4.36}$$

$$g_z = (q.x * q.x - q.y * q.y - q.z * q.z + q.w + q.w) \tag{4.37}$$

Linear acceleration can be obtained by subtracting the raw acceleration with these gravity components as seen in Equations 4.38, 4.38, and 4.38.

$$acc_{lx} = (-1 * (a.x + g_x)) \tag{4.38}$$

$$acc_{ly} = (-1 * (a.y - g_y)) \tag{4.39}$$

$$acc_{lz} = (-1 * (a.y - g_z)) \tag{4.40}$$

The BNO055 however, in sensor fusion mode, provides the two acceleration data in separate registers. LIA registers hold data for linear acceleration and GRV registers hold data for gravity acceleration as seen in Figure 4.22. The received data is divided by 100 to as $1m/s^2$ is equal to 100LSB. This acceleration is however relative to the body of the ATV and not Earth-referenced. To convert these linear acceleration data to Earth-referenced data in the North-East-Down (NED) reference frame, the Equations 4.41, 4.42, and 4.43 were used. Since it is in NED frame, the up acceleration

| Parameter | Data type | bytes |
|---|---|---|
| LIA_Data_X | signed | 2 |
| LIA_Data_Y | signed | 2 |
| LIA_Data_Z | signed | 2 |

| Unit | Representation |
|---|---|
| $m/s^2$ | 1 $m/s^2$ = 100 LSB |
| mg | 1 mg = 1 LSB |

(a) Linear Acceleration Register

| Parameter | Data type | bytes |
|---|---|---|
| GRV_Data_X | signed | 2 |
| GRV_Data_Y | signed | 2 |
| GRV_Data_Z | signed | 2 |

| Unit | Representation |
|---|---|
| $m/s^2$ | 1 $m/s^2$ = 100 LSB |
| mg | 1 mg = 1 LSB |

(b) Gravity Acceleration Register

Figure 4.22: LIA and GRV registers as shown in BNO055 datasheet. [34]

is multiplied by '-1'.

$$\begin{aligned} acc_{east} = (1.0 - 2.0(q.y \times q.y + q.z \times q.z)) \times acc_{lx} \\ +2.0(q.x \times q.y - q.w \times q.z) \times acc_{ly} \\ +2.0(q.x \times q.z + q.w \times q.y) \times acc_{lz} \end{aligned} \tag{4.41}$$

$$\begin{aligned} acc_{north} = (2.0(q.x \times q.y + q.w \times q.z)) \times acc_{lx} \\ +(1.0 - 2.0(q.x \times q.x + q.z \times q.z)) \times acc_{ly} \\ +2.0(q.y \times q.z - q.w \times q.x) \times acc_{lz} \end{aligned} \tag{4.42}$$

$$\begin{aligned} acc_{up} = -1 \times (2.0(q.x \times q.z - q.w \times q.y)) \times acc_{lx} \\ +2.0(q.y \times q.z + q.w \times q.x) \times acc_{ly} \\ +(1.0 - 2.0(q.x \times q.x + q.y \times q.y)) \times acc_{lz} \end{aligned} \tag{4.43}$$

The North and East components need to be compensated for the magnetic declination. Sin and Cos components of the declination again were extracted. These components were then multiplied by the acceleration North and East values to get North and East component of both the acceleration values using the Equations 4.44, 4.45, 4.46, and 4.44. The North and East components of both the axes were then added to get the final declination compensated absolute acceleration values as seen in the eqauations 4.48, and 4.49.

$$dec_{easternN} = \sin(7.82 \times (PI/180)) \times acc_{east} \tag{4.44}$$

$$dec_{northernE} = -\sin(7.82 \times (PI/180)) \times acc_{north} \tag{4.45}$$

$$dec_{northernN} = \cos(7.82 \times (PI/180)) \times acc_{north} \tag{4.46}$$

$$dec_{easternE} = \cos(7.82 \times (PI/180)) \times acc_{east} \tag{4.47}$$

$$acc_{east} = dec_{easternE} + dec_{easternN} \tag{4.48}$$

$$acc_{north} = dec_{northernE} + dec_{northernN} \tag{4.49}$$

The sensor was kept stationary and their linear acceleration output was read and converted to the absolute acceleration using the Quaternion output. This data was then plotted to get the bias of the system as shown in Figure 4.23. Once the desired values were received from the GPS and IMU, it was time to run the Kalman filter on them. Having one Kalman filter for all the axes combined would lead to 3 times the size of the matrices and would be complicated to debug. Instead, it was chosen to apply a Kalman filter individually on each axis. For predicting latitude, the absolute acceleration in the North direction is used. For predicting longitude,

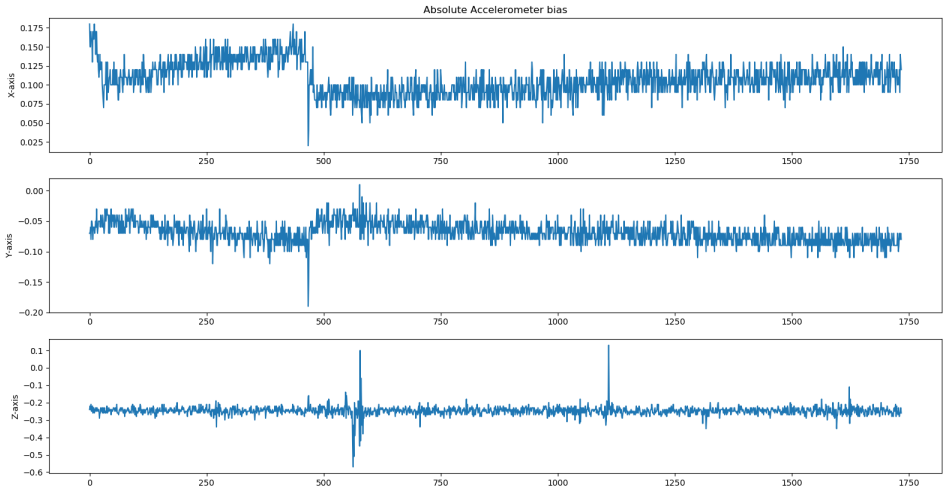


Figure 4.23: Absolute Acceleration Bias

the absolute acceleration in the East direction is used. The absolute acceleration in the up direction is used to predict the altitude. For this thesis, the measurement is the GPS and control is the accelerometer. The control is used to estimate positions and the measurement is used to update. The Kalman filter code waits for a valid GPS reading before it initializes the three Kalman filters. Once a GPS reading is received, matrix 'X', i.e. the state matrix, is initialized with GPS position readings. The matrix 'Q', i.e. the error covariance matrix for the process is initialized with the standard deviation received from bias readings of each accelerometer axis reading. The standard deviation of the GPS cannot be measured in the same way that the accelerometer standard deviation was received. This is due to the fact that if GPS was kept stationary, it will acquire a better fix over time. For this reason, the GPS standard deviation was received from the data-sheet. This was then initialized into the 'R' matrix which is the measurement covariance matrix. The process covariance matrix was initialized as an identity matrix, to begin with.

During each prediction step, accelerometer reading for each axis was passed into the separate Kalman filter. The state matrix and error matrix was updated during

this step with the time and accelerometer. During the update step, the GPS input was converted to meters and passed into the 'z' matrix and state and state covariance matrices were updated. The predicted estimate was stored in the state matrix. To predict latitude and longitude from the predicted position estimate, the Equations 4.50 and 4.51 were used [44]. Since North and East directions were considered positive and South and West are negative, the East and then the NorthEast component were calculated. To calculate the East position, the predicted longitude in meters was passed into the Equations 4.50 and 4.51 as distance, and the bearing angle used was 90 degrees as it was the East direction. This gives latitude and longitude points in the East. These points were then passed into the same Equations with a distance which is the predicted longitude in meters and a bearing angle of 0 degrees to point to the North. The resultant latitude and longitude were the final estimated points.

$$\varphi_2 = \operatorname{asin}\left(\sin\varphi_1 \cdot \cos\delta + \cos\varphi_1 \cdot \sin\delta \cdot \cos\theta\right) \tag{4.50}$$

where:

$\varphi$: Latitude

$\delta$: d/R; d is the distance traveled and R is the Earth's radius.

$\theta$: Bearing angle.

$$\lambda_2 = \lambda_1 + \operatorname{atan}2\left(\sin\theta \cdot \sin\delta \cdot \cos\varphi_1, \cos\delta - \sin\varphi_1 \cdot \sin\varphi_2\right) \tag{4.51}$$

where:

$\varphi$: Latitude

$\lambda$: Longitude

$\delta$: d/R; d is the distance traveled and R is the Earth's radius.

$\theta$: Bearing angle. Due to the fact that the Kalman filter operates on matrices and since the data-type of GPS and accelerometer readings were of the datatype double,

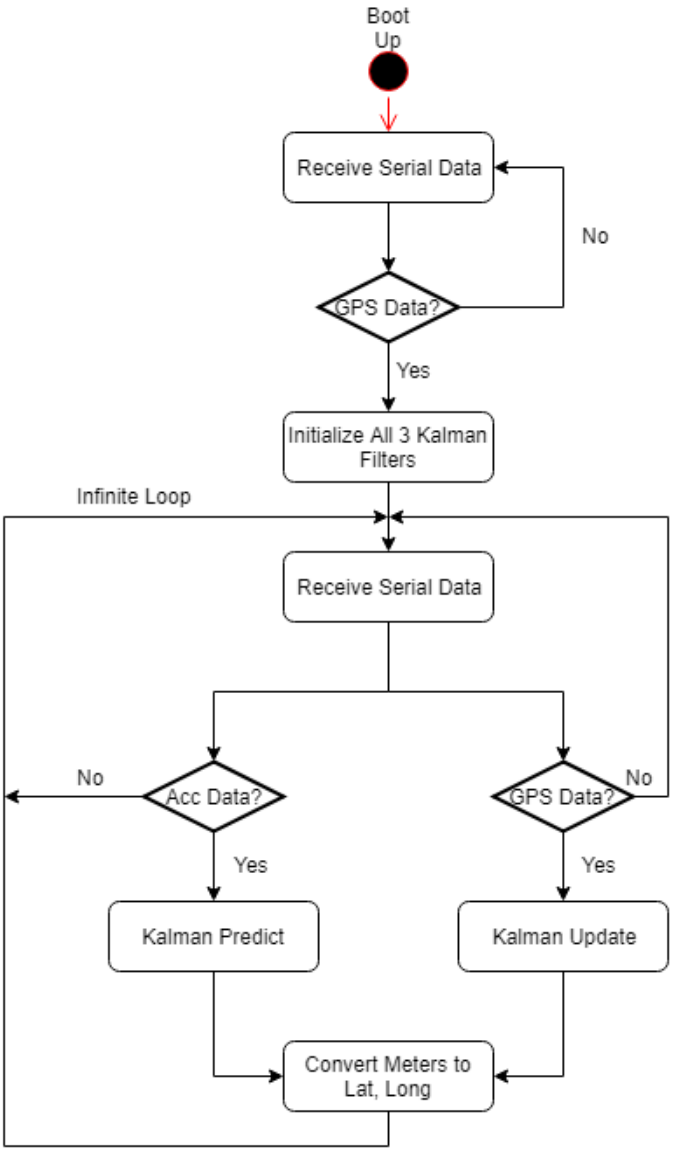


Figure 4.24: Kalman filter Implementation Flow.

there were doubts whether an MSP430 could run it recursively. Since there was already a bit of math done in the MSP430, it was chosen to implement the Kalman filter on a Raspberry Pi 3. The flow for the code can be seen in Figure 4.24.

# CHAPTER 5: TESTING AND RESULTS

To start the testing, the GPS module was kept stationary just outside the EPIC building and powered on. On average the module took about 25 seconds after booting up to start giving readings. However, the readings obtained were still a bit noisy. The GPS device gives GPS reading quality indicator in GPGGA messages. When there were no GPS readings, the quality indicator returns 0. As the readings start, i.e. a fix is obtained, the quality could be either a standard position fix (1) or a differentially corrected position fix (2). Differentially corrected position indicates

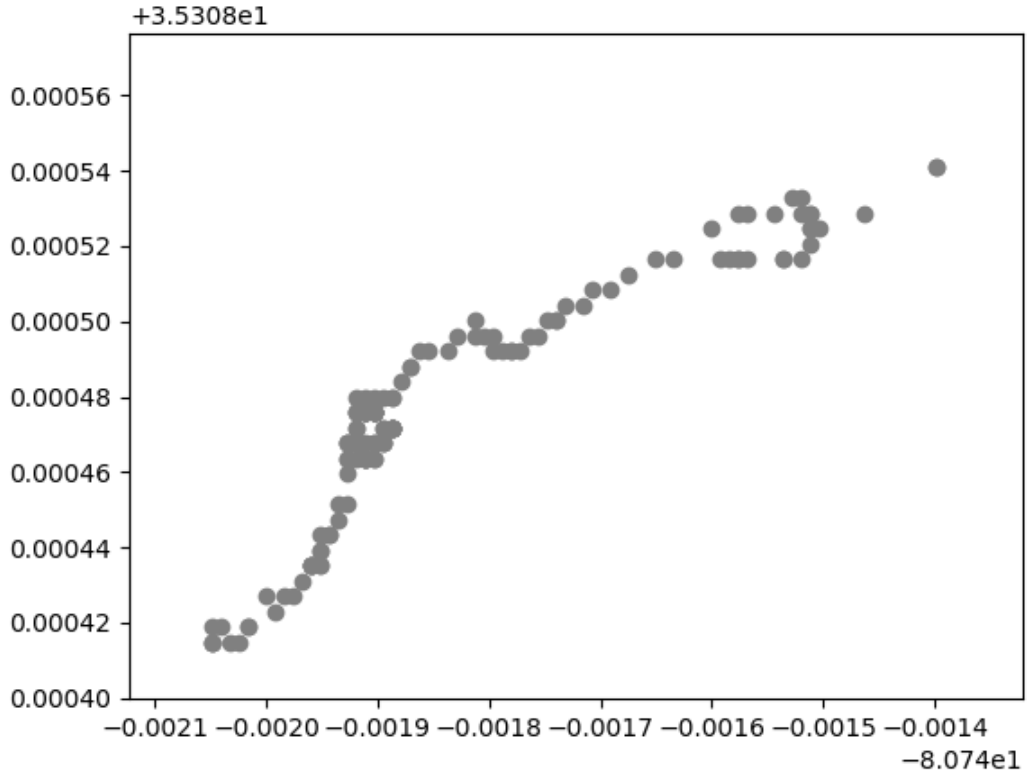


Figure 5.1: Latitude and Longitude with a standard fix.

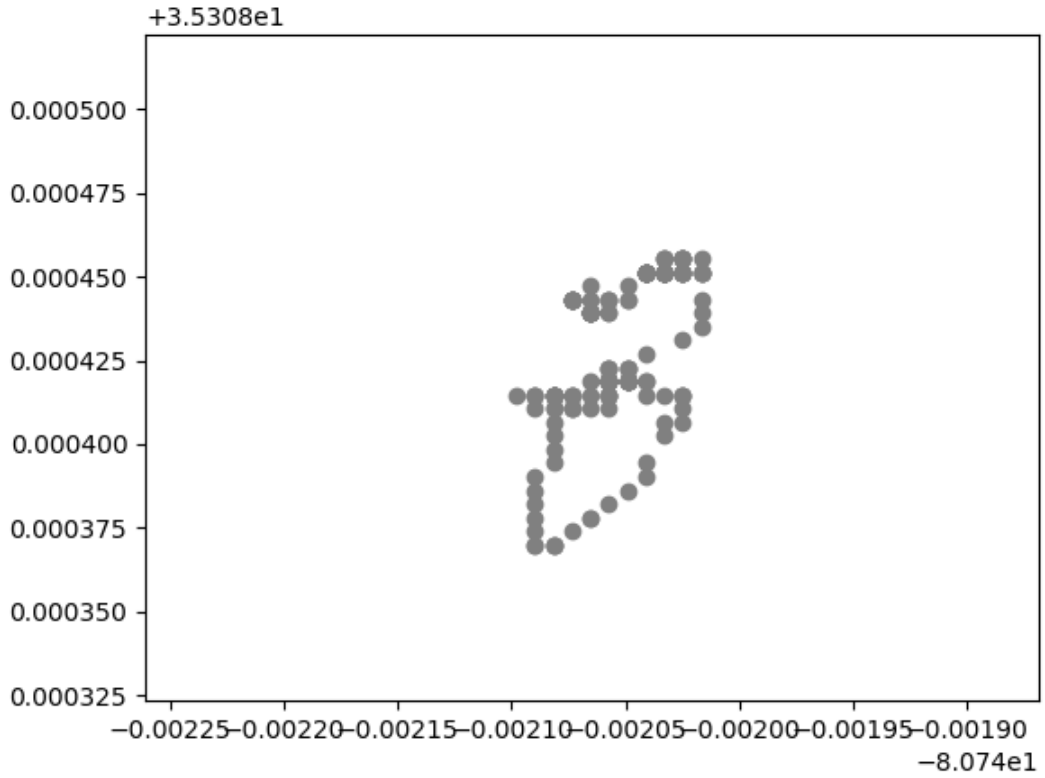


Figure 5.2: Latitude and Longitude with differential fix.

that a Differential GPS (DGPS) is being used to provide increased accuracy over a standard fix. It uses fixed known positions to give the difference between the position indicated by the satellite and the known position. The position is then adjusted and pseudo-range errors are eliminated. The difference in the readings for a standard fix and a differential fix can be seen in Figures 5.1 and 5.2. These points were then plotted on a map and it was clearly seen how error prone standard fix was and DGPS provided much better readings. This can be seen in the Figure 5.3. To test the distance and bearing formulas, two latitude, longitude points were taken along a track and the distance and bearing angle were measured. The start point was (35.308742, -80.744580) and the end point was (35.308603, -80.744221) as can be seen in the Figure 5.4. The distance received from Vincenty's formula was 39.487991 yards while the distance received from the Haversine formula was 39.4356955. This gives a difference

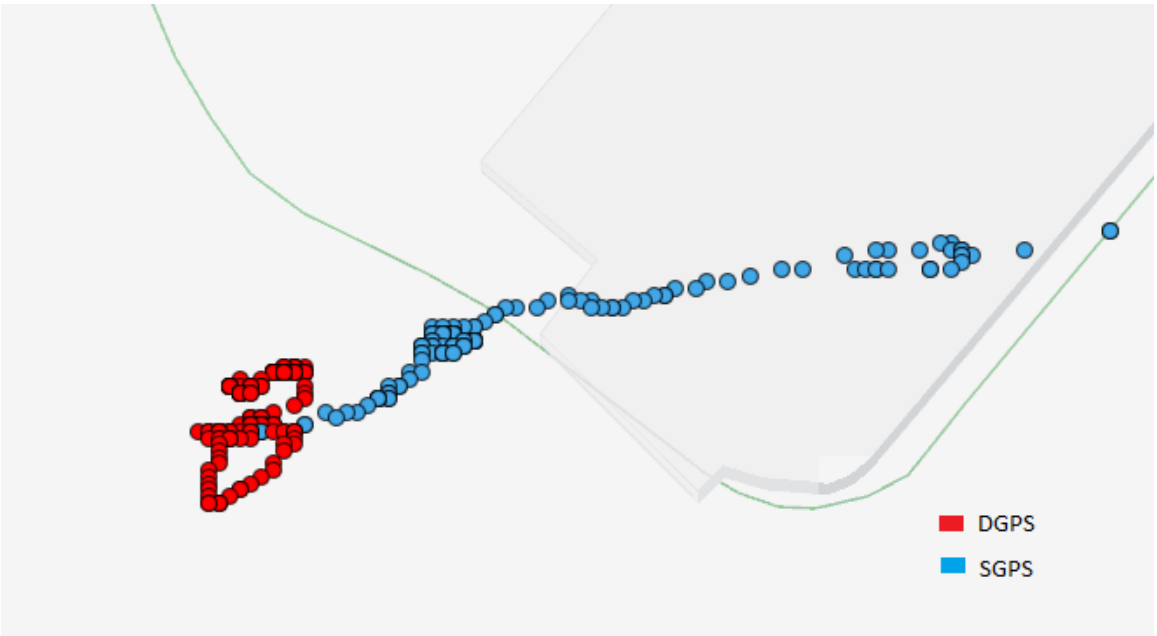


Figure 5.3: DGPS And Standard GPS Plotted On The Map.

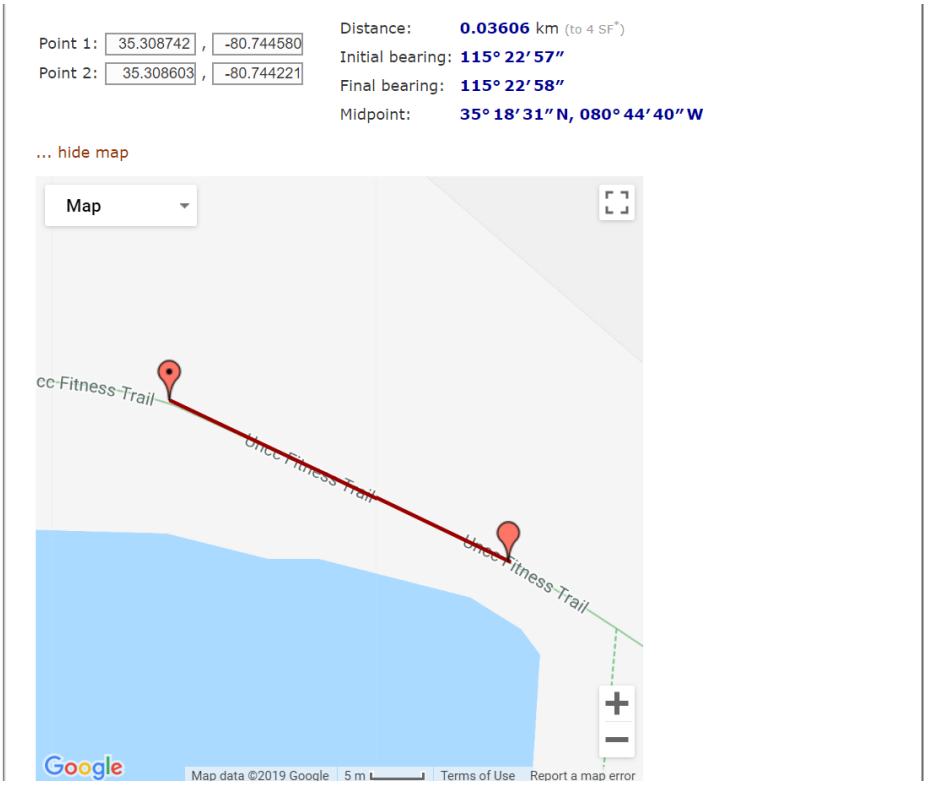


Figure 5.4: Map of two lat long points [44].

of 0.05229549 yards which is not significant. These distances were also compared to the distance Google map shows, i.e. 39.4 yards, which was close to the distances

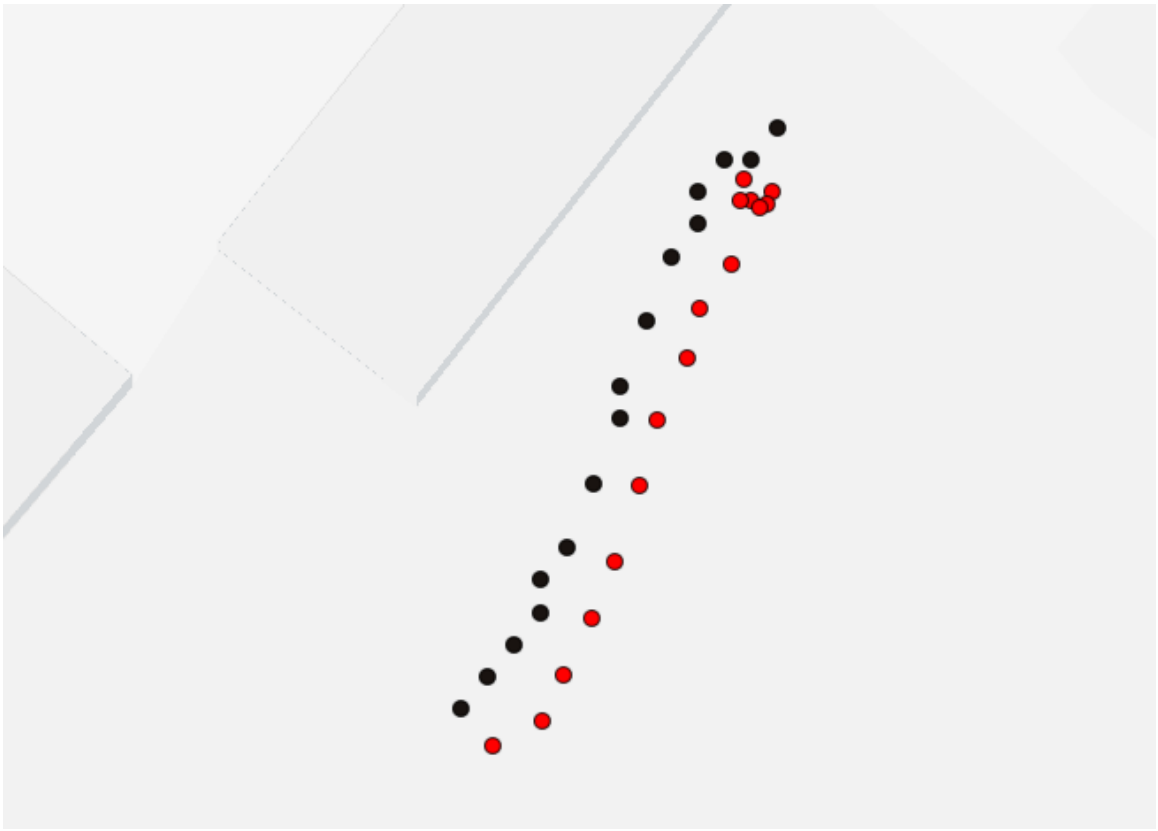

Figure 5.5: GPS path comparison

calculated from the formulas. The Figure 5.4 also shows the bearing angle between the two points. The next step was to test the GPS module with another reference. For this reason, GPS readings were taken from a mobile phones' GPS module using the MATLAB application. The data rate was set to 1Hz to match the data rate of the sensor in comparison. Along the same route, the GPS sensor readings were taken. The data from this was plotted and a difference of about 2 meters was observed. As seen in Figure 5.5, the red dots were Adafruit's Ultimate GPS readings and the black dots were GPS readings from a mobile phone. The comparison of the data received from two systems is shown in the Table 5.1 where 'P' is the distance between the two positions and 'O' is the orientation difference. The next step was to test the Kalman filtering process real-time. The sensor readings were taken outdoors. The data from the GPS and IMU was sent to the Raspberry Pi via serial communication.

Table 5.1: Comparison of the ATVs navigation and Mobile Phone data.

| Navigation System | Mobile Phone Sensors | P | O |
|---|---|---|---|
| (35.30951, -80.7422), 126.62 | (35.30952, -80.7422), 128.32 | 2.45 | 1.7 |
| (35.30951, -80.7423), 126.56 | (35.30952,-80.74222), 130.52 | 1.78 | 3.96 |
| (35.30951, -80.7422), 128.50 | (35.30952,-80.74223), 131.27 | 2.20 | 2.77 |
| (35.30951, -80.7422), 128.06 | (35.30951,-80.74224), 131.58 | 1.99 | 3.52 |
| (35.30951, -80.7422), 130.37 | (35.30950,-80.74224), 132. 14 | 2.377 | 1.77 |
| (35.30948, -80.7422), 130.59 | (35.30949,-80.74225), 128.62 | 3.32 | 1.97 |
| (35.30949, -80.7422), 129.94 | (35.30947,-80.74226), 129.56 | 3.88 | 0.38 |
| (35.30947, -80.7422), 133.31 | (35.30945,-80.74227), 135.34 | 4.22 | 2.03 |
| (35.30946, -80.7422), 132.17 | (35.30944,-80.74227), 136.35 | 3.41 | 4.18 |
| (35.30944, -80.7423), 131.29 | (35.30942,-80.74228), 130.48 | 3.40 | 0.81 |
| (35.30942, -80.7423), 130.08 | (35.30940,-80.74229), 127.71 | 3.62 | 2.37 |
| (35.3094, -80.7423), 128.38 | (35.30939,-80.74230), 124.12 | 2.89 | 4.26 |
| (35.30938, -80.7423), 125.13 | (35.30938,-80.74230), 126.75 | 1.96 | 1.62 |
| (35.30936, -80.7423), 127.56 | (35.30937,-80.74231), 131.96 | 2.16 | 4.4 |
| (35.30935, -80.7423), 130.44 | (35.30936,-80.74232), 130.48 | 2.65 | 0.04 |
| (35.30934, -80.7423), 130.06 | (35.30935,-80.74233), 132.48 | 1.83 | 2.42 |

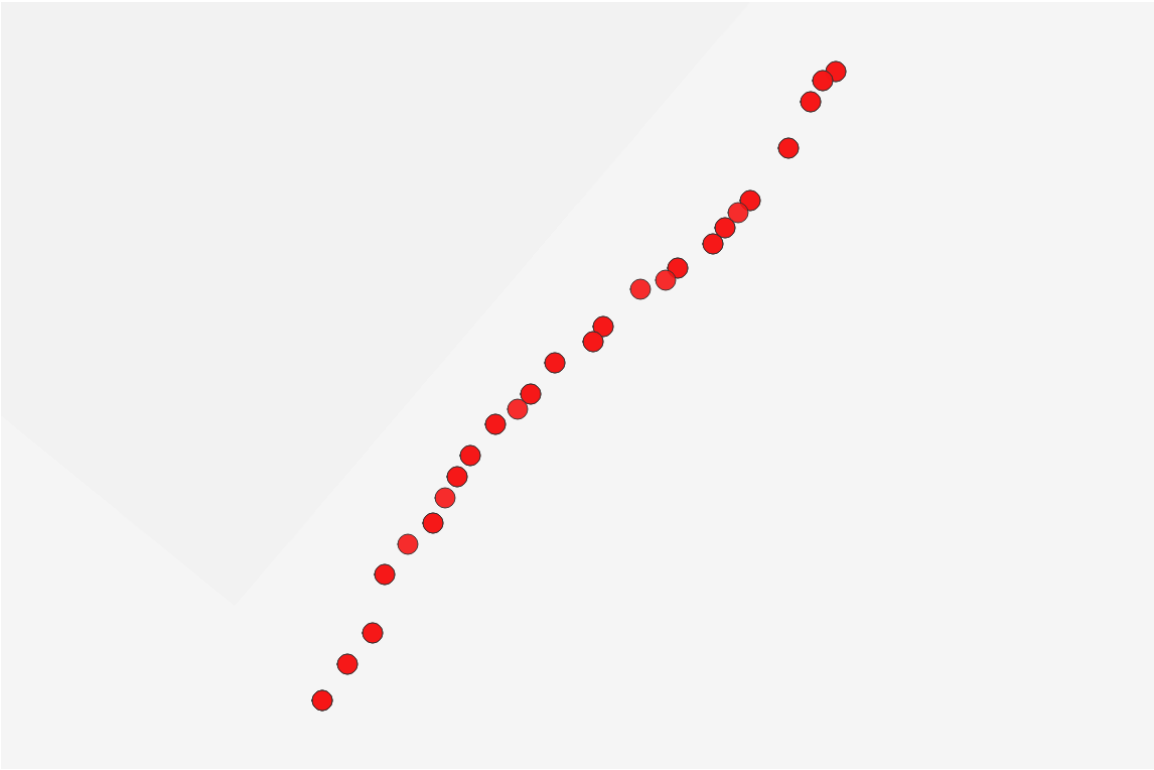

Figure 5.6: Original GPS readings.

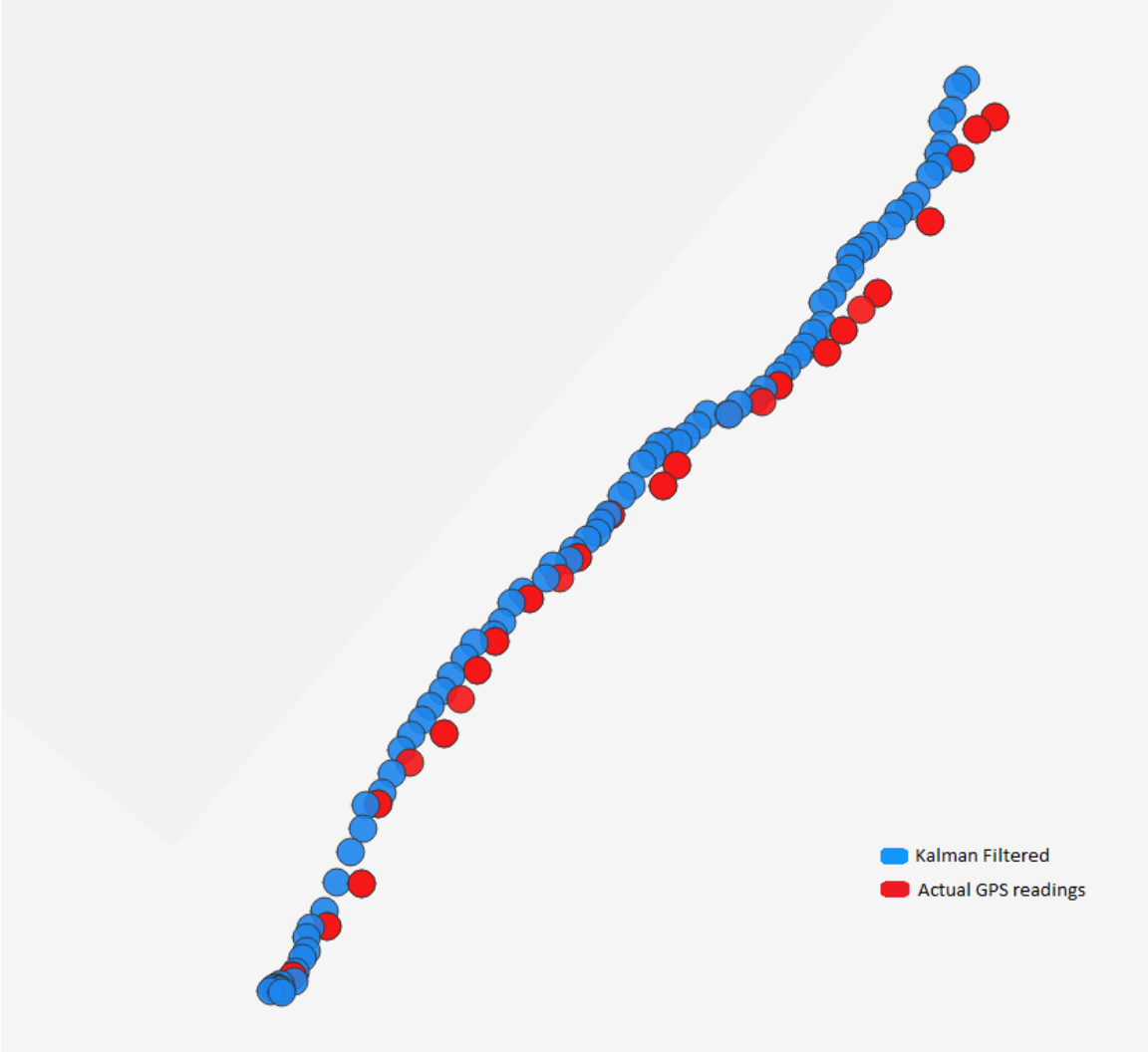


Figure 5.7: Kalman filtered every 500ms.

Based on these readings the Kalman estimates and the original GPS readings were stored in a file to verify the correctness. The original GPS readings can be seen in the Figure 5.6. The accelerometer data was sent once every 50milliseconds and the GPS data once every second. The Kalman filtered readings based on both these sensors were plotted once every 500milliseconds. The Kalman readings were plotted in blue and the GPS readings in red. The Kalman was able to predict points in between GPS readings as seen in Figure 5.7. After testing the system in a straight line, the next step was to get results on a path involving some turns. For this purpose, the navigation system, and mobile phone were steered across a trail as seen in Figure

Figure 5.8: Testing environment for the straight path.

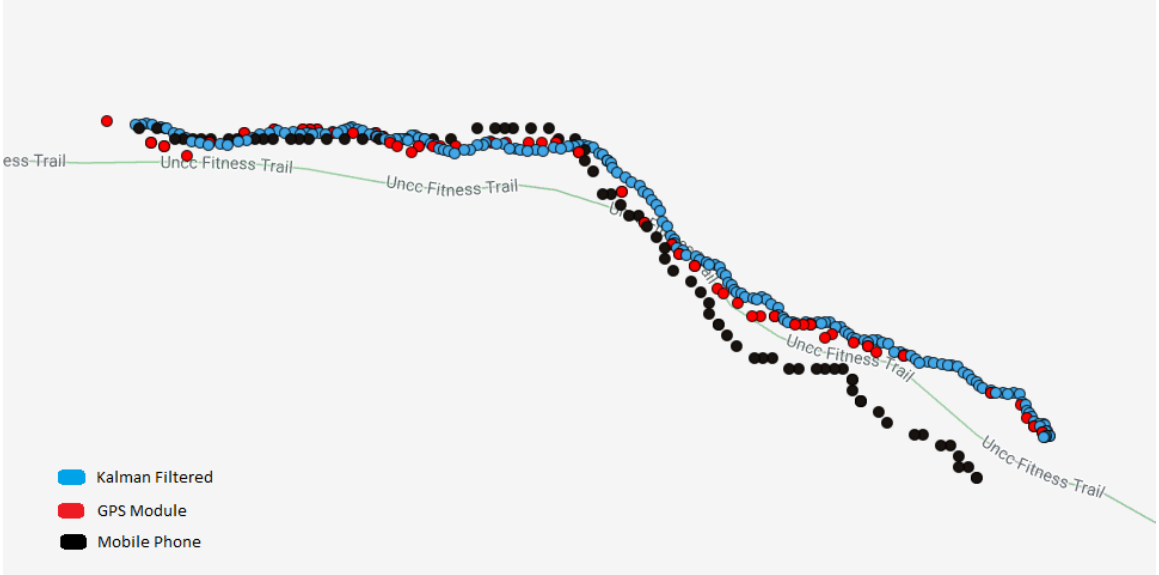


Figure 5.9: Navigation System vs Mobile phone readings on a trail.

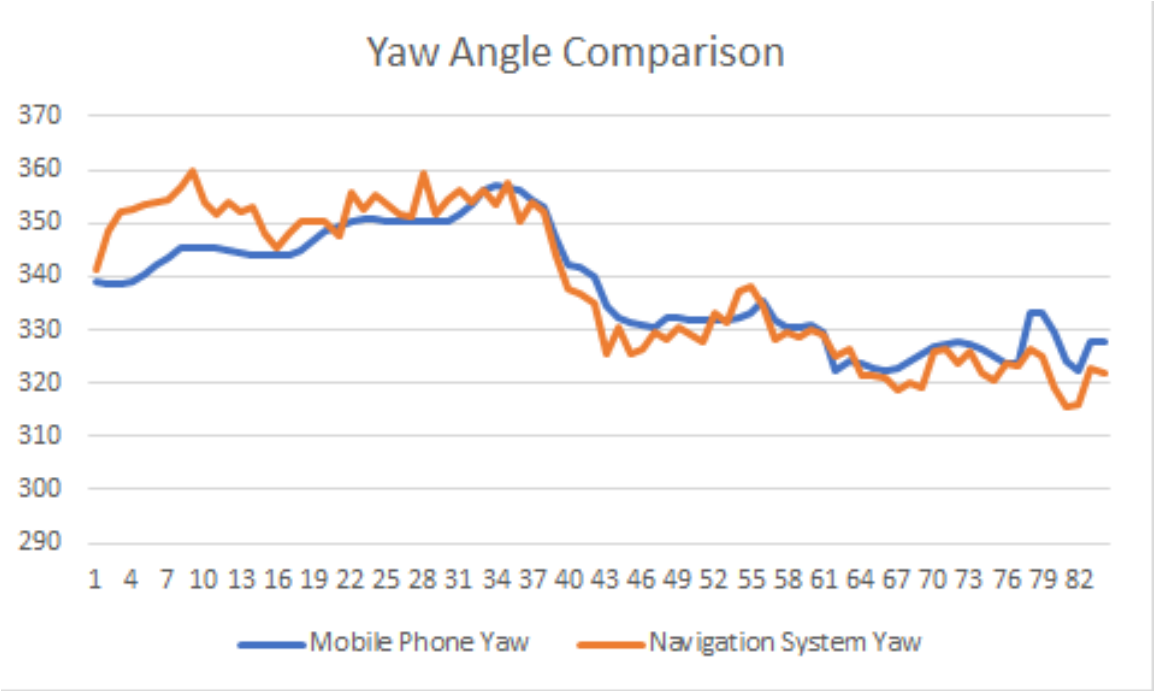


Figure 5.10: Heading angle comparison.

5.9. The red dots indicate the mobile phone positions, the red dots indicate the navigation system positions, and the blue dots indicate the Kalman filtered readings which were outputted every 500 milliseconds. Distance between these points was calculated using the Haversine formula, and then plotted on a map. The Kalman

Figure 5.11: The trail where the Kalman filtering was tested.

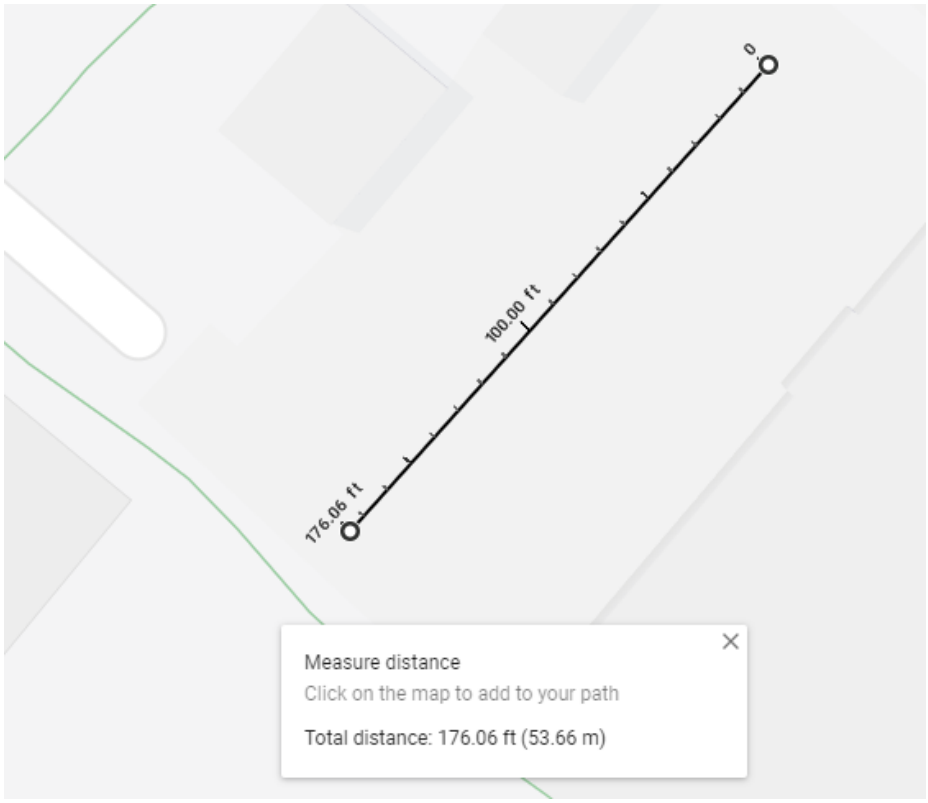


Figure 5.12: Distance calculation on Google Map.

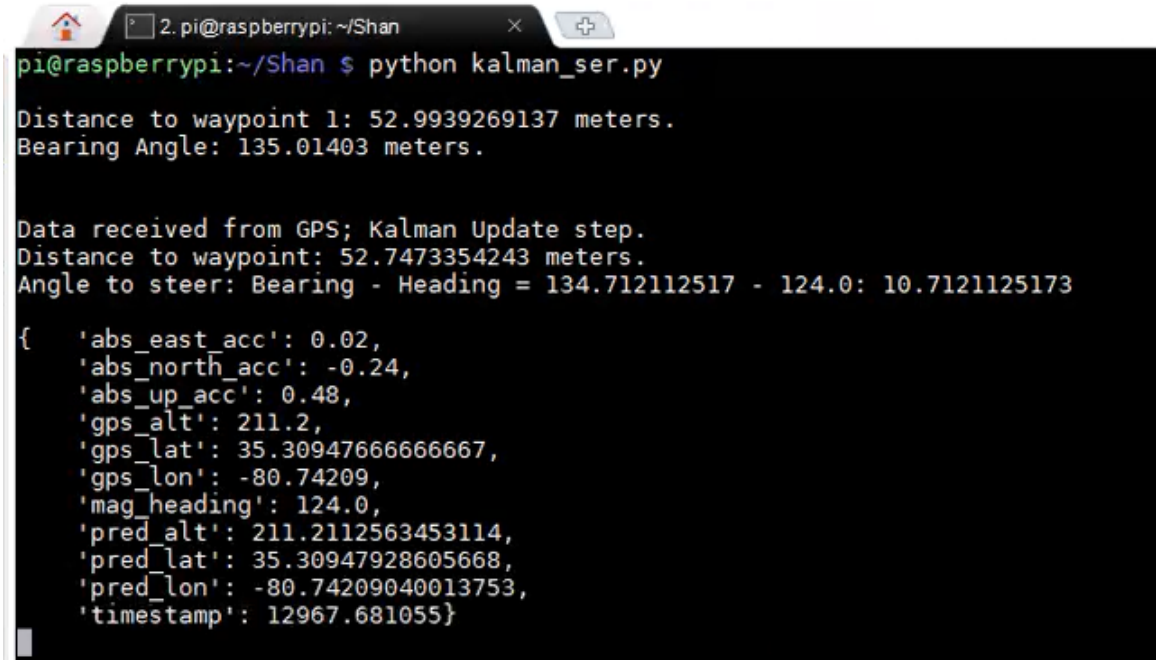


Figure 5.13: Start of the code.

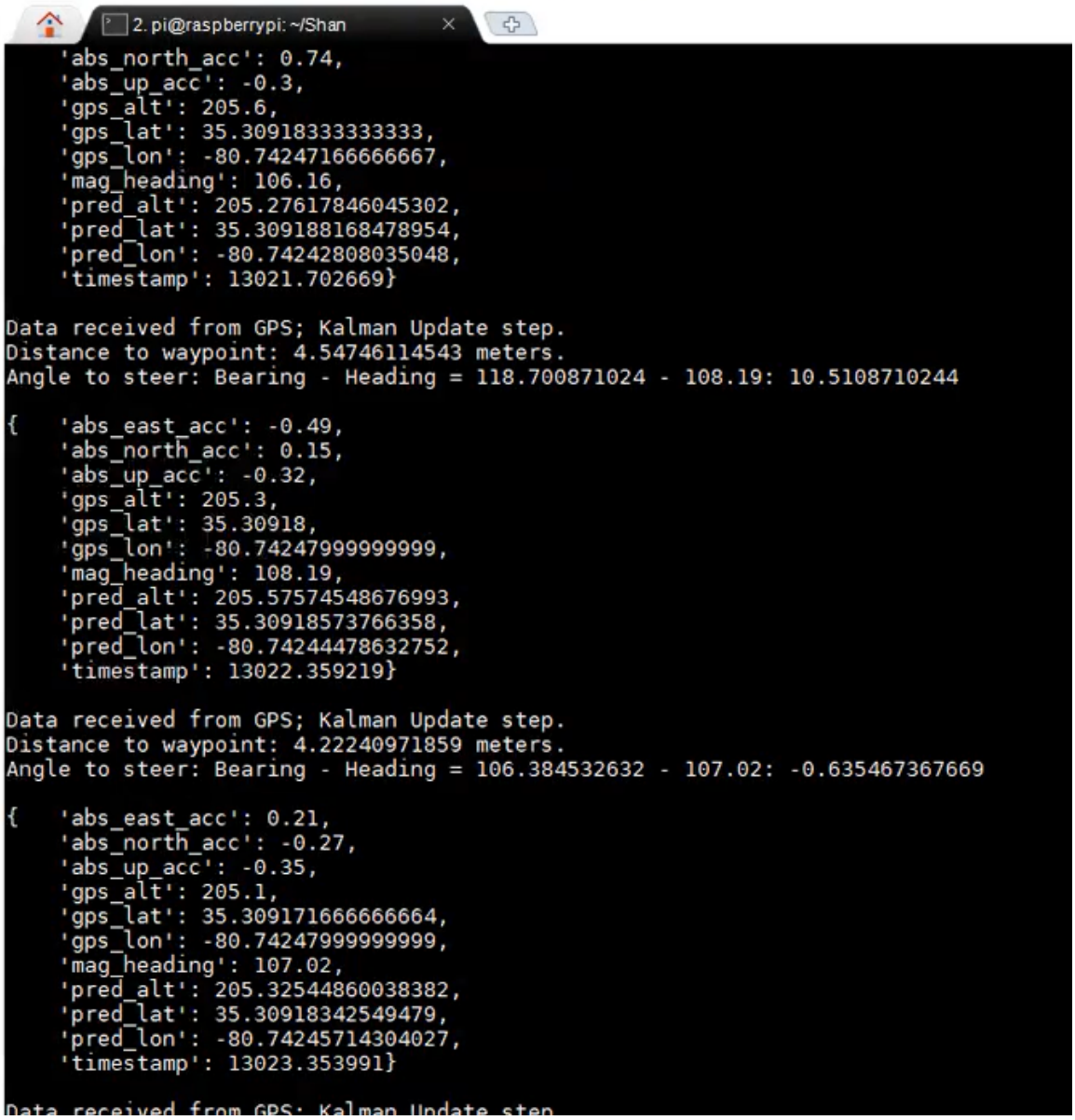


Figure 5.14: Reading when near the waypoint.

filtered readings can be seen plotted between the GPS module readings as expected, and smoothing the GPS module data jumps. The heading angle difference between the Navigation System, and Mobile Phone is shown in 5.10, and were found to be very similar with a deviation of about 15 degrees max. Once the system was tested with a straight path, and a turning path, the distance calculation, and steering angle determination was integrated into the code. The path was first put on Google maps to

Figure 5.15: The location where the system was tested for GPS outages.


Figure 5.16: Results obtained from the system when GPS was manually switched off.

get the distance as seen in Figure 5.12. The code was then run by moving along that path. The beginning of the code can be seen in Figure 5.13. The distance displayed by the Raspberry Pi seemed to match the distance showed on Google maps. The code only displays the data during Kalman update stage as printing every 50ms is not ideal. A screenshot of the system display when it reaches near the waypoint can be seen in Figure 5.14. The next test was to check how the system works when the GPS module doesn't output data. For this, the GPS was disconnected manually for roughly 4 to 5 seconds and the Kalman filter was still able to predict the position using accelerometer. This can be seen in the Figure 5.16. This testing was done in

an environment seen in Figure 5.15.

# CHAPTER 6: CONCLUSION

The aim of this Thesis was to build a navigation system and a standardized API library that can be used to make an ATV autonomous. The initial phase of the work was to understand the previous work, and control the actuators with embedded boards. The BSL built using just API calls proved very useful for debugging during this initial phase. The modularity added by the BSL was demonstrated when a switch in the controller board was made, and the only change needed to be made was the way the registers were accessed. The API library was built in such way that only the base register access layer and the communication protocols was dependent on the embedded board used. Higher level libraries like those used for CAN bus, brake, throttle and the navigation system were independent of the embedded platform used. The next step was to build a navigation system using this BSL. A GPS sensor, and INS system was used for this purpose. The NMEA frames received from the GPS were studied, and the ones not useful for this application were disabled. A parsing mechanism was implemented that would extract the desired data from the NMEA frames enabled. Using this, the position, and speed were obtained from the GPS module. The methods for distance calculation were studied, and tested for their accuracy, and memory usage on the MSP430. The Haversine formula, and the Vincenty formula were used to obtain distance between two GPS points and their outputs were compared to Google maps. The Haversine formula proved to give accurate readings for the distance the waypoints were assumed to be under. The Haversine formula also wasn't computation heavy like the Vincenty formula. Bearing angle from the ATV to the waypoint was also determined and tested.

For the IMU, the BNO055 was chosen to avoid dealing with a lot of floating point

math in the microprocessor. While the BNO055 performs its own sensor fusion on a ARM Cortex M0 processor, there were still errors associated with their Euler angles. To avoid this, the Quaternion output was used and converted to rotational matrix in the MSP430. While the GPS sensor and the INS in general were a popular choice for navigation, the results were very noisy.

To make the system efficient, a Kalman filter was implemented by using accelerometer values to estimate positions. For this purpose, the accelerometer values were converted to an absolute Earth-referenced form such that they provide the same values in any orientation. Making the accelerometer orientation independent was to make sure the modules remained interchangeable. Once the Kalman filter was studied, the next step was to determine error of the sensors to input into the filter. The accelerometer was kept stationary and the Earth-referenced readings on each axis were taken. While the expected output was 0 on each axis, the actual output values ranged from -0.09 to +0.09. For the GPS, while the GPS errors are broadcast in the GPGST field for costly receivers, this field was found to be absent from the Adafruit's GPS module. However, the datasheet mentioned the GPS readings to be accurate up to 3 meters 50 % CEP. This was used in error matrix for the GPS. The accelerometer data was received at 20Hz and the GPS data was received every 1Hz. Prediction was made every 50ms, i.e., every time the acceleration data was received. However, the Kalman output was set to once every 500s. The accelerometer data suffers from drift overtime. This is why only the accelerometer couldn't be used to predict position. The GPS data outputted every 1 second is used to update the Kalman filter. Although the GPS data is noisy, the errors do not compound overtime. The whole Kalman filtered system was tested outdoors with real-time data and smoother positions were observed when compared to the GPS readings. The Kalman filter was also able to predict positions in between GPS readings fairly accurately.

# REFERENCES


[1] K. Cheung, "World's Top 33 Companies Working on Self Driving Cars." https://algorithmxlab.com/blog/2018/12/14/worlds-top-33-companies-working-on-self-driving-cars/, Dec. 2018.

[2] A. Hawkins, "Toyota's e-Palette is a weird, self-driving modular store on wheels." https://www.theverge.com/2018/1/8/16863092/toyota-e-palette-self-driving-car-ev-ces-2018, Jan. 2018.

[3] B. Merrill, "Disabled Americans deserve the benefits of self-driving cars." https://thehill.com/blogs/congress-blog/technology/407362-disabled-americans-deserve-the-benefits-of-self-driving-cars, Sept. 2018.

[4] J. Levinson, J. Askeland, J. Becker, J. Dolson, D. Held, S. Kammel, J. Z. Kolter, D. Langer, O. Pink, V. Pratt, *et al.*, "Towards fully autonomous driving: Systems and algorithms," in *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pp. 163–168, IEEE, 2011.

[5] J. Zhao, B. Liang, and Q. Chen, "The key technology toward the self-driving car," *International Journal of Intelligent Unmanned Systems*, vol. 6, no. 1, pp. 2–20, 2018.

[6] K. Angelova, "US-Mexico Border Patrol Agents Have Sweet Rides." https://www.businessinsider.com/us-mexico-border-patrol-uses-cool-atvs-2013-4, Apr. 2013.

[7] A. Speier, "Patient Access and ATV." https://www.firerescuemagazine.com/articles/print/volume-8/issue-3/vehicle-operation-and-apparatus/patient-access-atvs.html.

[8] D. Murphy, "The Safe Use of ATVs in Agriculture." https://extension.psu.edu/the-safe-use-of-atvs-in-agriculture, June 2014.

[9] J. Darukhanawala, "Honda Showcases Self-driving ATV Concept." https://www.zigwheels.com/news-features/news/hondas-new-autonomous-atv-concept-to-aid-largescale-operations-unveiled-at-2019-ces/32717/, Jan. 2019.

[10] T. Ort, L. Paull, and D. Rus, "Autonomous Vehicle Navigation in Rural Environments without Detailed Prior Maps," in *International Conference on Robotics and Automation*, 2018.

[11] D. Silver, "Self-Driving Cars; CAN Bus." https://medium.com/self-driving-cars/can-bus-22024d35ce63, July 2016.

[12] W. Rahiman and Z. Zainal, "An overview of development GPS navigation for autonomous car," in *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, pp. 1112–1118, IEEE, 2013.

[13] F. Beainy and S. Commuri, "Development of an autonomous ATV for real-life surveillance operations," in *2009 17th Mediterranean Conference on Control and Automation*, pp. 904–909, IEEE, 2009.

[14] D. Li, "Research on applications of LBS based on electronic compass assisted GPS," in *2009 International Symposium on Information Engineering and Electronic Commerce*, pp. 599–602, IEEE, 2009.

[15] R. A. McKinney, M. J. Zapata, J. M. Conrad, T. W. Meiswinkel, and S. Ahuja, "Components of an autonomous all-terrain vehicle," in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pp. 416–419, IEEE, 2010.

[16] A. Cortner, J. M. Conrad, and N. A. BouSaba, "Autonomous all-terrain vehicle steering," in *2012 Proceedings of IEEE Southeastcon*, pp. 1–5, IEEE, 2012.

[17] J. R. Henderson, J. M. Conrad, and C. Pavlich, "Using a CAN bus for control of an All-terrain Vehicle," in *IEEE SOUTHEASTCON 2014*, pp. 1–5, IEEE, 2014.

[18] K. H. Erian, S. Mhapankar, S. Gambill, and J. M. Conrad, "System Integration over a CAN Bus for a Self-Controlled, Low-Cost Autonomous All-terrain Vehicle," in *IEEE SoutheastCon 2019*, pp. 1–8, IEEE, 2019.

[19] "Honda, Technology." `https://www.honda.co.nz/about-honda/technology/`.

[20] "MSP430." `https://www.ti.com/store/ti/en/p/product/?p=MSP-EXP430F5529LP`.

[21] "Model IMU, GPS, and INS/GPS." `https://www.mathworks.com/help/fusion/gs/model-imu-gps-and-insgps.html`.

[22] "What is GPS?." `https://www8.garmin.com/aboutGPS/`.

[23] "Adafruit Ultimate GPS - One GPS to rule." `https://learn.adafruit.com/adafruit-ultimate-gps/overview`.

[24] "PA6H GPS Standalone Module Data Sheet." `https://cdn-shop.adafruit.com/datasheets/GlobalTop-FGPMMOPA6H-Datasheet-V0A.pdf`.

[25] M. J. Caruso, "Applications of magnetoresistive sensors in navigation systems," tech. rep., SAE Technical Paper, 1997.

[26] D. DePriest, "GPS Data." `https://www.gpsinformation.org/dale/nmea.htm`.

[27] R. W. Sinnott, "Virtues of the Haversine," *Sky Telesc.*, vol. 68, p. 159, 1984.

[28] T. Vincenty, "Direct and inverse solutions of geodesics on the ellipsoid with application of nested equations," *Survey review*, vol. 23, no. 176, pp. 88–93, 1975.

[29] C. Veness, "Vincenty solutions of geodesics on the ellipsoid." `https://www.movable-type.co.uk/scripts/latlong-vincenty.html`.

[30] H. Mahmoud and N. Akkari, "Shortest path calculation: a comparative study for location-based recommender system," in *2016 World Symposium on Computer Applications & Research (WSCAR)*, pp. 1–5, IEEE, 2016.

[31] "Wikipedia - atan2." `https://en.wikipedia.org/wiki/Atan2`.

[32] T. Zaman, "Calculating heading with tilted compass." `http://www.timzaman.com/2011/04/21/heading-calculating-heading-with-tilted-compass/`.

[33] W. Holder, "Calibrating the Compass." `https://sites.google.com/site/wayneholder/self-driving-rc-car/calibrating-the-compass`.

[34] Bosch Sensortech, *BNO055 data sheet*, 6 2016. Rev. 1.4.

[35] "Understanding Euler Angles." `http://www.chrobotics.com/library/understanding-euler-angles`.

[36] R. Curtis, "OA Guide to Map and Compass - Part 2." `http://www.princeton.edu/~oa/manual/mapcompass2.shtml`.

[37] "Magnetic Declination Estimated Value." `https://www.ngdc.noaa.gov/geomag/calculators/magcalc.shtml`.

[38] K. Shoemake, "Animating rotation with Quaternion curves," in *ACM SIGGRAPH computer graphics*, vol. 19, pp. 245–254, ACM, 1985.

[39] M. Hughes, "Capturing IMU Data with a BNO055 Absolute Orientation Sensor." `https://www.allaboutcircuits.com/projects/bosch-absolute-orientation-sensor-bno055/`.

[40] M. Amin, M. B. I. Reaz, M. Bhuiyan, and S. Sheikh Nasir, "Kalman filtered GPS accelerometer based accident detection and location system: A low-cost approach," *Current science*, vol. 106, pp. 1548–1554, 06 2014.

[41] F. Caron, E. Duflos, D. Pomorski, and P. Vanheeghe, "GPS/IMU data fusion using multisensor Kalman filtering: introduction of contextual aspects," *Information fusion*, vol. 7, no. 2, pp. 221–230, 2006.

[42] H. Hermsen, "Using gps and accelerometer data for rowing race tracking," *Dept of Artificial Intelligence*, 2013.

[43] G. Czerniak, "Kalman Filter Guide." `http://greg.czerniak.info/guides/kalman1/`.

[44] C. Veness, "Calculate distance, bearing and more between Latitude/Longitude points." https://www.movable-type.co.uk/scripts/latlong.html.

# APPENDIX A: APIs

Input/Output Code APIs are as follows:

```
1  void pinSetIP(unsigned int port, int pin); // Sets a pin in
       input mode.
2  void gpioSetIP(unsigned int port, int pin); // Sets a pin in
       gpio input mode.
3  int gpioGetVal(unsigned int port, int pin); // Gets the value
        of a pin.
4  void pinSetOP(unsigned int port, int pin); // Sets a pin in
       output mode.
5  void gpioSetOP(unsigned int port, int pin); // Sets a pin in
       output GPIO mode.
6  void gpioSetVal(unsigned int port, int pin, int val); // Sets
        an output pin as high or low depending on the value.
7  void pinSetOffsetSet(unsigned int port, int pin, int offset);
        // Sets the register offset for a particular port's pin.
8  void pinSetOffsetClear(unsigned int port, int pin, int offset
       ); // Clears the register offset for a particular port's
       pin.
9  unsigned int pinSetOffsetGet(unsigned int port, int pin, int
       offset); // Gets the register offset value for a
       particular port's pin.
10 void pinClearSEL(unsigned int port, int pin); // Clears the
       pin's SEL register.
11 void pinSetSEL(unsigned int port, int pin); // Sets the pin's
        SEL register.
12 void pinSetREN(unsigned int port, int pin); // Sets the pin's
```

```
    REN register.
13 void pinClearREN(unsigned int port, int pin); // Clears the
    pin's REN register.
14 void pinSetPull(unsigned int port, int pin, int mode); //
    Sets the pin as a pullup or pulldown.
15 void pinSetPWM(unsigned int port, int pin); // Sets a pin in
    pwm mode by setting the SEL register, period and clock.
16 void pwmSetDuty(unsigned int dutycycle); // Changes dutycycle
     by setting CCR1 register.
```

Watchdog Timer Code APIs are as follows:

```
1 void wdtHold(void); // Holds the watchdog timer.
2 void wdtStart(void); // Starts the watchdog timer by clearing
     the hold.
3 void wdtInit(int interval); // Initializes the watchdog timer
     but doesn't start it.
4 void wdtResetTimer(void); // Resets the watchdog counter to
    prevent reset.
```

SPI Code APIs are as follows:

```
1 void spiMasterInit(void); // Initializes the SPI as master.
2 void spiSlaveInit(void); // Initializes the SPI as slave.
3 void spiMasterPinInit(void); // Sets SEL & SEL2 to USCI mode
    for MISO, MOSI & CLK. Sets SS pin to low.
4 void spiSlavePinInit(void); // Sets SEL & SEL2 to USCI mode
    for MISO, MOSI & CLK.
5 void spiMasterConfigureCR(void); // Configures the SPI
    control registers to 3-pin, 8-bit SPI master, sync by
```

```
    clock; enable interrupts.
6 void spiSlaveConfigureCR(void); // Configures the SPI control
     registers to 3-pin, 8-bit SPI slave, sync by clock;
    enable interrupts.
7 void spiSelectSlave(void); // Sets the SS pin to low to
    select the slave.
8 void spiDeselectSlave(void); // Sets the SS pin to high to
    deselect the slave.
9 uint8_t spiTransmitData(char data); // Checks if TX buffer is
     ready and then sets the UCA0TXBUF register with the data
    to send. Reads the RX register to avoid overrun error.
10 void spiCheckTxReady(void); // Checks the UCTXIFG flag for
    interrupt.
11 uint8_t spiReceiveData(void); // Gets the received character
    from UCRXBUF.
12 void spiCheckUSCI(void); // Checks the UCSTAT status register
     to make sure it isn't busy.
```

---

I2C Code APIs are as follows:

---

```
1 void i2cConfigureCR(uint8_t add); // Configure the control
    registers to function as I2C as master and set the clock.
2 void i2cInit(uint8_t add); // Initialize the pins and control
     registers and set's the slave address.
3 uint8_t i2cRead(void); // Reads the RXBUF to get data sent
    from slave.
4 void i2cWrite(uint8_t data); // Write to the TXBUF
5 void i2cSendStartTx(void); // Put the master in Tx mode and
```

```
    send start condition.
6 void i2cSendStartRx(void); // Put the master in Rx mode and
    send start condition.
7 void i2cSendStop(void); // Sends stop condition to stop I2C
    communication
```

UART Code APIs are as follows:

```
1 void uartInit(void); // Initializes the UART at with a baud
    rate of 9600 with pins specified in the RX & TX
    pindescriptor.
2 void uartPinInit(void); // Sets the pins RX and TX SEL
    register.
3 void uartConfigureCR(void); // Configures the USCI register
    for clock and baud rate.
4 void uartCheckTxReady(void); // Returns when TX buffer is
    ready.
5 void uartCheckRxReady(void); // Returns when RX buffer is
    ready.
6 uint8_t uartReceiveChar(void); //  If uartCheckRxReady
    returns, return the data in the RX buffer.
7 void uartTransmitChar(uint8_t data); // If uartCheckTxReady
    returns, sends a character via UART.
```

GPS Code APIs are as follows:

```
1 void gpsInit(void); // Initializes the structures to zero and
     initializes UART.
2 void initStruct(struct gps_struct *gp); // Zeros out all the
    elements of the gps structure.
```

```
3  void initGGAStruct(struct gpgga_struct* gp); // Zeros out all
       the elements of the gpgga structure.
4  void initRMCStruct(struct gprmc_struct* gp); // Zeros out all
       the elements of the gprmc structure.
5  uint8_t gpsParseGGA(void); // Parses GPGGA UART message and
       updates the gga structure with time, lat, lat direction,
       long, long direction and fix quality.
6  uint8_t gpsParseRMC(void); // Parses GPRMC UART message and
       updates the rmc structure with lat, lat direction, long,
       long direction, speed, course and status.
7  struct gps_struct gpsReadNMEA(void); // Reads GPGGA & GPRMC
       messages. Returns gps structure of both data merged.
8  struct gps_struct gpsReadNMEAGGA(void); // Reads only the
       GPGGA message, parses it and updates the gps_struct with
       speed = 0.0. Takes 1 second.
9  void gpsConvertData(double *latitude, char ns,  double *
       longitude, char we); // Convert lat, long from degrees to
       decimal.
10 double gpsCovertToDec(double deg_point); // Converts degrees
       passed to decimals.
11 double gpsGetDistance(struct gps_struct* a, struct gps_struct
       * b); // Gets the distance between two lat long points
       using haversine formula.
12 double gpsGetBearing(struct gps_struct* a, struct gps_struct*
        b); // Calculates the bearing angle from point a to b.
13 uint8_t gpsGetSeconds(struct gps_struct* prev, struct
       gps_struct* curr); // Gets time difference between two
```

```
different readings.
```

IMU Code APIs are as follows:

```
1  void bno055Init(void); //Initializes the I2C with the address
       of BNO055, sets to normal power mode and then NDOF fusion
       mode.
2  void bno055WriteReg(uint8_t reg, uint8_t data); // Writes a
     character to a register via I2C.
3  uint8_t bno055ReadReg(uint8_t reg); // Read a character from
     a register via I2C.
4  void bno055SetMode(uint8_t mode); // Sets the mode of BNO055.
5  void bno055SetExtCrystalUse(uint8_t usextal); // Sets the
     external crystal use to either 1 or 0.
6  uint8_t bno055GetCalStat(void); // Gets the calibration
     status of the all the sensors and system.
7  void bno055GetCalData(offset_data* data); // Get offset
     register data. Used to read calibration profile.
8  void bno055SetCalData(offset_data* data); // Sets offset
     register data. Used to store calibration profile.
9  void bno055GetEuler(euler_data *data); // Reads the euler
     register data. Divides by 16.0 to convert as 1 degree = 16
      LSB.
10 void bno055GetQuat(quat_data *data); // Reads the quaternion
     register data. Divides by 2^14 to convert to quaternion
     units. Also converts it to yaw, pitch and roll.
11 void bno055GetLinearAcc(acc_data *data); //  Reads the linear
      acceleration register data. Divides by 100.0 as 1 m/s^2 =
```

```
    100 LSB.
12  void bno055GetRefAcc(quat_data data, acc_data linearaccel,
    acc_data *ref_data); // Converts acceleration to earth-
    referenced acceleration.
13  void bno055GetRawAcc(acc_data *data); // Reads the raw
    acceleration register data. Divides by 100.0 as 1 m/s^2 =
    100 LSB.
14  void bno055SetPower(uint8_t mode); // Sets the power mode of
    BNO055.
```