COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

DATA ACQUISITION FOR GO KARTS THROUGH SENSOR FUSION

Autor: Enrique Alejo Álvarez

Director: Alanson Sample

Madrid

Julio de 2019

AUTHORIZATION FOR DIGITALIZATION, STORAGE AND DISSEMINATION IN THE NETWORK OF END-OF-DEGREE PROJECTS, MASTER PROJECTS, DISSERTATIONS OR BACHILLERATO REPORTS

1. Declaration of authorship and accreditation thereof.

The author Mr./Ms. Enrique Alejo Álvarez

HEREBY DECLARES that he/she owns the intellectual property rights regarding the piece of work: Data Acquisition for Gekets through Sensor Fusion

that this is an original piece of work, and that he/she holds the status of author, in the sense granted by the Intellectual Property Law.

2. Subject matter and purpose of this assignment.

With the aim of disseminating the aforementioned piece of work as widely as possible using the University's Institutional Repository the author hereby **GRANTS** Comillas Pontifical University, on a royalty-free and non-exclusive basis, for the maximum legal term and with universal scope, the digitization, archiving, reproduction, distribution and public communication rights, including the right to make it electronically available, as described in the Intellectual Property Law. Transformation rights are assigned solely for the purposes described in a) of the following section.

3. Transfer and access terms

Without prejudice to the ownership of the work, which remains with its author, the transfer of rights covered by this license enables:

a) Transform it in order to adapt it to any technology suitable for sharing it online, as well as including metadata to register the piece of work and include "watermarks" or any other security or protection system.

b) Reproduce it in any digital medium in order to be included on an electronic database, including the right to reproduce and store the work on servers for the purposes of guaranteeing its security, maintaining it and preserving its format.

c) Communicate it, by default, by means of an institutional open archive, which has open and cost-free online access.

d) Any other way of access (restricted, embargoed, closed) shall be explicitly requested and requires that good cause be demonstrated.

e) Assign these pieces of work a Creative Commons license by default.

f) Assign these pieces of work a HANDLE (*persistent* URL). by default.

4. Copyright.

The author, as the owner of a piece of work, has the right to:

a) Have his/her name clearly identified by the University as the author

b) Communicate and publish the work in the version assigned and in other subsequent versions using any medium.

c) Request that the work be withdrawn from the repository for just cause.

d) Receive reliable communication of any claims third parties may make in relation to the work and, in particular, any claims relating to its intellectual property rights.

5. Duties of the author.

The author agrees to:

a) Guarantee that the commitment undertaken by means of this official document does not infringe any third party rights, regardless of whether they relate to industrial or intellectual property or any other type.

b) Guarantee that the content of the work does not infringe any third party honor, privacy or image rights.
c) Take responsibility for all claims and liability, including compensation for any damages, which may be brought against the University by third parties who believe that their rights and interests have been infringed by the assignment.
d) Take responsibility in the event that the institutions are found guilty of a rights infringement regarding the work subject to assignment.

6. Institutional Repository purposes and functioning.

The work shall be made available to the users so that they may use it in a fair and respectful way with regards to the copyright, according to the allowances given in the relevant legislation, and for study or research purposes, or any other legal use. With this aim in mind, the University undertakes the following duties and reserves the following powers:

a) The University shall inform the archive users of the permitted uses; however, it shall not guarantee or take any responsibility for any other subsequent ways the work may be used by users, which are non-compliant with the legislation in force. Any subsequent use, beyond private copying, shall require the source to be cited and authorship to be recognized, as well as the guarantee not to use it to gain commercial profit or carry out any derivative works.
b) The University shall not review the content of the works, which shall at all times fall under the exclusive responsibility of the author and it shall not be obligated to take part in lawsuits on behalf of the author in the event of any infringement of intellectual property rights deriving from storing and archiving the works. The author hereby waives any claim against the University due to any way the users may use the works that is not in keeping with the legislation in force.
c) The University shall adopt the necessary measures to safeguard the work in the future.
d) The University reserves the right to withdraw the work, after notifying the author, in sufficiently justified cases, or in the event of third party claims.

Madrid, on 20 of july, 2019

HEREBY ACCEPTS

Signed... 

Reasons for requesting the restricted, closed or embargoed access to the work in the Institution's Repository

Declaro, bajo mi responsabilidad, que el Proyecto presentado con el título

.....Data Acquisition for GoKarts through.....
.....Sensor Fusion.....

en la ETS de Ingeniería - ICAI de la Universidad Pontificia Comillas en el curso académico 2018-2019 es de mi autoría, original e inédito y no ha sido presentado con anterioridad a otros efectos. El Proyecto no es plagio de otro, ni total ni parcialmente y la información que ha sido tomada de otros documentos está debidamente referenciada.

Fdo.: Enrique Alejo Álvarez Fecha: 20 / 7 / 19

Autorizada la entrega del proyecto

EL DIRECTOR DEL PROYECTO

Fdo.: Alanson Sample Fecha: 1 / 5 / 2019

COMILLAS
UNIVERSIDAD PONTIFICIA
ICAI

GRADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES

TRABAJO FIN DE GRADO

DATA ACQUISITION FOR GO KARTS THROUGH SENSOR FUSION

Autor: Enrique Alejo Álvarez

Director: Alanson Sample

Madrid

Julio de 2019

ADQUISICIÓN DE DATOS PARA KARTS MEDIANTE FUSIÓN DE SENSORES

Autor: Alejo Álvarez, Enrique.

Director: Sample, Allanson.

Entidad Colaboradora: ICAI-Universidad Pontificia Comillas.

RESUMEN DEL PROYECTO

1 Introduccción

1.1 Problema a resolver

Todos los amantes de los deportes de motor se ven unidos por una pasión por las carreras y la velocidad. Disfrutan viendo el deporte, pero también practicándolo. Sin embargo, practicarlo nunca ha sido tarea fácil. El precio, espacio y preparación necesaria para correr en un circuito son los mayores obstáculos. Por ello, los aficionados buscan otras alternativas. Entre ellas, los karts.

Los mayores aficionados consideran los karts como su propia categoría del deporte. Les gusta competir y mejorar. Como en cualquier otra carrera, estas se ganan perfeccionando cada curva de la vuelta.

Actualmente, cuando un corredor de karts tiene cierta experiencia, mejorar se convierte en una tarea difícil. Practicar siempre ayuda; sin embargo, para mejorar más, un conductor necesita retroalimentación. Necesita información indicando que hace bien y que va mal. Un circuito de karts típicamente solo

proporciona el tiempo de la mejor vuelta de la ronda de 10 minutos. Algunas pistas excepcionales dan información de tiempos sobre todas las vueltas. Aún así, es muy difícil extraer información útil de estos pocos datos. La mayoría de las mejoras provienen de la intuición del piloto.

Entonces, ¿por qué no dan más información las pistas de karts? Principalmente, debido a costes. Este es un mercado nicho, dirigido solo para los pilotos serios. Por ello, no muchas opciones han sido desarrolladas.

1.2 Estado de la técnica

La opción más reconocida es RaceCloud [Rac19a]. Esta solución usa un procesador de alto rendimiento capaz de procesar toda la información en tiempo real y fusionar los datos del acelerómetro,giroscopio y el GPS que lleva incorporados. Simultáneamente, es capaz de mostrar los datos en una pantalla montada en el kart, y enviarlos a través de una antena 3G para mostrar los datos en una aplicación de iOS. El resultado es impresionante. Se consigue una precisión alta y unas velocidades de transmisión de datos sorprendente. El problema es la complejidad y el precio de la solución. El montaje requiere la integración de una grande unidad de procesamiento de datos con dos antenas, una para GPS y otra para 3G. A parte, hay que añadir la pantalla para mostrar los datos. Esto no sería un problema tan grande si pudieses instalar el hardware en un kart propio. Sin embargo, para los que no tienen kart propio, tener que preparar todos los componentes cada vez que se va a correr es una tarea tediosa. También es poco probable que las propias empresas de alquiler de karts incorporen esta solución. El producto no se vende directamente, sino que se alquila por un precio de alrededor $10 por hora de carrera. Esto implicaría un aumento del precio del

alquiler de un kart de alrededor el 15 %. Por lo tanto, RaceCloud solo parece una solución viable en el caso de profesionales de los karts, no para aficionados.

Otra solución que existe en el lado opuesto del espectro es RaceTime [Rac19b]. Esto consiste de una aplicación de Android con un coste de tan solo $4. Usa el GPS interno del teléfono para mostrar posición y velocidad. También usa el acelerómetro para mostrar las fuerzas G que sufre el kart. Sin embargo, no existe ninguna integración entre los dos sensores. Esto implica que la precisión depende totalmente del GPS interno. Lo cual significa baja precisión y baja actualización de datos. En el caso de karts, donde las pistas suelen ser muy pequeñas, esta baja precisión no es suficiente.

El producto que ya existe más parecido al objetivo que intenta cubrir este proyecto es el Mychron5 [Myc19]. En vez de usar verdadera telemetría, el Mychron5 guarda datos. Nunca manda los datos fuera del kart, sino que los muestra en una pantalla montada en el kart y los guarda en una memoria interna. Estos datos después se pueden descargar y analizar en un ordenador. Sin embargo, Mychron5 básicamente consta de un buen GPS. El problema de estos GPSs es que los costes que llegan a superar los $300, lo cual lleva a que el precio final del producto sea de $450. De nuevo, esto sería una opción excelente para los karts, pero el precio hace que sea inaccesible para aficionados.

1.3 Objetivos del Proyecto

La actual situación del mercado es lo que motiva este proyecto. El objetivo principal será crear un dispositivo que da información sobre el tiempo de la vuelta, posición, velocidad y aceleración (las variables más importantes en

las carreras) a un precio asequible. Esto permitirá que los aficionados a los karts sigan mejorando.

Esto se intentará conseguir apoyándose en software. Hardware implica piezas físicas. Cuanto más precisas sean, más caras serán. En cambio, reproducir software es algo muy fácil. Además, lo más importante es que permitirá fusionar datos de distintos sensores.

La fusión de datos implica recoger datos de distintos sensores y combinarlos para obtener la mejor estimación posible. Al combinar distintos sensores, se pueden hacer predicciones con menor error que cualquiera de los sensores de forma individual.

Por ello, la idea del proyecto es la siguiente: durante la carrera, un sistema embebido recogerá datos de una unidad de medición inercial (IMU) y un GPS. Estos datos se guardaran en una tarjeta SD. Cuando acabe la sesión, la tarjeta SD se insertará en un ordenador. Ahí, todo el procesamiento de datos tendrá lugar. Esto implica calibrar los sensores, aplicar un filtro Madgwick para estimar la orientación y un filtro Kalman para estimar la posición, velocidad y aceleración.

Se fijarán tres objetivos, correspondiendo a las tres fases del proyecto. El primer objetivo será la recogida de datos. Es la única parte que se basa puramente en hardware. Esto implica leer y guardar los datos en intervalos de tiempo constantes, y optimizar el código para poder leer los sensores más rápidamente.

En siguiente lugar, la calibración de los sensores tendrá lugar. Esto implica leer los datos de la tarjeta SD en el ordenador y prepararlos tal que el giroscopio no tenga sesgo [BK16], el magnetómetro corrija las distorsiones de

hierro blando y duro [Ozy15], y se encuentre un sistema de referencia fijo en el espacio, compuesto de ejes norte, oeste y arriba (NWU, de las siglas en ingles).

Por último, el proceso de fusionar los datos del IMU y el GPS tendrá lugar para dar la mejor estimación de estado posible.

2 Metodología

El primer objetivo esta relacionado con el sistema embebido. La primera pregunta es que sensores utilizar. El método de estimación de estados mediante fusión datos que se suele emplear es usar un filtro de Kalman, que típicamente integra los datos de un IMU y un GPS [FGM11].

Estos dos sensores se complementan de forma ideal. El GPS ofrece una velocidad de actualización baja y una precisión baja. Al contrario, el acelerómetro llega a mandar datos más de 1000 veces por segundo y tiene muy buena precisión. El problema que tiene es conocido como el drift. Para obtener la posición a partir de la aceleración hay que integrar dos veces. Esto implica que cualquier pequeño error crece cuadráticamente, y pierde la referencia original. Por ello, un GPS es el complemento ideal, ya que da datos sobre posición absoluta, lo cual ayuda a compensar el ”drift”[MPU19].

A la hora de elegir un módulo de GPS, la consideración principal era precio, ya que esta parte puede ser extremadamente cara. El módulo NEO6M GPS está disponible por un precio de alrededor $15 y es comúnmente usado. Viene con software que permite configurarlo con facilidad, y está respaldado por una gran comunidad de drones[Ubl11].

Como el IMU tiene que compensar por la imprecisión del GPS, un acelerómetro rápido y preciso era necesario. También era importante que incorporase un magnetómetro, ya que este después sería crucial para la estimación de la orientación del dispositivo. El MPU9250 ($15) satisface estos requisitos [MPU19].

Para poder recoger los datos, un microcontrolador (MCU) es necesario. A la hora de elegir un MCU, se buscó una alta velocidad, precio bajo, y la inclusión de los periféricos de protocolos de comunicación más comunes (UART,SPI,I2C).El STM32F103C8 es un MCU de 32 bits barato ($2) con un reloj de hasta 72MHz[STM15] . Es el procesador que se usa en la común placa de desarrollo conocida como ”Blue Pill”.

A la hora de guardar los datos, se consideraron las opciones de mandarlos de forma inalámbrica o guardaros en una memoria interna al sistema embebido. Mandar los datos de forma inalámbrica llama mucho la atención, ya que permite ver datos en tiempo real e imita a la telemetría de la Formula 1, que es de donde proviene esta idea. Sin embargo, el objetivo de este proyecto es bajar el precio y la complejidad. Por ello, se consideró que una solución más lógica era guardar la información en una tarjeta SD. Esto simplifica el proyecto por dos razones. En primer lugar, no se necesitará un nodo receptor, lo cual simplifica el hardware. En segundo lugar, esto permite analizar los datos sin tener restricciones de tiempos, lo que permite una solución de software más flexible. Los inconvenientes son pocos, ya que es poco probable que el mercado objetivo (aficionados a los karts) tengan a un técnico fuera del kart para preparar el nodo recibidor y analizar los datos en tiempo real. Tener datos en tiempo real es solo útil en ambientes más competitivos, donde ingenieros de carrera pueden usar

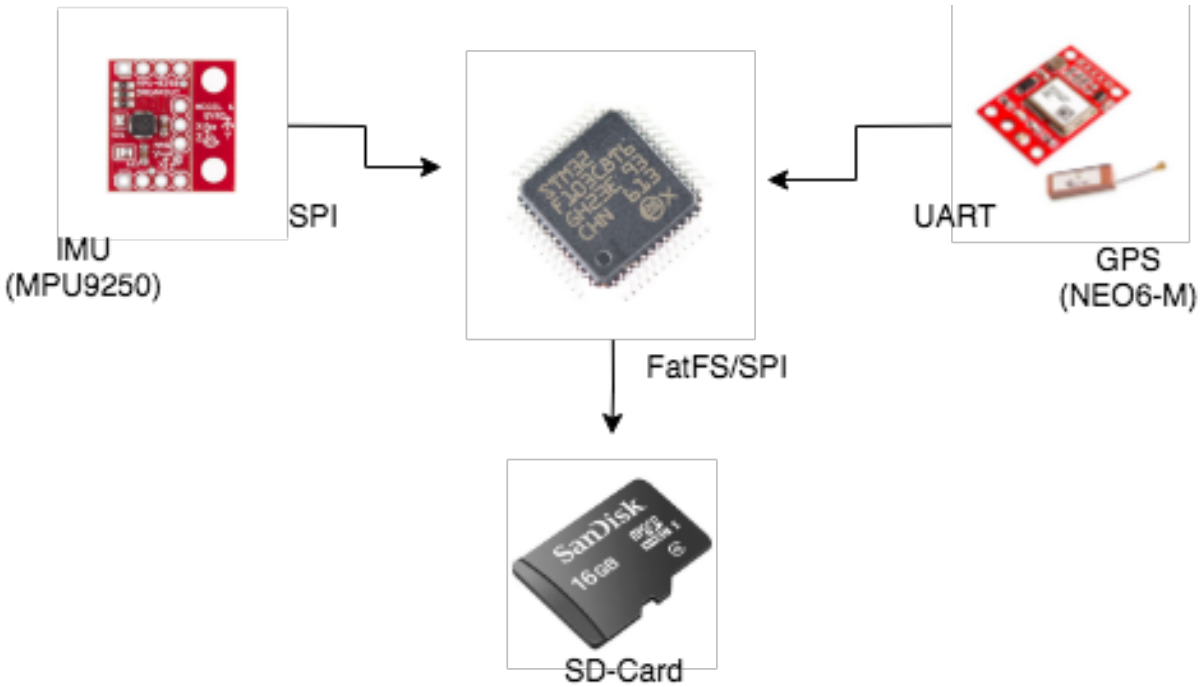


Figura 1: Conexiones de hardware con sus protocolos de comunicación.

todo su potencial.

Consecuentemente, una tarjeta SD genérica puede usarse para guardar los datos. Como el microcontrolador seleccionado no tenía periféricos de comunicación SD, la tarjeta se uso en modo SPI. Para poder guardar archivos, el módulo FatFS fue empleado [Cha19]. La figura 1 muestra una visión general de las conexiones hardware y sus protocolos de comunicación.

Un programa escrito en C en el microprocesador se encargará de recoger todos los datos en intervalos de tiempo constantes.

Una vez se hayan guardado los datos en la tarjeta SD, comenzaremos el siguiente objetivo: preparar los datos para fusionarlos. Este proceso se hará plenamente en software. Como el tiempo de ejecución ya no es una gran preocupación, se usará el lenguaje de alto nivel Python. Este código leerá los archivos de la tarjeta SD y calibrará los sensores.

La primera y más simple calibración ocurre con el giroscopio. Estos tienden a tener un sesgo constante. La media de una serie de datos estacionarios se puede calcular para estimar el sesgo y deshacerse de el.

En caso del magnetómetro, se deben compensar las distorsiones de hierro duro y blando. Antes de calibrar el magnetómetro, este tendrá datos con forma elíptica que no estarán centrados alrededor del origen. Sin embargo, el resultado esperado del campo magnético de la tierra debería ser una esfera perfecta centrada en el origen. Por ello, estos datos serán ajustados a una esfera por el método de mínimos cuadrados[Ozy15].

El último caso de calibración ocurre para los ejes no inerciales. El sistema de coordenadas que se empleará es norte, oeste y arriba (NWU). Este sistema de regla de la mano derecha es ideal ya que cuadra perfectamente con los ejes del sistema de coordenadas de GPS. Esta calibración se hará utilizando solo la información del IMU. El vector de la gravedad se usará para hallar dos de los tres ángulos de Euler. El magnetómetro servirá para encontrar el último ángulo[BK16]. Esto dará una matriz de rotación que permite encontrar la referencia NWU.

Por último, en esta fase también se llevaran acabo todas las conversiones de unidad. El giroscopio a ^{o}/segundo y longitud y latitud a metros. La segunda conversión se hará empleando un modelo que aproxima una tierra plana, el cual es válido para pequeñas distancias como las de un circuito de karts.

La última sección es donde se filtrarán y fusionarán los distintos sensores. Se emplearán dos filtros.

En primer lugar, se debe determinar la orientación del dispositivo. Co-

mo el sistema embebido va subido al kart, tiene ejes no inerciales. Es necesario saber como ha girado el kart para poder relacionar las medidas de aceleración a los ejes NWU. El uso de un filtro Madgwick [Mad10] permitirá estimar esta nueva orientación. Esto se consigue mediante el uso del giroscopio, el acelerómetro y el magnetómetro. El giroscopio sirve para dar la evolución en el tiempo de la orientación. El acelerómetro y el magnetómetro son vectores de dirección constante que se usan como referencia.

Una vez hallada la aceleración en dirección norte y oeste, datos del GPS y el acelerómetro podrán fusionarse mediante un filtro Kalman [FGM11] y dar una estimación de posición, velocidad y aceleración.

3 Resultados

3.1 Primer objetivo:

El primer objetivo era guardar los datos correctamente en la tarjeta SD. Inicialmente la transmisión de datos se hizo mediante UART, permitiendo mandar un paquete con aceleraciones y rotación angular cada a 200Hz. El magnetómometro y el GPS se vieron limitados por las propias características del sensor (10Hz y 5Hz respectivamente).

Un paquete se mandaba cada 5 milisegundos con longitud variante dependiendo en que datos estaban disponibles en cada momento. Esto permitió sincronizar los datos del IMU con el GPS y el magnetómetro, ya que se mandaban en el mismo paquete al mismo tiempo.

Cuando se pasó a escribir los datos a la tarjeta SD, como el sistema

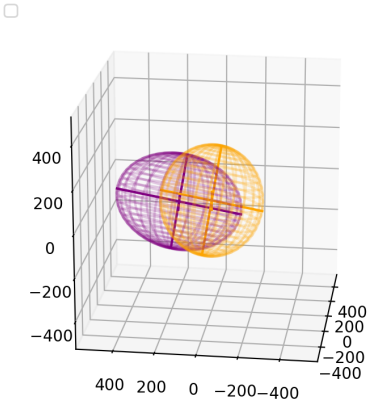


Figura 2: Magnetómetro: sin calibrar(morado) y calibrado(naranja)

se basaba en fatFS para escribir grandes bloques de datos, y este utilizaba el protocolo SPI en modo de bloqueo, la velocidad de mandar paquetes se tuvo que reducir a 50Hz.

3.2 Segundo Objetivo: Calibrar los datos

Las medidas del giroscopio se pasaron a ^{o}/segundo a partir de los valores de bits. Tomando valores en condiciones estáticas, el sesgo y la desviación estándar se pudieron calcular.

Para el magnetómetro, los datos se pudieron calibrar y pasar a encajar en una esfera centrada en el origen y de radio constante. La figura 2 muestra el resultado de calibrar los datos del magnetómetro cuando este se apunta en todas las posibles direcciones en 3D.

El acelerómetro se calibró usando la matriz de rotación para que, independientemente de la orientación inicial del sistema embebido, la aceleración en x e y correspondan con norte y oeste, y por lo tanto sean iguales a 0 en con-

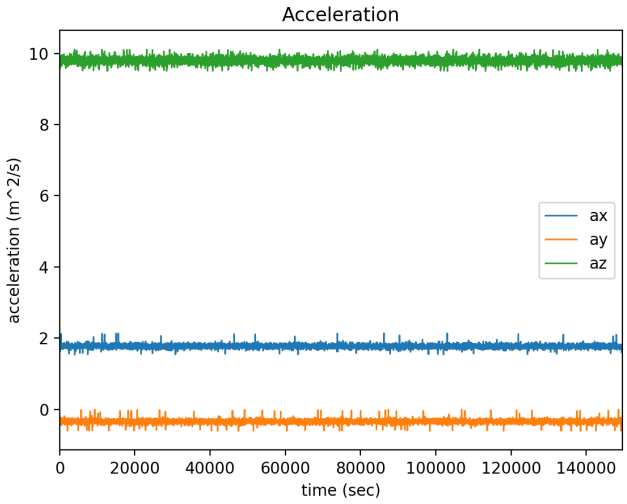


Figura 3: Acelerómetro sin calibrar

diciones estáticas. La figura 3 muestra los resultados de leer el acelerómetro en estado estacionario. La figura 4 muestra los resultados una vez la matriz de rotación se ha aplicado. El resultado muestra ser el deseado.

3.3 Tercer objetivo: Fusión de datos

El primer problema era determinar la orientación del dispositivo. La mejor forma de evaluar esto es con una representación visual de los ángulos de Euler. Usando Pygame, se simuló un cubo en tres dimensiones para que imitase el movimiento del sistema embebido a través de los ángulos de Euler estimados. La figura 5 muestra una imagen fija del cubo. El resultado fue muy satisfactorio, ya que la respuesta era inmediata (sin lag) y no se perdía el estado inicial cuando se volvía a él (sin drift).

Otro experimento que se usó para comprobar la estimación de orientación fue dejar el dispositivo quieto durante un periodo de tiempo y ver como evolucionaba la estimación de los ángulos de Euler. La figura 6 muestra el re-

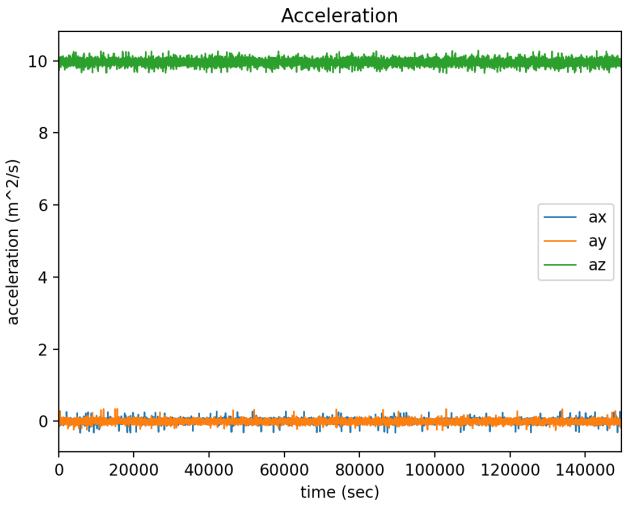


Figura 4: Acelerómetro referido a ejes NWU

Figura 5: Cubo simulado en Pygame

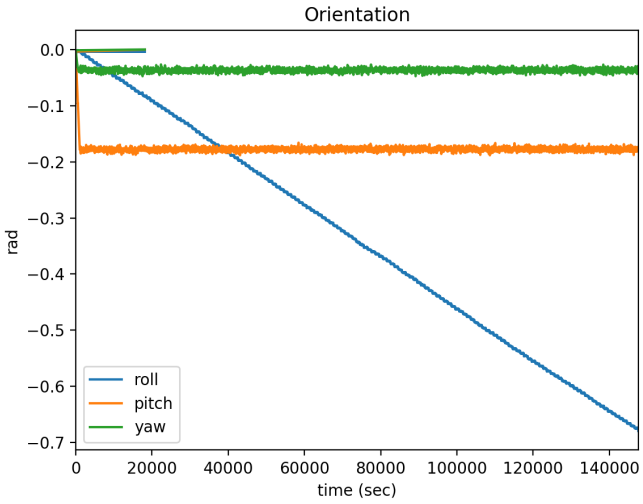


Figura 6: Estimación de los ángulos de Euler sin calibración de los sensores

sultado sin calibrar los sensores, en el cual se ve como hay un claro drift, y se pierde la referencia a los ejes NWU. La figura 7 muestra el resultado tras la calibración, en la que las pocas variaciones de ángulo que hay (menos de 0,03 radianes) siempre se corrigen rápidamente.

Una vez determinada la orientación, se pasó a usar el filtro Kalman para determinar la posición del kart. Sin embargo, el resultado no llego a ser el deseado. La precisión del GPS es muy similar a la precisión de los dos sensores fusionados. El acelerómetro no parece añadir información útil a los datos. Los posibles factores que causen esta falta de precisión se discuten más detalladamente en este proyecto. Sin embargo, la causa más probable es la acumulación de errores a través de cada fase del proyecto acaba en un gran error final.

4 Conclusión

No habiendo obtenido resultados satisfactorios en precisión hace pensar que dispositivos como el Mychron5 pueden justificar precios tan altos. Como el

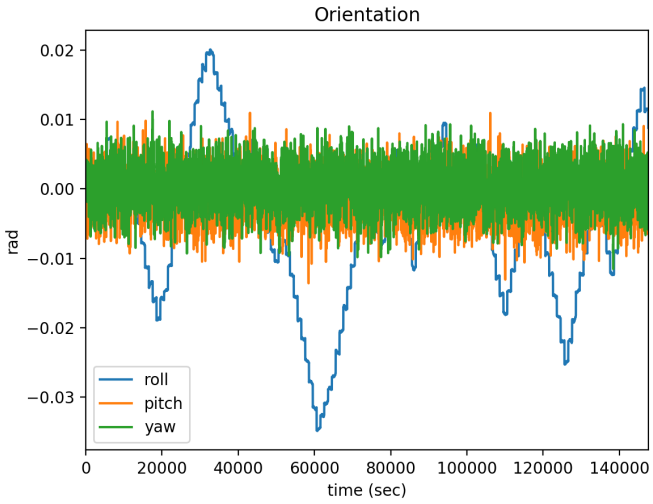


Figura 7: Estimación de los ángulos de Euler con calibración de los sensores

error de cada parte del proyecto se acumula al resultado final (errores propios de los sensores, calibración, orientación, tiempo de actualización...), y se está intentando conseguir una precisión de alrededor de un metro, es difícil conseguir este objetivo si no se prestó especial atención a la exactitud de los resultados durante el desarrollo del proyecto. Por ello, este sería el mayor objetivo si hubiese que enfrentarse a este problema de nuevo.

1 Introduction

1.1 Problem to solve

All motor sports fans are linked by a passion for engines and for racing. They enjoy watching the sport, but practicing it has always been difficult. The cost, space and preparation needed to run on a race track are the major obstacles. Most fans seek other ways to practice the sport. Go Karts are among the top alternatives accessible to anyone.

True fans of motor sports consider this a sport of its own. They like to compete, and they like to improve. So, like in any other race, it is won by perfecting every aspect of your lap.

Currently, after you have certain experience with GoKarts, improving is a very difficult task. Practice always helps; but, to improve, one needs feedback. You need information telling you where you go right and where you go wrong. Typical GoKart tracks only provide information on the best lap achieved during the 10 minute round. Some exceptional tracks will give you times for every lap you had. Even then, it is very difficult to associate what you did to achieve your best lap. Most of your improvement comes from your intuition.

So why are GoKart tracks not providing more information? Primarily, cost. This is a niche market, directed for only the serious GoKarters, which is already a small market in itself. So not many options have been developed.

1.2 Current Solutions

The most recognized option is RaceCloud [Rac19a]. This solution uses a high end CPU that is able to process all the data in real time and fuse GPS, accelerometer and gyroscope readings.It simultaneously renders data to the on-board screen and sends it through an incorporated 3G antenna to display the data live on an iOS application. The solution is very impressive, with high precision and high data through put. Great user interface and compatible with virtually every existing GoKart track. The problem is the complexity and price of the solution. The set-up requires the integration of a large processing unit set up with a large GPS antenna and a 3G antenna. Lastly, the screen with all configurations has to be added. This would be convenient if you had your own GoKart in which you could install the hardware and forget about it. However, if you are going to visit the track occasionally, bringing all the components and setting them up correctly is a tedious task. Rental GoKarts are not likely to use this solution either. The product is not for sale, instead they rent it, with a price of 10$ per racing hour. This would mean increasing the price of racing by around 15%. This solution only seems viable for professional GoKarting, not dedicated hobbyists.

On the opposite end of the spectrum, we find RaceTime [Rac19b]. This is an Android application available for a mere $4. It uses the phones internal GPS to display speed and location. It also uses the accelerometer to display a G-Force meter. However, no fusion between the two sensors exists. This means that position accuracy is totally dependent on the telephone's internal GPS. This implies low accuracy and low position update rate. In the case of Go-Karts, where the track tends to be very compact and narrow, this is not an

acceptable solution.

The most similar product to the one that this work will try to achieve is the Mychron5 [Myc19]. Instead of true telemetry, the Mychron5 is a datalogger. It never sends the data out of the GoKart. Instead, it displays timing data to the driver and saves it to an internal memory which is able to be downloaded and later analyzed on a computer. However, Mychron5 is basically only a GPS with a fairly high update rate (10Hz) and precision($\pm$1 m). However, a GPS of this specifications costs over $300, making the total price of the Mychron5 reach $450. Again, this option would be great for GoKarting, but the price makes the accessory inaccessible for hobbyists.

1.3 Project Objectives

The current market is what motivates this project. The main goal will be to provide a solution which gives information on lap time, position, speed and acceleration (the 4 most important parameters in racing) at a low cost. This will allow casual GoKarters to continue to improve.

The way to achieve this is to rely heavily on software. Hardware means physical pieces. The more precise they are, the more expensive. On the other hand, software is easily reproduced. Additionally, the most important feature is that it will allow us to achieve data fusion.

Data fusion means being able to take in data from different sensors and combining it to achieve the best possible estimate. By combining different sensors, we are able to make predictions with less error than any single sensor.

Hence the general idea of the project is as follows: during racing, an

embedded system will log data from an IMU and a GPS onto an SD-Card. Once the race sessions is over, the SD card will be inserted into a PC. There, all of the data processing will be carried out. This includes calibrating the sensors and applying a Madgwick filter for attitude estimation and a Kalman filter for position, velocity and acceleration estimation.

Three clear objectives can be set for this project. The first objective, corresponding to the first phase of the project will be the retrieval of sensor data. It is the only part relying on hardware. This means reading and logging data at constant rates and optimizing code to be able to read data faster.

Next, the calibration of sensors will take place. This means reading the SD card from the PC and preparing the data so that the gyroscope has no constant bias[BK16], the magnetometer corrects for the soft and hard iron distortions [Ozy15] and, most importantly, a fixed frame of reference independent from our embedded device is found. This frame of reference will be made of the axes north,west and up (NWU).

Lastly, the process of fusing the IMU data with the GPS data will take place to give the best possible estimate for position, velocity and acceleration.

2 Methodology

The first objective covers the embedded device. The first question is what sensors to read. The most common approach with the Kalman filter is integrating an inertial measurement unit (IMU) and a global positioning system (GPS)[FGM11].

Both of these devices are ideal compliments of each other. The GPS offers a slow refresh rate and low accuracy. On the contrary, an accelerometer can reach very high update rates with high precision. The problem with the accelerometer is drift. To calculate position from acceleration, a double integral must be taken. This means that all small noise errors and biases are integrated over time, making them grow quadratically. Hence, the position estimated by an accelerometer ”drifts” over time. Here the GPS is the ideal compliment, as it provides an absolute position, helping to reset the drift[MPU19].

The main factor for choosing a GPS was price, since this part can get extremely expensive. The NEO6M GPS module is available for $15 and is widely used . It has supporting software that allows a detailed configuration and is widely used in the drone community[Ubl11].

Since the IMU has to make up for the low GPS accuracy, a highly accurate and reliable module was needed. Another requirement is the presence of a magnetometer, as this will later be crucial for attitude estimation. The MPU-9250 ($15) satisfies the requirements[MPU19].

To actually retrieve the data, a microcontroller (MCU) is needed. When choosing the MCU, the main consideration was speed, price and peripherals for standard communication protocols (SPI and UART). The STM32F103C8 is a cheap 32 bit microprocessor (around $2) that can be clocked up to 72 MHz [STM15]. It is the microprocessor used in the widely used breakout board known as the ”Bluepill”, meaning there is a large community around it.

When logging data, both the options of sending it wirelessly or logging it on to the device were considered. Wireless transmission is appealing as it

provides real time data and mimics Formula 1 telemetry, which is what sparked this idea. However, considering that the goal of this project is to achieve low cost and low complexity, the more logical process is to save the data on an SD card on the GoKart. This simplifies the project for two reasons. Firstly, we will not need a receiver node. This simplifies the hardware requirements. Secondly, this removes the timing constraints for data fusion and analysis, since the process will no longer need to be real time, allowing a more flexible software solution. The drawbacks are nearly negligible, since it is unlikely that the target market (GoKart hobbyists) will have someone outside the GoKart to setup the receiver node and analyze the real time data. Having real time data is only useful in more competitive racing environments, where race engineers can truly take advantage of it.

Consequently, a generic SD card can be used for storing data. Since the chosen microcontroller does not have a peripheral for SD communications, SPI mode was used to write to the card. The FatFS file manager was used to create files and log data onto the SD card[Cha19]. A general overview of the hardware connections can be seen in figure 8.

A combination of timers and interrupts will ensure data logging at fixed frequencies.

Once this objective is reached, we will begin the next objective of this project. Preparing data for data fusion. This process is all done using software. Since time is no longer a major constraint, the high-level language Python will be used. This code will be in charge of processing the data from the SD card and calibrating the sensors.

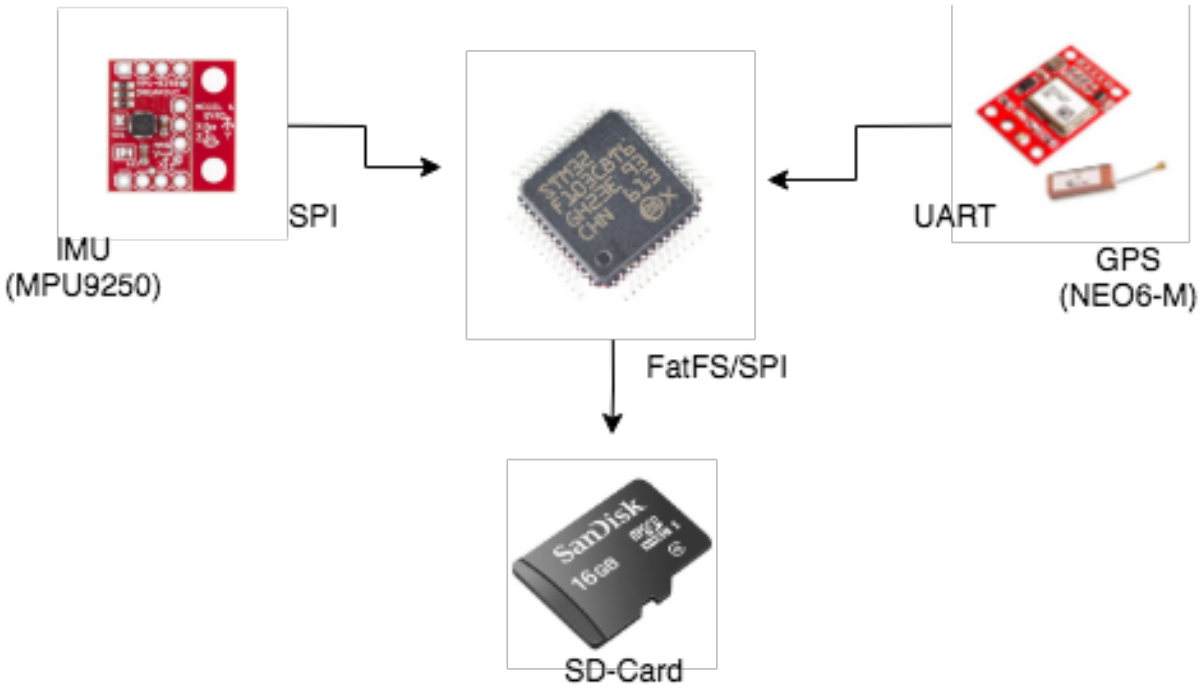


Figure 8: Overview of hardware connections and the respective communications protocol.

The first and simplest calibration will be done for the gyroscope. These tend to carry a DC bias which can be easily subtracted.

In the case of the magnetometer, both hard and soft iron distortions must be compensated for. The uncalibrated output of the magnetometer will be an ellipsoid that is not centered at the origin. However, a sphere representing the magnetic field of the earth is expected. For this reason, the data will be adjusted to fit a sphere through the minimum squares method[Ozy15].

The last calibration is done to find non-inertial axes. North, west and up are ideal in our case because they compose a right handed reference system and match perfectly with the axes already given by the GPS coordinate system. This is done purely using static IMU data. The gravity vector is used to determine roll and pitch. The magnetometer determines yaw[BK16].

In this phase, unit conversion also takes place. Gyroscope readings are

converted into degrees/^{o} and GPS coordinates to meters. The latter conversion is done by approximating the earth as flat, which is a valid approximation for distances handled in GoKart tracks.

The methodology for the last phase corresponds to the application of the data filters. Two filters will be applied.

First, we need to determine the attitude of the device. Since the embedded device is attached to a GoKart, it has a non inertial reference frame that rotates. It is necessary to determine how the object has rotated on the Kart to be able to relate the read accelerometer data to the NWU coordinate system. The use of a Madgwick filter [Mad10], will allow us to relate this non-inertial reference frame of the embedded system with the fixed reference frame found in the previous objective. It achieves this by fusing the gyroscope, magnetometer and accelerometer data. The gyroscope gives evolution over time. The accelerometer and magnetometer act as constant direction vectors to be used as references.

Once the absolute accelerations in the north and west direction are obtained, data from the IMU and GPS can be fused through a Kalman filter [Far12] and provide an estimate of position, speed and acceleration.

The process of creating this project will match the order of the objectives, as these are the logical building blocks. The idea is to have a working objective and then move on to the next one.

3 Results

3.1 First objective:

The first objective was to be able to save correctly timed data. The process was first carried out writing through UART, which allowed a update rate of 200Hz for the accelerometer and gyroscope. The update rates of the magnetometer (10Hz) and GPS(5Hz) were the maximum possible rates of the respective devices.

A package was sent every 5 milliseconds (200Hz) with varying length depending on if the sensors that had been read in the given period. This allowed GPS, magnetometer and accelerometer data to be sent synchronously, with the same time stamp.

When using the microSD, since the system relied on fatFS to write to the chip, a blocking mode SPI data transmission greatly slowled down the refresh rate of the accelerometer and gyroscope to 50 Hz. The other two refresh rates remained constant.

3.2 Second objective: Calibrating data

The gyroscope was converted to ^{o}/second from bit values, and by taking stationary values the offset and the standard deviation were calculated.

For the magnetometer, fitting on to a centered sphere was achieved, with figure 9 showing visual results of a constant magnetic field changing direction while the IMU pointed in different ways.

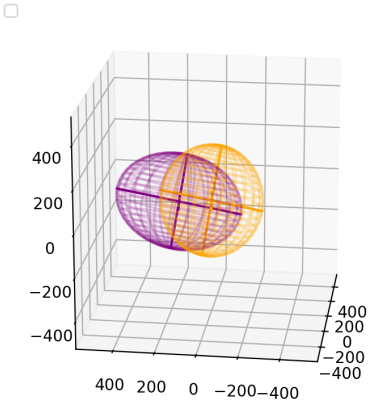


Figure 9: Magnetometer data: not calibrated(purple) and calibrated(orange)

The accelerometer was calibrated with a rotation matrix so that, regardless of the initial position of the embedded device, acceleration in x and y direction are 0 in static conditions and match with North and West, and acceleration in the z direction matches with Up and represents gravity. Figure 10 shows stationary readings of the IMU, and figure 11 shows the values after the rotation matrix has been applied. The results prove to be as expected.

3.3 Third objective:Data fusion

The first problem was determining attitude. The best way to evaluate the result of Euler angles is a visual interpretation. Using Pygame(see image 12, a cube was made to mimic the rotation of the embedded device). The result responded immediately and accurately, and had no orientation drift.

Another experiment that was made was leaving the embbeded device still during a period of time, and seeing how the estimated euler angles change. Figure 13 show the result before calibration in which the yaw angle clearly

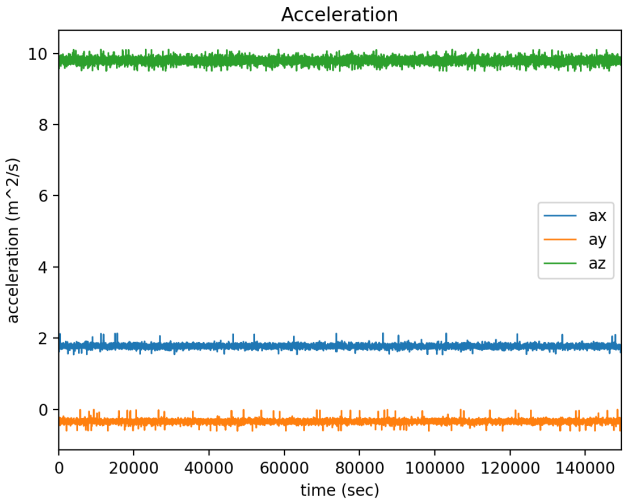


Figure 10: Uncalibrated accelerometer

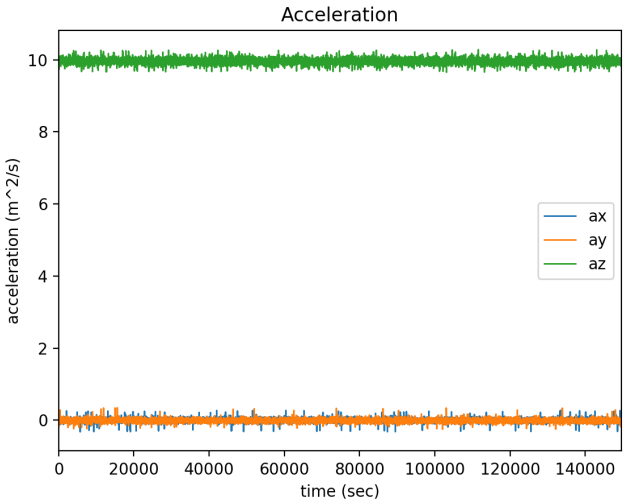


Figure 11: Rotated accelerometer to match NWU axes

Figure 12: Pygame representation of the embedded device

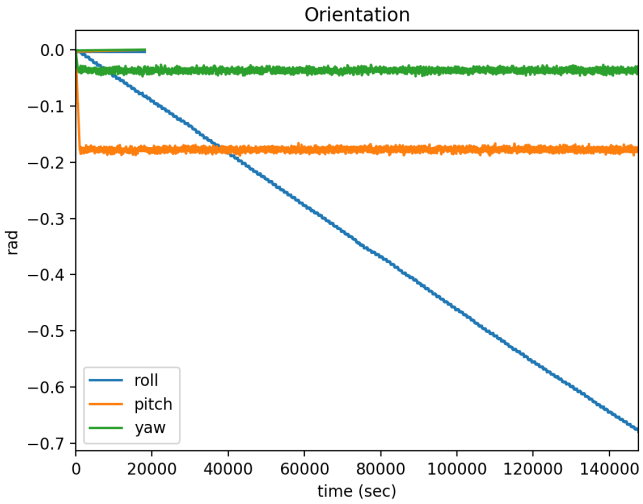


Figure 13: Estimated Euler angles without sensor calibration

drifts. Figure 14 shows the results after calibration, showing that the estimated rotation always returs to the initial possition.

With attitude determined, the Kalman filter attempted to determine the position of the GoKart. However, the result was not satisfactory. The precision of the GPS alone is on par with the precision of the fused estimate and significantly better than only the acceloremter (since it is greatly affected by drift). It is possible that through more accurate calibration and the addition of low pass filters to get rid of the accelerometer jitter, this could be made to work.

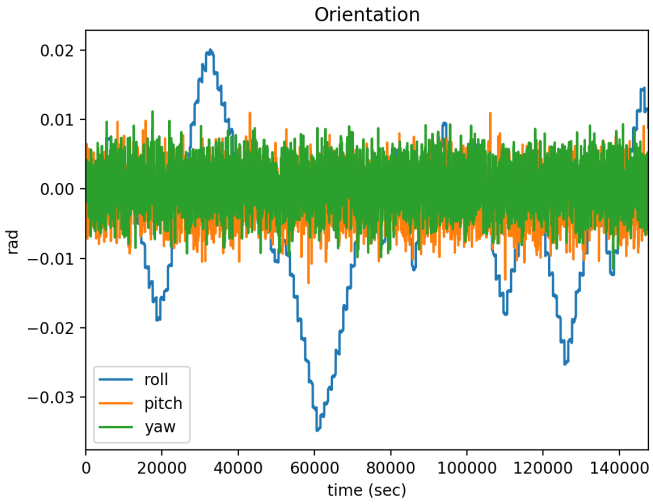


Figure 14: Estimated Euler angles with sensor calibration

4 Conclusion

Not having met the main goal due to accuracy reasons leads to consider that devices such as the Mychron5 can justify their high price. Since every single error accumulates to the final result (errors in IMU and GPS measurments, in timer interrupts, attitude estimation...) and we are attempting to achieve an accuracy of around one meter (typical significant distance in GoKart tracks), it is difficult to do so if special attention was not paid to accuracy. Hence, this would be a major objective if this problem were to be tackled again.

Contents

List of Figures

Chapter 1

Introduction

1.1 Objective

This project will attempt to create an embedded device capable of logging data from several sensor placed on a GoKart. This data will then be extracted and analyzed on a standard computer to give feedback to the driver on the course he took during his time racing. The goal is to provide the most important variables in racing: position, speed and acceleration

1.2 Motivation

Rental GoKarts offer very little feedback to drivers. Hence, lap times quickly plateau. If a driver wants more feedback, most options are aimed at professional GoKart racers or for automotive racing, which are not precise enough for GoKarts. There seems to be a gap in the market that does not meet the needs of hobbyist. This project will try to cover said demand.

1.3 Current Solutions

As mentioned in section 1.2, there are two main solutions to extract race data.

RaceCloud covers professional GoKart racers[Rac19a]. This is a very specialized market. Here, GoKarts act like a feeder league to all other track racing categories. Those who take it seriously plan to go big. Here, teams already have their own GoKart, a method of

Figure 1.1: Race Cloud Hardware [Rac19a]

transporting it, and a mechanic to take care of tuning and reparations. Cost for more performance is usually not the biggest concern. This is why a solution as expensive, precise and complex as RaceCloud makes sense. The way they achieve data logging is through 3G, which already adds complexity and the need for a receiver. They achieve precision through the integration of numerous sensors, but specifically thanks to a highly precise GPS system. The whole set up is bulky (see image 1.1) and expensive(rented at $10 per racing hour).

RaceTime is the opposite side of the spectrum[Rac19b]. The only hardware it uses is an android phone. Basically, it displays data logged from the phone's GPS on a graphical user interface(GUI). There is no sensor fusion and it relies on the cheap GPS inside phones. This means low update rate and low precision. The accuracy it provides is not enough for small tight corners found in a GoKart track.

Chapter 2

Project

Considering the two sides of the spectrum that try to give GoKart racers more information about their laps, there seems to be a clear gap in the middle. This project will attempt to fill said gap of providing precise location tracking but at a reasonable price and with a low number of components.

The way it will do so is by using one single embedded device that will be placed on the a GoKart. This device will log data onto an SD-Card from both an inertial measurement unit (IMU) and a cheap GPS. The SD card will then be inserted into a computer, and software will process the data and give an estimate of position, velocity and acceleration at each given time.

There are two key elements in this project. First, data is never transmitted from the GoKart. This reduces complexity (no need for a receiver or screen) and price. The second and most important factor is that it relies on software. Models will attempt to give good estimates of kinematic values. This means that expensive and highly accurate modules will not be needed.

The process of solving this problem is divided into three main blocks:

- Hardware: this is the development of the embedded system. It will mention how the microcontroller retrieves data from the IMU and GPS and logs it into the SD card.

- Sensor Data Preparation: This is the first step once the SD card reaches the computer. It will explain how the data was retrieved and prepared for the next step.

- Data Fusion: With the prepared data, fusion algorithms will be used to estimate attitude and position of the embedded system.

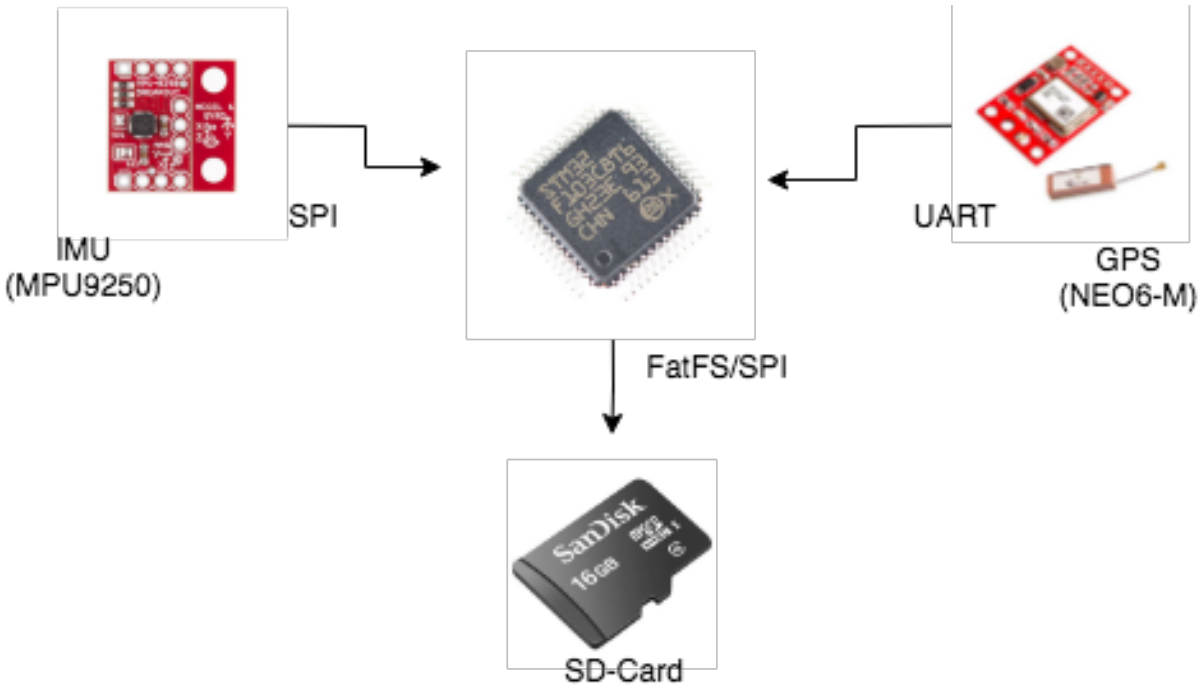


Figure 2.1: Hardware connections with communications protocol

2.1 Hardware

This section will explain everything related to the embedded device. Figure 2.1 shows all hardware connections and their respective communication protocols. Everything will be explained in greater detail below.

2.1.1 Microcontroller

The component in charge of coordinating our sensors is the microcontroller. However, in our case, no complex operations are taking place, it merely routes traffic from the sensors to the SD card. What is critical, however, is timing. Faster update rates and data logging will be crucial to higher accuracy.

A good solution for this problem is the STM32F103C8[STM15]. For around $2, this microcontroller can be clocked up to 72MHz with an external resonator (compared to Arduino's 8MHz). It's 128kB of flash are far more than enough to buffer the sensor data before it is stored in the SD card.

All programming of the device will be done in C, compiled through GCC. Debugging will be done using an ST-Link V2.

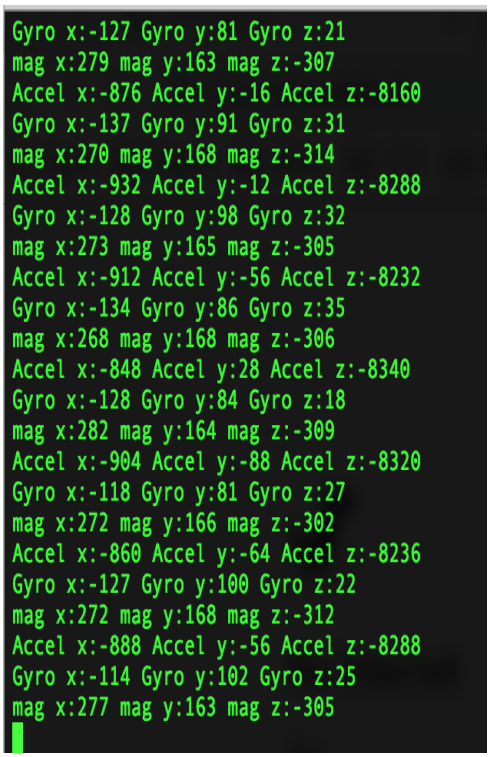


Figure 2.2: *myprintf* onto serial connection.c

2.1.1.1 Initialization

By using STM's hardware abstraction layer (HAL) library, the microcontroller's peripherals are controlled.

In this project we need to initialize two SPI peripherals. One for the IMU and one for writing to the SD card. Although a single SPI peripheral can be used to control both devices, since they need to operate at different speeds and to optimize timing, two different channels were used.

To read data from the GPS module, a UART peripheral was used. However, since the GPS transfers large chunks of data and very slowly, a direct memory access peripheral (DMA) was associated to the UART receiver. Although not necessary for the final product, a useful tool for development was to use a UART peripheral associated to a standard $printf()$ function. By using a UART bridge, real time data and messages could be viewed for debugging. In appendix 5.A code for the $myprintf()$ function used can be seen. It is able to take in strings, integers and floats and display them in ASCII code on a terminal as seen in figure 2.2.

Two timers were also used. One is commonly referred to as the system tick. It is initialized when the microcontroller powers on and ticks at a constant 1000Hz, serving as a time reference for the whole system. The other initialized timer was used to interrupt the

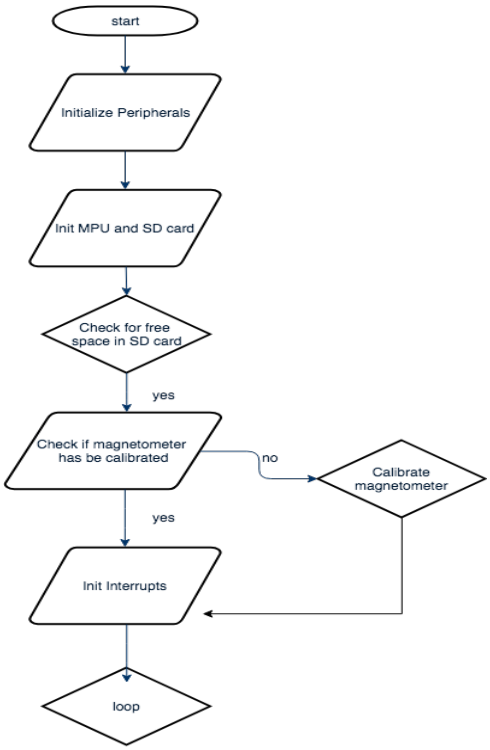


Figure 2.3: Flowchart showing main.c

microcontroler at a constant interval and request to have the IMU data read.

2.1.1.2 Main

The main program in the microcontroller follows the flowchart seen in figure 2.3.

We can see that it initializes the peripherals, followed by the sensors (IMU and SD-Card). It checks if data for magnetometer calibration has already been collected, an does so in case it has not. After that it initializes all the interrupts and lets them take care of the rest of data logging.

Note that although not explicitly shown in the flowchart, each bubble carries its associated error checking.

2.1.1.3 Interrupts

This is the most crucial part of the microcontroller. Precise timing is very important.

There are three interrupts with different priorities. The interrupt with highest priority (excluding hard faults and other exceptions) is the system tick. Since this timer is in

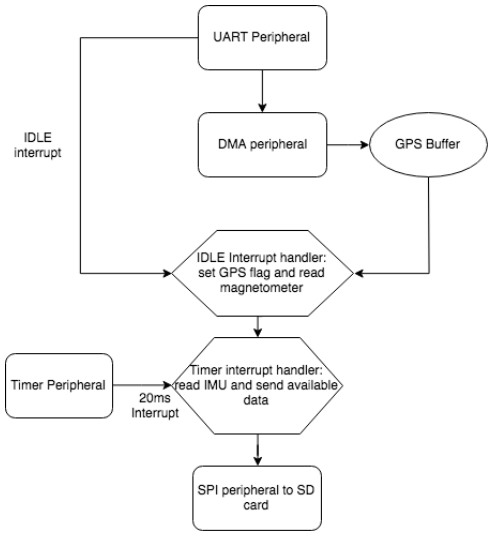


Figure 2.4: Flowchart showing how the interrupts log data

charge of keeping precise timing for the microcontroller, it is crucial that it is never preempted. This means that we will have a one millisecond counter whose precision only depends on the microcontroller's external oscillator.

The second most important interrupt is the second timer interrupt. It has middle priority. It is in charge of providing an interrupt every 20 milliseconds. During this interrupt routine, accelerometer and gyroscope data are read and logged to the SD card. Then the program checks if there is GPS data available. If there is, it is also sent.

The last interrupt has the lowest priority. It must be lower than the second timer so that data can be logged at constant time intervals, which is crucial for accuracy. This last interrupt is in charge of making the data from the GPS available. As mentioned earlier, data from the GPS is stored to memory by DMA. The initial solution that was attempted was to signal that the GPS data is ready when the DMA buffer is full (known as a full transfer interrupt [STM17]). The problem with this solution is that the GPS sends packets of varying lengths. Hence, sometimes you would get a full package and part of the next.

A work around to this problem was found by using the UART's interrupts instead. An idle line interrupt goes off when no bits are transmitted for more than one cycle[STM17]. This means that when a GPS package is received, an interrupt will go off. This interrupt then sets a flag stating that a GPS packet is ready to be sent. It also reads the magnetometer, since they both have similar refresh rates. If this flag is seen by the timer interrupt, GPS data from the DMA buffer and magnetometer data is logged on to the SD card.

Figure 2.4 shows the dynamic of the interrupts.

2.1.2 Inertial Measurement Unit(IMU)

The IMU is a very important piece in this project, since it must be accurate and fast enough to compensate for the sluggishness and inaccuracy of a cheap GPS module. What was needed was a precise, low noise IMU that incorporated a magnetometer for attitude estimation. The MPU9250 fit this role perfectly[Inv16]. It provides digital tri-axial values for acceleration, angular velocity and magnetic field. Additionally, the device can be set to apply low pass filters and different scales.

To interface the IMU with the microcontroller, two communication protocols are available. I2C and SPI. SPI mode was chosen because it is much faster. I2C is a protocol that supports multiple masters. This means that a considerable amount of overhead is needed(I2C address and acknowledge bytes). It also means that masters cannot drive lines through push pull GPIO pins, as this could lead to problems in the case that two masters were to drive the data line simultaneously. Instead, I2C uses an open collector system. This means that data lines can be pulled down instantaneously. However, a pull up resistor is in charge of driving a data line high. Due to parasitic capacitance, there will be a slow rise time associated to the RC circuit.

On the other hand, SPI is an extremely simple protocol that allows one master. There is no overhead and the master can drive lines quickly through push-pull. Clock limits are usually determined by physical distances between master and slave. In the case of the MPU9250, configuration registers can be written at 1MHz and data registers can be read at 20MHz.

Since the STM32F103C8 microcontroller is not as widely used as Arduino, no C library for interfacing the MPU9250 with it using STM32's HAL was found. Instead, this project developed a working library that can be seen in appendix 5.B. Its main functionality will be explained below.

2.1.2.1 Backbone

The MPU9250 has a total of 126 read/write registers. Data can be read from some of these registers. The values written to others serve to configure the device. Hence, the most important functions are reading and writing to these registers. Almost every function in 5.B is based on these low level functions that are able to read from a certain register address or write a single byte to a given register address. Hence $writeReg()$ and $readReg()$ take care of doing this.

2.1.2.2 Initialization

Two initialization functions exist. One for the MPU9250 as a whole, and one specifically for the magnetometer inside it.

These functions enable data registers and set the low pass filters, scale and power consumption mode.

To check that the IMU and magnetometer were initialized correctly, two *whoami*() functions exist. These are in charge of reading read only registers in the devices that have a fixed value. In the case of the IMU, register 117 should always return 0x71. For the magnetometer, the result should always be 0x48[Inv15].

2.1.2.3 Reading data

Both the accelerometer and gyroscope can be read directly in their respective data registers. Each axis is made up of two bytes which are read and returned as a single signed half-word(2 bytes).

The magnetometer cannot be accessed directly. This is because it is its own separate device integrated in the MPU9250 package. Internally the MPU9250 connects to this magnetometer through an I2C interface. Hence, to read the magnetometer, this I2C interface must be controlled by manipulating MPU9250 registers. The result is three compact functions reading each axis of the magnetometer.

2.1.2.4 Euler function

Fusing the three sensors (accelerometer, gyroscope and magnetometer) can give a good estimate of attitude using Madgwick's filter [Mad10a]. However, this function was finally not used since all data processing was taken care on the PC. Consequently, the process of attitude determination will be later explained in detail in section 2.3.1.

2.1.3 Global Positioning System(GPS)

GPS modules can be extremely accurate. However, accuracy makes the price grow exponentially. Since the hope of this product is to create an affordable system, the low cost

NEO-6M GPS module was used[Ubl11]. This GPS module has a large hobbyist community supporting it, can be easily configured using U-Center software.

The GPS module starts up and automatically begins to send National Marine Electronics Association (NMEA) data. This is a protocol for sending GPS data with a specified structure[Dep12]. Information on satellites, signal strength and other parameters can be obtained. However, in this project the only useful information in longitude, latitude, speed and the GPS time stamp. This is all available in the $GPRMC (minimum recommended data) NMEA message. For this reason, the GPS module was set up to send one GPRMC packet at 5Hz.

An internal data filter was also applied to the GPS data. Since the module will be mounted on a GoKart, this filter assumed small vertical acceleration and complete traction (fair assumption for hobbyist GoKarters as it is important not to slide).

Receiving this data is done through DMA. This is important because the GPS sends large amounts of data at a very slow rate (9600 baudrate). Hence, if we needed to read the data with a blocking function, too much time would be spent doing so. By using DMA, other processes can take place until the data is fully read.

2.1.4 SD Card

The previous sections have explained what data the sensors send and how interrupts take care of collecting it. This next section explains how the data is logged on to the SD card.

2.1.4.1 FatFS Module

To be able to create and write files to the SD card, a file manager module was needed. The File Table Allocation (FAT) system is widely used in the case of SD cards. FatFS provides a generic filesystem module for small embedded systems [Cha19].

How it works is shown in figure 2.5. The application calls the Fatfs module to open, read or write to a specified file. This module then uses a user provided device control (commonly referred to as diskIO) to actually initialize and be able to communicate with the storage device. In this project, a diskIO system for an SD card using SPI communications protocol was created.

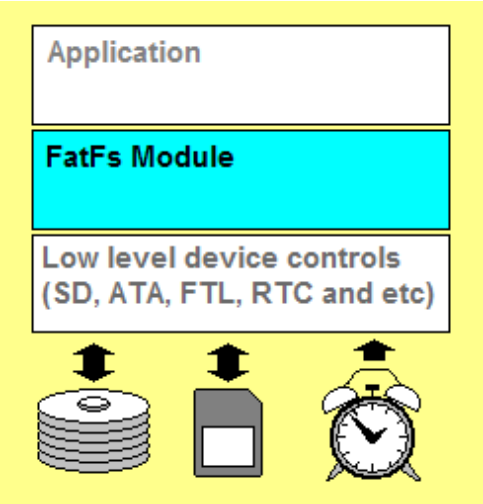


Figure 2.5: Basic structure of an application using FatFS [Cha19]

2.1.4.2 DiskIO

Six functions make up the diskIO interface. However, this project only needed to implement three

- disk_initialize: In charge of initializing the SD card in SPI mode. A flow chart explaining the process can be seen in figure 2.6.
- disk_read: In charge of reading a sector of the card. The minimum sector size in an SD card (smallest readable/writable block) is 512 bytes [Tec]. This will be important later on.
- disk_write: In charge of writing n bytes to an SD card. However, since the minimum manageable block for SD cards is 512 bytes, if n is smaller than the sector size, it must be read completely, overwrite the desired positions, and write the whole sector once again.

Appendix 5.C shows the created diskIO library.

2.1.4.3 Writing Data

Having set up FatFS with its corresponding diskIO, the process of writing data can begin. As mentioned before, it is not possible to write only one byte to the SD card. Only a 512 byte sector can be read or written to. Writing a single byte this way would mean an extremely slow process, which would slow down update rates greatly. A solution to this problem is creating a SD output buffer. When a byte needs to be written, it is appended to a buffer of 512 bytes. If

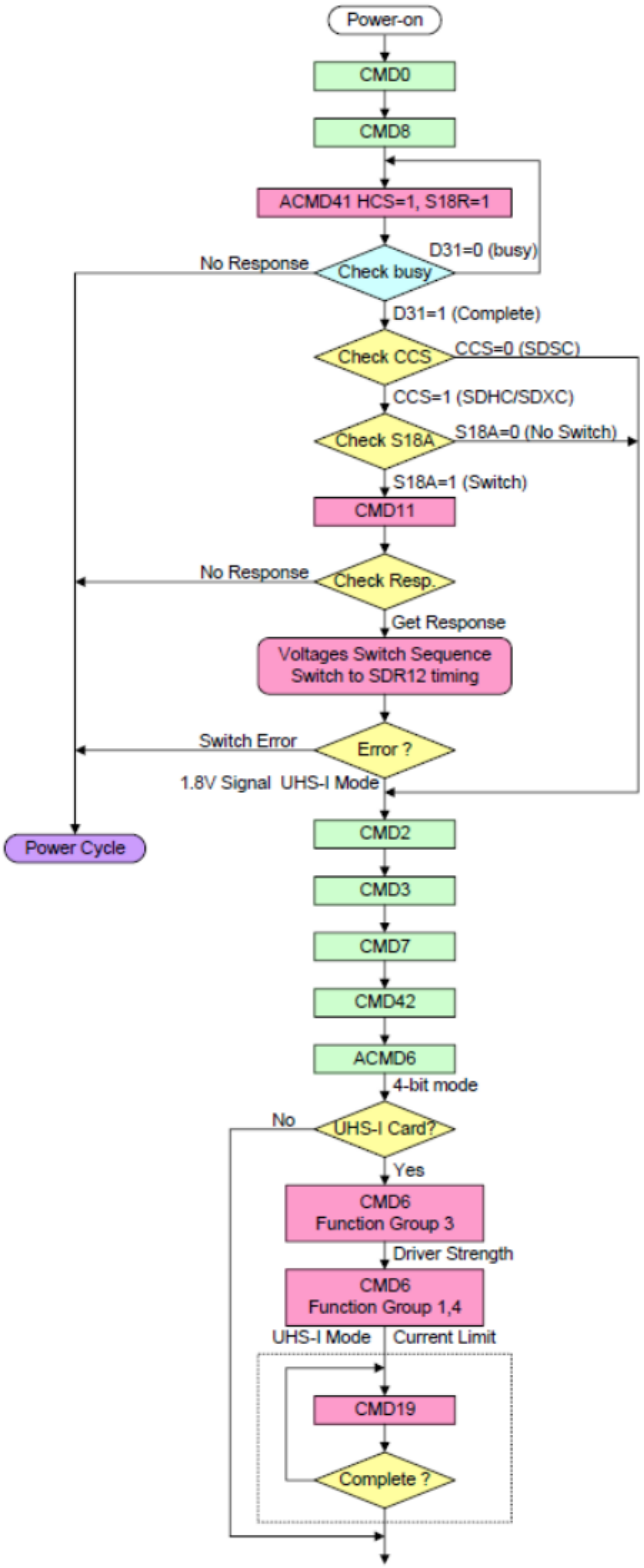


Figure 2.6: Flow chart explaining how to initialize an SD card in SPI mode [Tec]

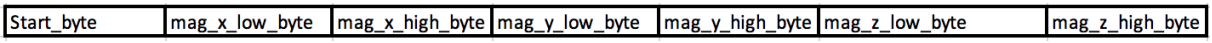


Figure 2.7: Byte structure of Magnetometer calibration data packet

the buffers gets filled, then the sector is written to the SD card and the buffer is cleared. The following code shows the process:

```
void sd_buffer_append(uint8_t data){

  sd_buffer[sd_buffer_index]=data;
  sd_buffer_index++;
  if(sd_buffer_index>=512){
    write_sector(sd_buffer);
    sd_buffer_index=0;
  }

}
```

The next step is actually logging the data. Two files are created in the process.

2.1.4.4 Magnetometer Calibrate

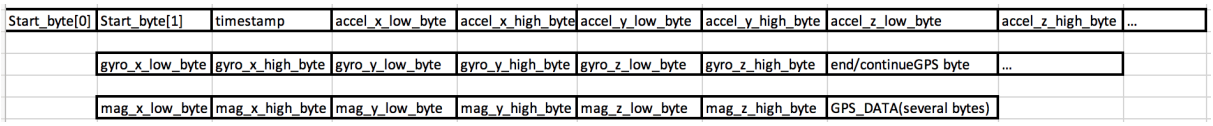
As mentioned earlier, data to calibrate the magnetometer needs to be collected. A *mag_calib*() function is used to create a mag_calib.txt file (if it is not already created). During this process of calibrating, the user must spin the embedded device so that it points in all possible directions.

While the device is being spun, the magnetometer is read in all three axis. A start byte is written to the SD buffer to signal a new data point. Then, the three read half words(in the x,y and z direction) are written to the SD buffer in little endian form. This process is repeated until 500 data points are collected. Figure 2.7 shows how the data packet is structured. This information will later be parsed on a PC and used to calibrate the magnetometer.

2.1.4.5 Data Collection

If everything up to this point has been done correctly, the actual data logging begins. This is done in the timer interrupt handler.

A file called "out.txt" is opened in writing mode. First a start byte is sent to signal a new data sequence. This is followed by a time stamp. The time stamp is taken from the



Start_byte[0]	Start_byte[1]	timestamp	accel_x_low_byte	accel_x_high_byte	accel_y_low_byte	accel_y_high_byte	accel_z_low_byte	accel_z_high_byte	...
	gyro_x_low_byte	gyro_x_high_byte	gyro_y_low_byte	gyro_y_high_byte	gyro_z_low_byte	gyro_z_high_byte	end/continueGPS byte	...	
	mag_x_low_byte	mag_x_high_byte	mag_y_low_byte	mag_y_high_byte	mag_z_low_byte	mag_z_high_byte	GPS_DATA(several bytes)		

Figure 2.8: Byte structure of output data packet

```
YYªEt‡ˇ"âˇıˇ·ˇqÃÆà$GPRMC,190959.00,A,4037.87689,N,00335.73814,W,0.157,,110719,,,A*67
“EYYË¸†¸\~ˇ~PYY(¨ˇò¸8#tˇ¨Íˇ(YY<Ï¨§#∫ˇ¨ˇˇˇ<YYP‰˝‡¸–˝;`PYYdxˇl¸xΩˇw◊ˇdYYxh¸(ˇ‘Û¸¢yYYåÑúˇp!
>ˇWˇ˜ˇçYY†h¸Dˇ‹ˇJD°YY¥$`˝…ˇ`&µYY…ˇ0ˇË Ô¸‘∞ˇq»µÉ$GPRMC,191000.00,A,4037.87688,N,
00335.73816,W,0.065,,110719,,,A*60
–YY›»¸àÑ›ˇF›YYÒÑ˝–˝È¸ÔÒYY
¸¸!„ˇtˇ·ˇYYtˇà\"Üˇ&ˇˇYY–@ˇ Ï!Nˇ4–YYA,¸∞¸Lôˇ À ˇ"AYYU0ˇ–¸ò#iˇˇˇ#UYYi"Üˇëˇ Ÿˇ iYY}l¸îˇT£ˇ¸ˇn}
ŸŸëH¸Xóˇµë YY•,ˇåˇî fi¸1‰ˇ•YYπ` ïˇ¸ˇ:πYŸÕPˇ ¸LÎ¸ã–ÕYY·$Ù˝È ˇÚˇ·YY ı†ˇXˇ¸!'ˇ€ˇ ı YY ¯ˇh<"°ˇ§ˇ0
YYÑˇà¸ºLˇeIYY1–
¿¸l"¶ˇfÌˇ1YYEx(ú#qˇI›ˇEYYYˇ‡ˇÿ›ˇ0XYYYm4ˇx¸(hˇ≠mYYÂÑ¸¥¸lˇÿ˜ˇÅYY†∏'¸T"√ˇS†YY©¿¸Ù¸X6ˇk@©YYΩ¥ˇ
x˝Îˇ#
ΩYY–,¸–¸òˇÂ–ˇ–YYÂlˇ¿»ˇÏˇÂYYˇX˝§¸®ˇô;ˇYY
Ëˇ H¸L »ˇfiÓˇ
YY!Lˇ$#$ˇ~Ÿˇ!YY5ˇ»Ï!é˜X5YYI$¸Ï¸p I˜–ˇHJYY]ƒˇxˇ<$jˇ¯ˇ^YYqº–ò#íˇŒˇÁˇrYYÖ¿¸»ˇ»!
ÿˇtˇ_ÜYYö¸ÑÛˇ¨ˇöYYÆÑ¸ÿ¸l °¸rˇˇÆYY¬úPà!õˇÊˇ¬YY÷Lˇl¸(Mˇ^T÷YYÍ¥ÿ˝@Óˇkıˇ ÍYY¸\ˇåˇ∞
ˇÚ¿ˇ¸YYˇˇíº/ˇí9YY&ê¸®¸º ˇ¸†&&YY:∏ˇ\ˇP!±ˇ¸:YYN88¸Ìˇº%XYYb\¸ú®
◊ˇxˇIbYYvP¸H¸¯àˇüevYYä@¸¿¸@"Bˇ!„ˇäYYûÿ¸x ∂ˇÇˇ q≈µÉ$GPRMC,191001.00,A,4037.87677,N,
00335.73821,W,0.262,,110719,,,A*60
∞YY≤‰¸¸¸@bˇ a≤YYΔ\P¸ºˇN¸ˇΔYY/ˇLˇd;ˇØ›ˇ/YYÓ`‡»!ŸˇÓˇ$ÓYY ¸ ˇúˇ
RYYd‹˝»Ÿˇ;YY*åˇ∞ˇÏ!(ˇL¥ˇ*YY>Lˇ»4 œˇI>YYR‹¸∞¸∞ ˇ0/RYYfxÙ¸º"[ˇ
·ˇfYYzÑÙˇÃ!∏ˇEÒˇzYYéˇ∞ˇÄ!‡ˇ@ˇ@éYY¢ˇÄ¸T§ˇS'¢YY∂»¸`¸ ° ¸†°ˇ∂YY `Ëÿ"ªˇ∑ˇ‡ˇ YYfiúˇ‹¸$
Wˇìafi YYÚ
0¸¿!°ˇöÓˇÚYYˇˇƒˇt ˇÂÈˇYY$| ¿ˇi.YY.º¸<ˇÙ ˇ3/YYB0ˇ‘!
ºˇÁ$CYYVƒTˇ∏#eˇÜ∞ˇWYYuÄˇ†¸∞"·¸¬ˇuYY¸0¸º%ˇ◊EYYìˇ¸¸Ë"Kˇ2·ˇìYYßàˇhhìˇ $ßYYªT¸†¸$!
ïˇ.=ªYYœÁˇ¿ˇ<ıˇÌˇœYY„†˝8¸–&ˇh3„YY˜0à"Öˇºˇ∆ˇ˜YY
<¸¿ˇ@7ˇ«ˇ^
YYH∞˝
```

Figure 2.9: Example of output data (NMEA sentence is highlighted)

previously mentioned system tick timer. Next accelerometer and gyroscope data is written in little endian.

Now the device checks if the flag signaling that GPS data is ready. If it is not available, it appends an end byte of 0x00 to the SD buffer. If data is available, a continuation byte of 0x71 is appended. These values will later be useful to decode the data packets.

If the continuation byte is written, magnetometer data is sent in little endian form followed by the $GPRMC NMEA sentence. This is the only information that actually written using character, not the byte values.

Figure 2.8 shows how the data packet is structured. Figure 2.9 shows an example output file. Note that the only decodable information visually is the $GPRMC string. The rest is bytes that will later need to be decoded and parsed.

Appendix 5.D shows the code for this process. It is part of the second timer's interrupt routine.

2.2 Sensor Data Preparation

From here onward, all code development will be done in Python. Python was chose because it is a versatile, high level language that is great for data analysis. Graphing was done using Matplotlib and arrays were managed using Numpy.

Two data files have now been logged onto the SD file. "mag_calib.txt" and "out.txt". As figure 2.9 shows, this data needs to be decoded and prepared for data fusion. The next section describes said process.

2.2.1 Receive Data

One function was created for reading each of the two files. The function opens the file created by the embedded device for reading. It then proceeds to read bytes until it finds a starting byte sequence. Then it begins to unpack data using Pyhton's "struct" module and appends it to a numpy array. In the case of the "mag_calib.txt", the array has the following columns:

Magnetometer-x	Magnetometer-y	Magnetometer-z

For "out.txt", two numpy arrays are returned. One adds data every start byte sequence. The other only adds data when the GPS data flag had been set. The structures are as follows:

Data1:

Time	X-accel	Y-a	Z-a	X-gyro	Y-g	Z-g	GPS_flag

Data2:

Time	gpstime	lon	lat	vel	x-Magnetometer	Y-m	Z-m

Reading continues until the end of the file is reached, discarding data points that did not arrive completely. Code for this decoding can be seen in appendix 5.E

Note that in the numpy array Data2, values of GPS Timestamp, longitude, latitude and velocity are included. These had been previously parsed from the $GPRMC NMEA sentence using the function seen in appendix 5.F.

All of the data is now available in ordered arrays. It now needs to be fine tuned.

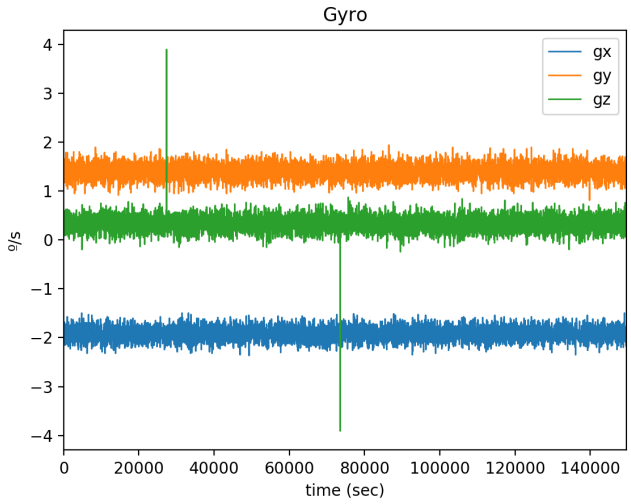


Figure 2.10: Gyroscope measurements (average value is not 0)

2.2.2 Gyroscope

The gyroscope is the easiest element to tune. The values read from the gyroscope can be modeled as:

$$g_{measured} = g_{true} + b + n \tag{2.1}$$

Where b is a constant bias and n is random white noise[BK16]. To correct this bias, we find the average value of the gyroscope readings whilst stationary and subtract them from gyroscope readings to remove the constant bias.

We can assume that the user was standing completely still during the first thirty seconds of data logging. At a logging rate of 50 Hz, we can be certain that the first 1000 data points in our Data1 numpy array were taken in a stationary position. Hence, we can estimate the constant bias as:

$$\hat{b} = \frac{\sum_{i=1}^{1000} g_{measured,i}}{n} \tag{2.2}$$

Where $\hat{b}$ is the estimated bias.

In figure 2.10 we can see the recorded data before removing the constant bias. Figure 2.11 shows the data after the bias has been removed.

Here we also calculate the standard deviation of the gyroscope while stationary for the

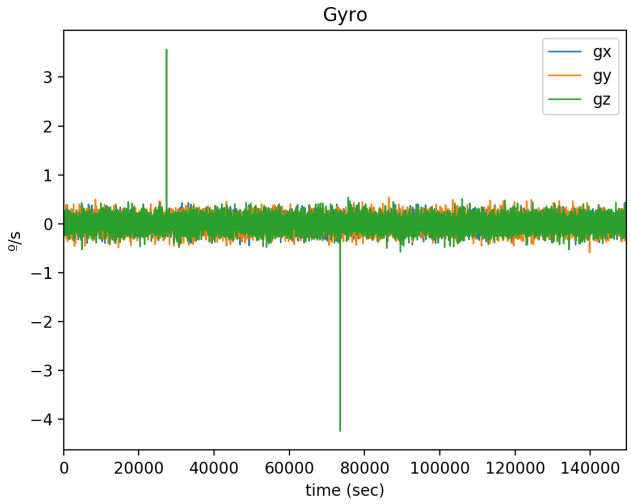


Figure 2.11: Calibrated gyroscope (average values all 0)

three axis, as it will later be useful in the future. The estimation of σ^2 is done by:

$$\sigma_{ii}^2 = \frac{\sum_{i=1}^{N} g_{measured,i}^2}{N-1} \tag{2.3}$$

Similarly the co-variance between different axes is calculated as

$$\sigma_{i,j}^2 = \frac{\sum_{i=1,j=1}^{N} g_{measured,i} * g_{measured,j}}{N-1} \tag{2.4}$$

With $\sigma_{i,j}^2$ being the co-variance between axis i and j

2.2.3 Longitude and Latitude

The data given to us by the GPS is longitude and latitude. However, we need meters in Cartesian coordinates (x and y), as these can be compared with accelerations in meters per seconds.

Since the earth is spherical, there is no exact way to project onto 2 dimensions. However, good results can be achieved if two assumptions are made. First, we assume that the earth is perfectly round. Next, that the change in longitude and latitude is very small. In the case of a GoKart track, this is a good assumption.

Subsequently, we use an equirecangular projection. It states[Alp19]:

$$\Delta x = R * (\lambda_2 - \lambda_1) * cos(\phi_0) \tag{2.5}$$

$$\Delta y = R * (\phi_2 - \phi_1) \tag{2.6}$$

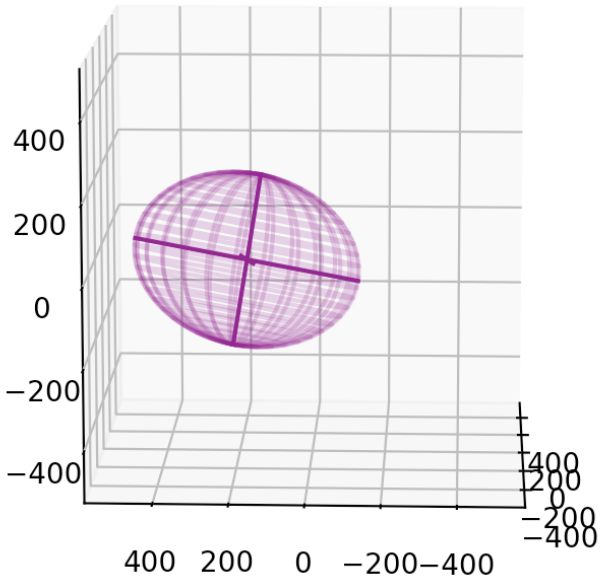


Figure 2.12: Read magnetometer ellipsoid plotted in 3 dimensions.

Where R is the radius of the earth, λ_2 and λ_1 are final and initial longitudes, ϕ_2 and $\phi 1$ are final and initial latitudes. Lastly ϕ_0 and serves as a form correction factor since meridians are closer together at higher latitudes.

Since we only care about the change in displacement, adding or subtraction a constant value to all of the data will have no effect. Hence, we subtract the average position so that our collected data is centered around [0,0].

2.2.4 Magnetometer

When collecting magnetometer data, we spun the embedded device in every possible direction. If the magnetometer was only measuring the earth's magnetic field, plotting the output in three dimensions would give us a perfect sphere with magnitude of the earth's magnetic field. However, when we plot the result we obtain something like figure 2.12.

This is an ellipsoid that is not centered around [0,0]. Two factors cause this deformation[Ozy15].

- Hard Iron: This distortion is caused by permanently magnetized components inherent to the embedded device. For example, if the device's PCB carries a magnet, this will cause a

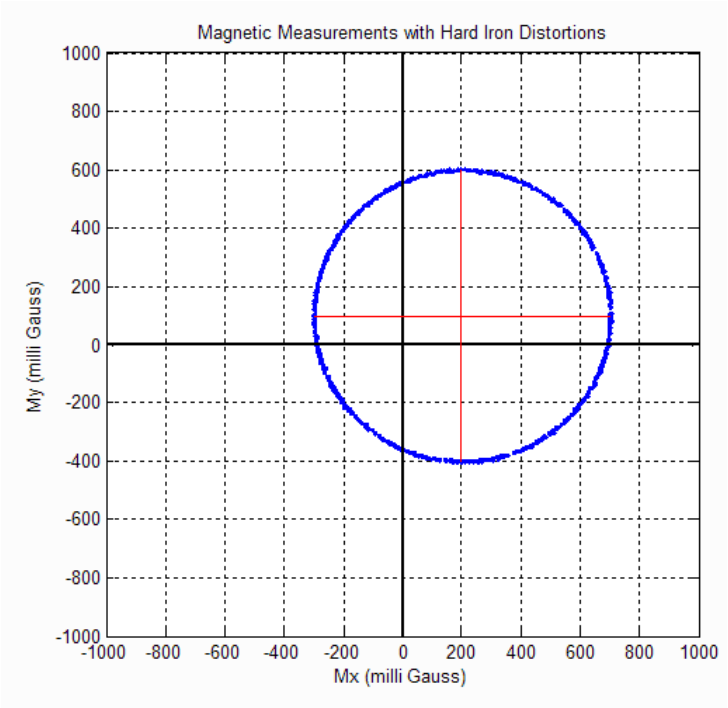


Figure 2.13: Two dimensional effects of hard iron distortion [Vec08].

constant magnetic flux. However, since the embedded device rotates with the magnet(that is to say, they are both attached to the same reference frame), this magnetic flux will be invariant. We can then model hard iron distortion as a constant offset:

$$m_{measured} = m_{earth} + harddistortion \tag{2.7}$$

- Soft Iron: Is caused by materials such as iron and nickel and distort the existing magnetic field. Their effects depends on their surrounding magnetic field, so rotating the embedded device will change the effects of these soft iron deformations. The model for soft iron is as a 3*3 matrix, where the diagonal components will correct scale errors (to convert the ellipse into a sphere) and the rest of the components will compensate for skew angle(field rotations caused by the distortion).

To correct for hard iron distortion, a process similar to what was described in section 2.2.2 is carried out [Ozy15]. The average values are calculated and subtracted as constant biases. Mathematically:

$$b = \begin{vmatrix} b_x \\ b_y \\ b_z \end{vmatrix} \tag{2.8}$$

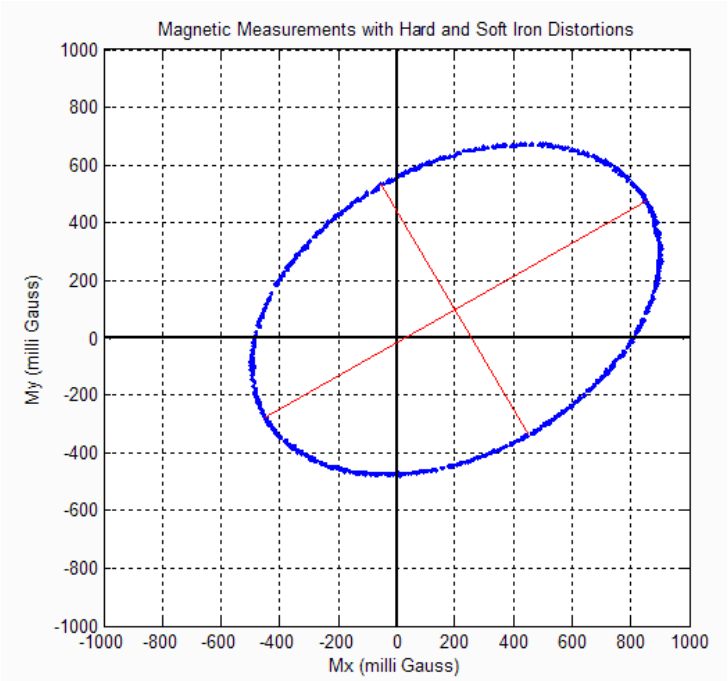


Figure 2.14: Two dimensional effects of soft and hard iron distortion [Vec08].

Soft Iron is more complicated. What should be a sphere gets deformed into a rotated ellipsoid. Mathematically, this can be described with six parameters. A scale factor for each axis(s_x, s_y, s_z) that can be modeled in matrix form as:

$$S = \begin{vmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{vmatrix} \tag{2.9}$$

And three successive rotations around the x,y and z axis. The first rotation of angle ϕ being:

$$R_x(\phi) = \begin{vmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{vmatrix} \tag{2.10}$$

Then around new axis y with angle θ.

$$R_y(\theta) = \begin{vmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{vmatrix} \tag{2.11}$$

Finally around the rotated z axis (axis that has rotated in conjunction with the ellipsoid) with angle ψ:

$$R_z(\psi) = \begin{vmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{vmatrix} \tag{2.12}$$

The combined effect is able to stretch a sphere into any ellipsoid (with matrix S) and then rotate it to any ·D orientation (as described by Euler angle rotations [Cur]). It can be represented with a single matrix as:

$$M = R * S \tag{2.13}$$

where $R = R_z * R_y * R_x$

Hence any point

$$\hat{x} = \begin{vmatrix} x \\ y \\ z \end{vmatrix} \tag{2.14}$$

on a unit sphere can be moved to an origin centered ellipsoid through:

$$x_{ellipse} = M * \hat{x} \tag{2.15}$$

Consequently, the inverse transformation from our measured data is:

$$x_{fitted} = M * (x_{measured} - b) \tag{2.16}$$

By adding the constraint that $x^2_{fitted}[0] + x^2_{fitted}[1] + x^2_{fitted}[2] = 1$ we make sure that the values fit a sphere of unit radius [MIC14].

As a measurement of error, we calculate the minimum distance between x_{fitted} and a point on the unit sphere.

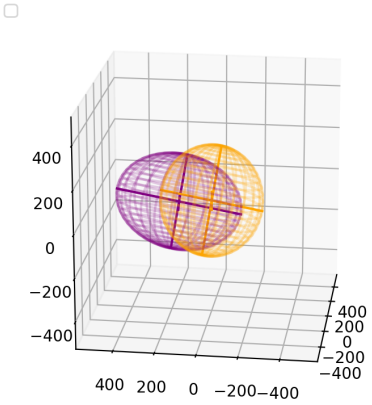


Figure 2.15: Orange: fitted magnetometer data Purple:Read magnetometer data.

A function from [Sem18], is able to take this error and optimize matrix M and vector b to minimize the sum of errors squared for all the points fitted onto the ellipsoid. This is knows as a least squares regression. The function returns the optimal values for matrix M and vector b. The result of applying this transformation to data collected from the embedded device can be seen in figure 2.15 . Note that the fitted data is scaled so that its shape can be visually compared to the non-fitted data. Also note that it is okay to fit the data to a unit sphere because magnitude is not important in this project. The only use found in the magnetometer data is in its direction.

When calibrating the magnetometer it is also important to do so in its final package and installed on the GoKart. This is because the metals in the magnetometer's surrounding environment will affect soft and hard distortions. Hence, where it is installed affects the calibration process.

2.2.5 Non-Inertial axes

The last important calibration is finding non-inertial axes. This is necessary because GPS data gives us values in the north and west directions. However, the initial orientation of the IMU is unknown. Readings of acceleration in x direction do not necessarily correspond with the north direction. Y direction does not necessarily correspond with west. Figure 2.16 shows how the x,y and z axis of the IMU are not aligned with north, west and up directions.

Since the origin of the reference system is not of importance, solving this problem only requires a rotation matrix R (like the one described in section 2.2.4) to align the axis. To find the

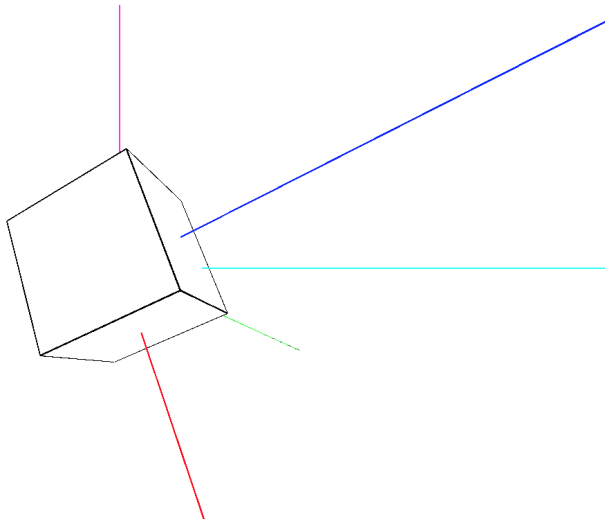


Figure 2.16: IMU axis (x in read, y in green and z in blue) represented with north axis(cyan) and up axis(purple)

Euler angles of this transformation, we begin by calling rotation matrix:

$$R_{IMU}^{NWU} = R_z(\psi) * R_z(\theta) * R_x(\phi) = \tag{2.17}$$

$$\begin{vmatrix} c_\theta c_\psi & s_\phi s_\theta c_\psi - c_\phi s_\psi & c_\phi s_\theta c_\psi + s_\phi s_\psi \\ c_\theta s_\psi & s_\phi s_\theta s_\psi + c_\phi c_\psi & c_\phi s_\theta s_\psi - s_\phi c_\psi \\ -s_\theta & s_\phi c_\theta & c_\phi c_\theta \end{vmatrix} \tag{2.18}$$

where R_{IMU}^{NWU} is the rotation matrix to go from IMU axis to a NWU reference frame, c_α is $cos(\alpha)$ and s_β is $sin(\beta)$.

Using properties of rotation matrix, we know that the inverse transformation (from NWU angles to IMU angles) can be expressed as[Cur]:

$$R_{NWU}^{IMU} = R_{IMU}^{NWU\,-1} \tag{2.19}$$

We know that an IMU measures the negative gravity vector whilst stationary. This vector in NWU axis can be represented as:

$$g_{NWU} = \begin{vmatrix} 0 \\ 0 \\ g \end{vmatrix} \tag{2.20}$$

The IMU reads

$$a_{IMU} = \begin{vmatrix} a_x \\ a_y \\ a_z \end{vmatrix} \tag{2.21}$$

Hence, applying the proper rotation matrix to be able to compare values:

$$R^{IMU}_{NWU} * g_{NWU} = a_{IMU} \tag{2.22}$$

Substituting the matrices for their values and solving:

$$s_\phi * c_\theta * g = a_y \tag{2.23}$$

$$c_\phi * c_\theta * g = a_z \tag{2.24}$$

$$-s_\theta * g = a_x \tag{2.25}$$

Solving for θ and ϕ:

$$\phi = arctan(\frac{a_y}{a_z}) \tag{2.26}$$

$$\theta = arctan(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}) \tag{2.27}$$

It is important to use a two argument arctan function to get a result in the proper quadrant.

To determine the last Euler angle, ψ, we will use the readings from the magnetometer.

Ideally, in NWU angles, the magnetometer would point north. However, there are two problems. Magnetic north and geographic north do not match. The difference between them is measured by an angle called declination. The next factor is that the magnetic field is not always parallel to the earth. Consequently, it has a component in the Up direction. This is measured by an angle known as inclination. Figure 2.17 shows both angles.

Hence, the magnetic field of the earth b can be expressed in NWU axis as:

$$b = \begin{vmatrix} c_\gamma c_\delta \\ c_\gamma s_\delta \\ -s_\gamma \end{vmatrix} \tag{2.28}$$

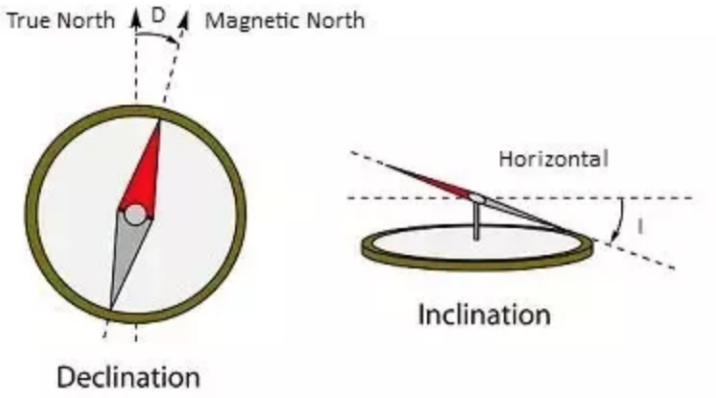


Figure 2.17: Magnetic declination and inclination[Ber17]

where δ is the magnetic declination and γ the magnetic inclination.

In Madrid, Spain, the magnetic declination is -0.5^{o} and the inclination is 55^{o} in the downwards direction[Dec18]. Since δ is such a small value, we can approximate $c_\delta \approx 1$ and $s_\delta \approx 0$. Giving a magnetic field of:

$$b = \begin{vmatrix} c_\gamma \\ 0 \\ -s_\gamma \end{vmatrix} \tag{2.29}$$

The IMU magnetometer measures

$$m_{IMU} = \begin{vmatrix} m_x \\ m_y \\ m_z \end{vmatrix} \tag{2.30}$$

Applying the magnetometer data calibration described in section 2.2.4 and normalizing earths magnetic field b to a unit vector, the following comparison can be made:

$$R_{NWU}^{IMU} * b_{NWU,normalized} = M * (m_{IMU} - bias) = m_{IMU,calib} \tag{2.31}$$

with M and $bias$ being values calculated in 2.2.4. Solving for ψ gives [BK16]:

$$\psi = arctan(\frac{c_\phi * m_{y,calib} - s_\phi * m_{z,calib}}{c_\theta * m_{x,calib} + s_\phi * s_\theta * m_{y,calib} + c_\phi * s_\theta * m_{z,calib}}) \tag{2.32}$$

With ϕ and θ being the previously calculated Euler angles and $m_{IMU,calib} = \begin{vmatrix} m_{x,calib} \\ m_{y,calib} \\ m_{z,calib} \end{vmatrix}$.

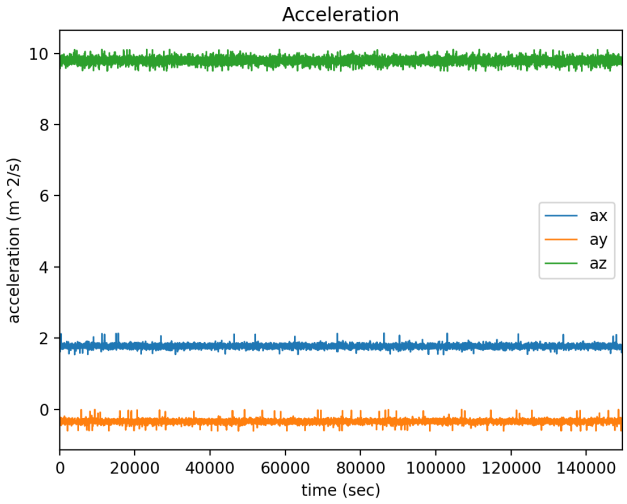


Figure 2.18: Acceleration in IMU axis

Note that g_{IMU} and a_{IMU} are taken as the average values of the first 1000 points of the "out.txt" file, which are assumed to be taken while stationary. $m_{IMU,calib}$ is taken from the data points in "mag_calib.txt" which is assumed to point in every 3D direction while calibrating.

All three Euler angles have been calculated and matrix R_{IMU}^{NWU} can be determined. If we collect data whilst the embedded device is stationary but has an inclination, we will see gravity projected in the three axis, as seen in figure 2.18. When we apply the transformation, gravity is only seen in the up direction (z axis), as shown by figure 2.19

A similar effect occurs with magnetometer data. In IMU axis, it is distributed amongst the three axis. However, in NWU axis, the W component (y direction) will be zero. This can be seen in figures 2.20 and 2.21.

2.3 Data Fusion

The data has been retrieved and prepared. Now, two different sensor fusion algorithms will be used to estimate attitude and position, velocity and acceleration.

2.3.1 Attitude estimation

While the device is stationary, we have been able to refer the IMU axes to north, west and up directions. However, this is not enough. When racing with a GoKart, the device will

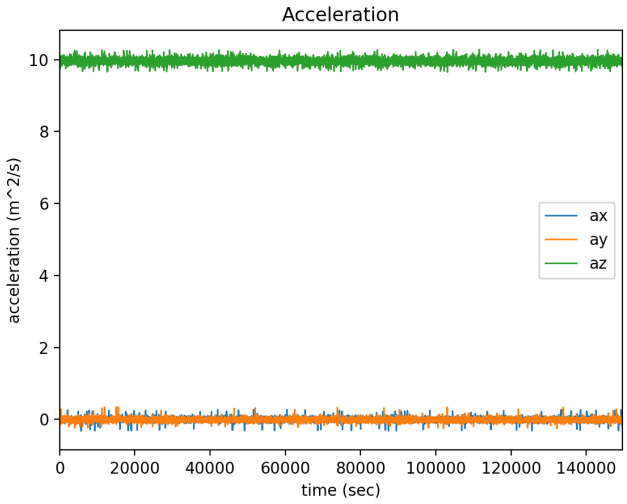


Figure 2.19: Acceleration in NWU axis

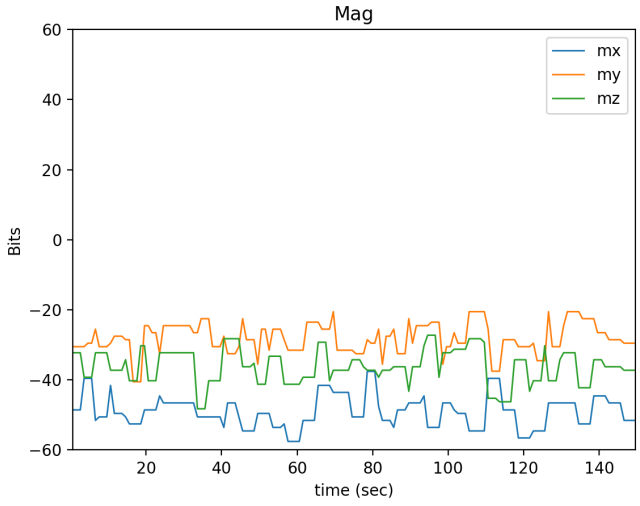


Figure 2.20: Magnetic field in IMU axis

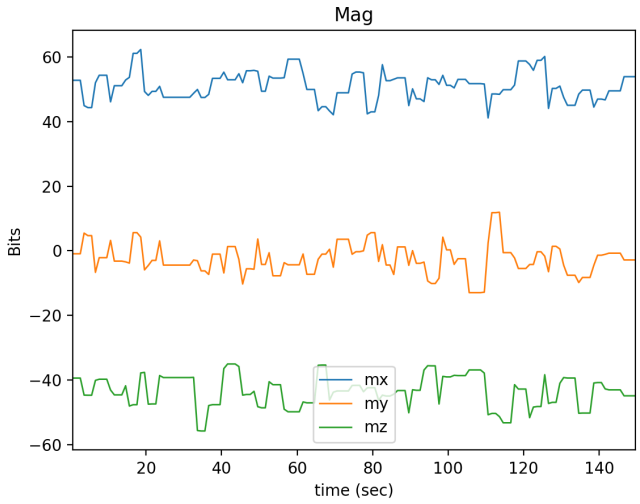


Figure 2.21: Magnetic field in NWU axis

continue to rotate. We need to estimate how much it has rotated from that initial position to refer the axes back to NWU. This is known as attitude estimation. Many commercial solutions exist, most of them applying Kalman filters. However, designing these Kalman filters has proven difficult when there are varying sampling rates. Tuning the numerous parameters present in a Kalman filter and designing a precise model has also proven to be challenging. Madgwick's proposed filter has been proven to be accurate, compensates for drift, can merge IMU data with a magnetometer, and only has two tuning parameter which are determined by observable physical system characteristics [Mad10b; Far12].

This filter relies on quaternions to represent the rotation. Similar to complex numbers, quaternions introduce abstract symbols i, j and k. They follow the general relation:

$$i^2 = j^2 = k2 = i * j * k = -1 \tag{2.33}$$

In the same way that imaginary numbers have proven to be a useful tool, quaternions are ideal for representing 3D rotations. It is used as an alternative to Euler angles because it avoids glitches caused by problems such as gimbal lock(the loss of one degree of freedom due to the alignment of two or more axis of rotation) and interpolation errors[SE18].

Just like unitary complex numbers are useful for representing a 2D rotation, where:

$$P_2 = P_1 * (cos(\theta) + sin(\theta) * i) \tag{2.34}$$

With P_1 and P_2 being the starting and rotated points represented as a vector with complex numbers (i.e. [x,y*i]) and θ representing the rotation angle.

Quaternions have a similar effect but in 3 dimensions, where:

$$P_2 = q * P_1 * q' \tag{2.35}$$

with q being the rotation quaternion q' its conjugate and P_i being represented as a 3D quaternion [0,x*i,y*j,z*k]. If we represent a quaternion q as:

$$q = cos(\theta/2) + sin(\theta/2) * (\hat{x} * i + \hat{y} * j + \hat{z}k) \tag{2.36}$$

This would create a rotation of angle θ around axis $v = \begin{vmatrix} \hat{x} \\ \hat{y} \\ \hat{z} \end{vmatrix}$, in the case that v is a unit vector.

In the case of the Madgwick filter, we choose an initial quaternion of $= \begin{vmatrix} 1 \\ 0 \\ 0 \\ 0 \end{vmatrix}$. This means that, initially, we assume that there is no rotation. This is a valid assumption because, section 2.2.5 already takes care of aligning the IMU axes with NWU.

The Madgwick filter then begins updating this quaternion to represent the new orientation of the embedded device. Hence, to be able to reference IMU readings with respect to the fixed NWU frame again, all that needs to be done is revert the rotation. Keeping with the use of Euler angle matrices through out this paper, we know that[Mad11]:

$$\phi_{mad} = arctan(\frac{2(q_2q_3 - q_0q_1)}{2q_0^2 - 1 + 2q_3^2}) \tag{2.37}$$

$$\theta_{mad} = -arctan(\frac{2(q_1q_3 + q_0q_2)}{\sqrt{1 - (2q_1q_3 + 2q_0q_2)^2}}) \tag{2.38}$$

$$\psi_{mad} = arctan(\frac{2(q_1q_2 - q_0q_3)}{2q_0^2 - 1 + 2q_1^2}) \tag{2.39}$$

With $q = \begin{vmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{vmatrix}$. We can then construct the rotation matrix:

$$R_{NWU}^{NewOrient} = R(\psi)_{z,mad} R_(\theta)_{y,mad} R_(\phi)_{x,mad} \tag{2.40}$$

With $R(\alpha)_{i,mad}$ representing the rotation matrix of Euler angle α predicted by the Madgwick filter around axis i. $R_{NWU}^{NewOrient}$ represents the rotation matrix from NWU reference frame to the estimated current orientation of the embedded device. Then:

$$R_{NewOrient}^{NWU} = R_{NWU}^{NewOrient-1} \tag{2.41}$$

With $R_{NewOrient}^{NWU}$ representing the rotation matrix to pass current IMU data back to the NWU reference. Consequently, to get data in NWU reference:

$$a_{NWU} = R_{NewOrient}^{NWU} * a_{IMU} \tag{2.42}$$

This data now has the same reference as GPS coordinates.

Note that this process could have been done using quaternions, since a rotation of $q * P * q'$ is exactly equivalent to a rotation around of Euler angles ϕ,θ and ψ if these angles were calculated using equations 2.37

The process the Madgwick filter uses to estimate current quaternion orientation is as follows:

By taking in the gyroscope readings, the rate of change of the quaterion at time t estimated with w ($\dot{q}_{t,w}$) can be calculated as:

$$\dot{q}_{t,w} = \frac{1}{2}\hat{q}_{t-1} \otimes \bar{w}_{t-1} \tag{2.43}$$

Where $\hat{q}_{t-1}$ is the estimated quaterion at the previous instance, $\otimes$ is a quaternion multiplication and $\bar{w}_{t-1} = [0, w_{x,t-1}, w_{y,t-1}, w_{z,t-1}]$ represents the gyroscope readings at time $t-1$. This w vector acts as the axis of rotation. To get the quaterion at time t estimated by w ($q_{t,w}$), numerical integration is carried out:

$$q_{t,w} = \hat{q}_{t-1} + \dot{q}_{t,w}\Delta t \tag{2.44}$$

Where Δt is the update rate of the gyroscope.

Another estimation is carried out using vector observation. The detailed derivation can be seen in [Mad10b]. In short, it uses both the magnetometer and the accelerometer as constant magnitude vectors with fixed directions in NWU axes.

The result is an estimated quaternion from accelerometer and magnetometer data (when available): $q_{t,(a,m)}$. As seen in section 2.2.4, the accelerometer alone can estimate ϕ and θ Euler angles. Without a magnetometer, ψ is estimated with the gyroscope, which is subject to drift. The magnetometer can compensate for this drift, even though it has slower update rates.

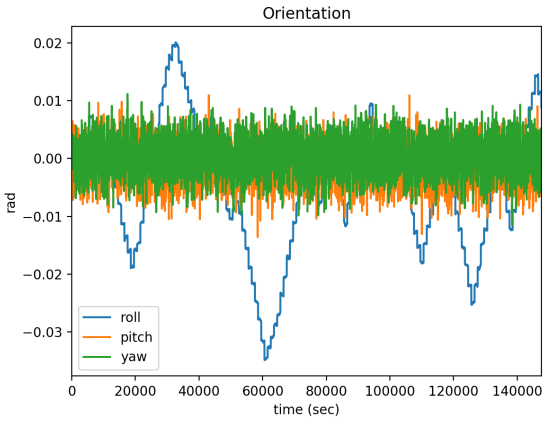


Figure 2.22: Estimated Euler angles while stationary

Finally, the two estimates are fused through a weighted average:

$$q_t = (1-\gamma)q_{t,w} + \gamma q_{t(a,m)} \tag{2.45}$$

γ can be calculated as a function of β and α, the two filter parameters. These two parameters do not have an explicit units. Instead β is related to gyroscope error (bias and variance) and α is related to horizontal accelerations which change the apparent direction of gravity. Higher β will mean faster gyro error compensation but higher sensitivity to horizontal acceleration . Higher α has the opposite effect. Several values were tested until what seemed like an optimal solution was reached.

Appendix 5.G includes the code for estimating attitude and referring the collected data back to NWU axes. The madgwick function call is taken from [Mad10b]. All other functions are self-defined and available at [Ale19].

To check the accuracy of this code, figure 2.22 plots the estimated Euler angles when the embedded device was left completely still. The results show that there is no drift (even yaw is corrected back to 0) and the max variation of precision is only of around .03 radians.

However, this was done during static conditions. Creating an accurate test bench for a dynamic estimation is not an easy task, since creating precise angular rotations requires a complicated setup. Instead a real time visual representation of the embedded device was created using Pygame. This allowed to compare the physical orientation of the IMU device with the visual representation on the PC.

The embedded device outputs IMU and magnetometer data in real time through UART. A UART bridge was used to connect to the PC. In Python, a serial connections was established and the IMU data was read. Finally the Madgwick filter was applied. Using Pygame, a three

Figure 2.23: PyGame visualization of the embedded device

dimensional cube was drawn using code from [Mac11]. This cube was then rotated and displayed according to the estimated Euler angles. The results were very satisfactory. There was no noticeable lag, and when returned to its initial position after harsh movement, there was no noticeable drift. Code for this can be found in appendix 5.H.

2.3.2 Position estimation

Having estimated the attitude of the GoKart, which allows us to find accelerations relative to the NWU reference frame, we are able to fuse position data from the GPS with acceleration data from the IMU. The most common approach to sensor fusion of this type has been the use of a Kalman filter [FGM11].

The Kalman filter serves to estimate a non-observable state of a linear dynamic system subject to white noise. A common example problem where a Kalman filter could be useful is in measuring the temperature inside a combustion chamber. Inserting a temperature sensor inside the chamber is not a viable option (hence, it is a non-observable state). By measuring water temperature of the cooling system and applying heat transfer knowledge(giving us the dynamic system model), a Kalman filter is able to give the best estimate of temperature at a given time t.

In our case, our hidden variables are position, velocity and acceleration. GPS and accelerometer data, along with knowledge of kinematics (representing the linear dynamic system), will help us find the best estimate for these hidden variables.

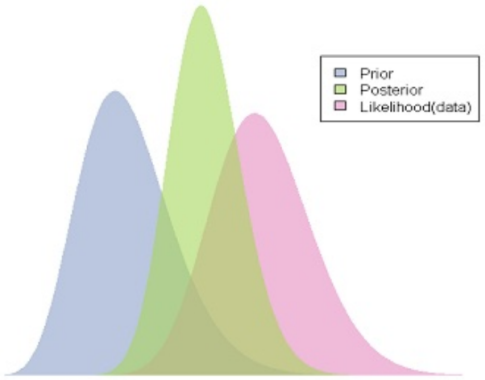


Figure 2.24: Bayes Theorem[Nyc17]

Note that what is being referred to as hidden variables in this case are not truly hidden. GPS measures position and the IMU acceleration. However, these measurements are subject to error. The Kalman filter takes into account the error due to each sensor, and comes up with the most likely outcome. This works similarly to Bayes theorem, where an a priori distribution and a likelihood distribution are merged to give a posterior prediction with less error (less variance). This can be seen in image 2.24.

The Kalman filter can be divided into three parts: a priori prediction, Kalman gain calculation, and update process.

The a priori prediction consists of predicting the next step based on the previous state. Without taking into account GPS and IMU data. It uses the dynamic model to do so. In our case, the dynamic model is represented by basic discretized kinematic equations with constant acceleration:

$$x_t^p = x_{t-1} + v_{t-1} * \Delta t + \frac{1}{2} a_{t-1} * \Delta t^2 \tag{2.46}$$

$$v_t^p = v_{t-1} + a_{t-1} * \Delta t \tag{2.47}$$

$$a_t^p = a_{t-1} \tag{2.48}$$

Where x_t^p,v_t^p, a_t^p are the a priori position, velocity and acceleration estimates at time t and Δt is the update rate. There is one big problem with this model. It assumes that acceleration does not change, which is not true, specially for GoKarting. Later on we will describe how an error term Q_t and the accelerometer data help to mitigate this problem.

Expressing equations 2.46 in matrix form and in two dimensions[FGM11].:

$$X_t^p = A * X_{t-1}^p \tag{2.49}$$

Where X_t^p is known as the state matrix. This contains the estimation of the hidden variables. In our case, we are trying to calculate position, velocity and acceleration. Hence, in two dimensions:

$$X_t^p = \begin{vmatrix} p_x \\ p_y \\ v_x \\ v_y \\ a_x \\ a_y \end{vmatrix} \tag{2.50}$$

A is known as the state transition matrix, in our case:

$$A = \begin{vmatrix} 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 & 0 \\ 0 & 1 & 0 & \Delta t & 0 & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 1 & 0 & \Delta t & 0 \\ 0 & 0 & 0 & 1 & 0 & \Delta t \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix} \tag{2.51}$$

During the a priori prediction, we also predict the a priori process co-variance matrix P_t^p. This matrix is a measure of the error in our prediction. Its value is updated through [FGM11]:

$$P_t = A * P_{t-1} * A^T + Q_t \tag{2.52}$$

Finally, Q_t serves to take into account model imperfections. Since our assumption that acceleration does not change is wrong, we add an error term here. It makes sure that P_t does not converge to 0, as this would render the model useless since data from the IMU and GPS would be discarded. We will consider this a time invariant matrix that will be used as a tuning

parameter. Since it comes from the noise present in X_t^p, which can be modeled as [Shi09]:

$$w_t = \begin{vmatrix} \sigma_{\Delta a_x} \frac{\Delta T^2}{2} \\ \sigma_{\Delta a_y} \frac{\Delta T^2}{2} \\ \sigma_{\Delta a_x} \Delta T \\ \sigma_{\Delta a_y} \Delta T \\ \sigma_{\Delta a_x} \\ \sigma_{\Delta a_y} \end{vmatrix} \tag{2.53}$$

Where $\sigma \Delta a_i$ is the variance of acceleration in axis i. Assuming $\sigma_{\Delta a_x} = \sigma_{\Delta a_y} = cte. = \alpha$, and a co-variance between same axis variables of 1 and of 0 between different axis variables:

$$Q = Q_t = \alpha * \begin{vmatrix} \frac{T^4}{4} & 0 & \frac{T^3}{2} & 0 & \frac{T^2}{2} & 0 \\ 0 & \frac{T^4}{4} & 0 & \frac{T^3}{2} & 0 & \frac{T^2}{2} \\ \frac{T^3}{2} & 0 & T^2 & 0 & T & 0 \\ 0 & \frac{T^3}{2} & 0 & T^2 & 0 & T \\ \frac{T^2}{2} & 0 & T & 0 & 1 & 0 \\ 0 & \frac{T^2}{2} & 0 & T & 0 & 1 \end{vmatrix} \tag{2.54}$$

With α acting as a tuning parameter.

The next step is calculating the Kalman filter Gain K_t. This gain matrix serves to give more weight to the estimated X_t that has less error: either the one coming from the dynamic model or the one coming from the input data. It is calculated as[FGM11]:

$$K_t = P_t^p H^T (H P_t^p H^T + R)^{-1} \tag{2.55}$$

Matrix H is used so that input measurements (GPS and IMU data) can be compared with the X_t. It serves to convert units by meeting the requirement:

$$Y_t = H X_t^p + n \tag{2.56}$$

Where Y_t is the measurements vector. Here:

$$Y_t = \begin{vmatrix} x_{GPS} \\ y_{GPS} \\ a_{x,IMU} \\ a_{y,IMU} \end{vmatrix} \tag{2.57}$$

and n is random noise. Here, the numerator of the Kalman gain represents the error (transformed to input units through H) of the dynamic system estimate. The denominator represents the sum of the dynamic system error and the measurement error. Hence, A higher K represents smaller measurement error.

In our case, two H matrices are defined, one when GPS data is available, and one when it is not. When GPS data is available:

$$p_x = x_{GPS} + n \tag{2.58}$$

$$p_y = y_{GPS} + n \tag{2.59}$$

$$a_x = a_{x,IMU} + n \tag{2.60}$$

$$a_y = a_{y,IMU} + n \tag{2.61}$$

In matrix form:

$$H = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix} \tag{2.62}$$

Without GPS data:

$$a_x = a_{x,IMU} + n \tag{2.63}$$

$$a_y = a_{y,IMU} + n \tag{2.64}$$

In matrix form:

$$H = \begin{vmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{vmatrix} \tag{2.65}$$

Matrix R represents the co-variance matrix of the input data (measures input data error). In our case, when we have GPS data:

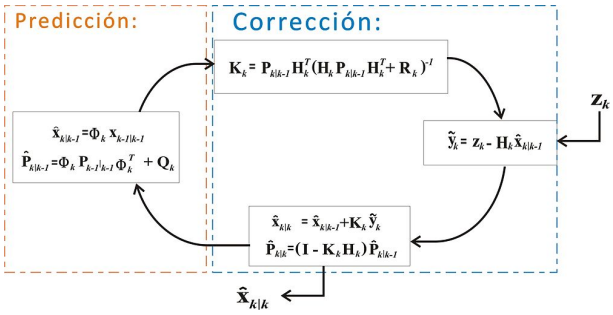


Figure 2.25: Kalman Filter Diagram [19]

$$R = \begin{vmatrix} var(x_{GPS}) & covar(x_{GPS}, y_{GPS}) & 0 & 0 \\ covar(x_{GPS}, y_{GPS}) & var(y_{GPS}) & 0 & 0 \\ 0 & 0 & var(a_{x,IMU}) & covar(a_{x,IMU}, a_{y,IMU}) \\ 0 & 0 & covar(a_{x,IMU}, a_{y,IMU}) & var(a_{y,IMU}) \end{vmatrix} \tag{2.66}$$

and without GPS data:

$$R = \begin{vmatrix} var(a_{x,IMU}) & covar(a_{x,IMU}, a_{y,IMU}) \\ covar(a_{x,IMU}, a_{y,IMU}) & var(a_{y,IMU}) \end{vmatrix} \tag{2.67}$$

These values were previously calculated while calibrating the data.

The last step is the update process. Here, we find the updated values for P_t[FGM11]:

$$P_t = (I - KH)P_t^p \tag{2.68}$$

where I is the identity matrix, and X_t

$$X_t = X_t^p + K(Y_t - HX_t^p) \tag{2.69}$$

Intuitively, a large Kalman gain gives more importance to the input data than to the dynamic model.

Finally, this process is repeated for each new data point. Figure 2.25 shows a diagram of the algorithm. The implementation of the Kalman filter in Python was taken from [Teo18].

The last important note to take into account is that X_t and P_t need to be initialized. In our case, since we assume that the first 1500 data points (30 seconds) are taken in a stationary

position, the initial values are not very important, since they will quickly converge to reasonable values.

Chapter 3

Review of Results and Conclusions

Images of the results of the development process have been shown through this paper. This section will describe the result of applying the Kalman filter.

Two test cases were used to check the effects of the Kalman filter. In one case, the embedded system was left completely still. In another case, it was driven around the road while on a GoKart. Due to the difficulty of more precise test experiments, the results were compared visually. In the future, more precise tests will be carried out.

The first case is standing still. Figure 3.1 shows the result in position given directly by both reading the GPS and estimated by the Kalman filter. After testing various filter parameters, it seems like the accelerometer took over and drifted or the GPS dominated the measurements and brought its imprecision. In the figures mentioned above, the GPS dominates the result. In fact, the accelerometer only seems to add noise to the desired result. A precision of less than 12 meters is achieved, which is in no way nearly enough to accurately track a GoKart.

When testing the filter in dynamic conditions, we can observe the results in figure 3.2. The result is similar to the static test. The accelerometer does not add any new information. In fact, it only adds noise.

This short visual analysis of position estimate is enough to prove that the sensor fusion was not successful. The analysis will continue to focus on what can be done in the future to increase accuracy.

First of all, the source of the noise is very likely coming from the engine vibrations. Using a low pass filter on the accelerometer and gyro readings to get rid of this noise, or even filtering the Kalman filter output could help get rid of these oscillations and get better results.

Secondly, all though an update rate of 50Hz for accelerometer and magnetometer reading is not bad, this can be improved. The major pit fall for update rate has been using FatFs module, which writes to the SD card in blocking mode, and greatly slows down the project. Finding an alternative or restructuring the module so that we can write to the SD card faster could be of great help. Another alternative is using a microprocessor with an SD communications peripheral. In this project, the SD card was used in SPI mode, which only uses one data line. An SD peripheral uses four data lines, making it up to four times faster.

Most importantly, attention to precision. Not enough care was taken to make sure our readings were precise, considering the whole point of this project was to increase accuracy. For example, to log our data, An external 8MHz oscillator was used. It was assumed to be perfectly accurate. The test to make sure that the Euler angles were being estimated correctly in dynamic conditions was purely visual and done in real time. This is not acceptable considering that a small angular difference can result in a large error for large vectors. Lastly, the tests for the Kalman filter were also not good enough. Again, they were visual and had no reference to the true expected values. This is also due to the complexity of carrying out accurate dynamic tests. However, a solution to this could have been to simulate data from the GPS and IMU with some random white noise. This data could have been used to correctly tune the values of our Kalman filter.

3.1 Economic Study

The total price of this project is summed up in the following table:

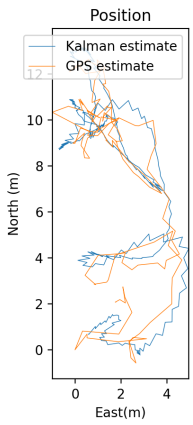


Figure 3.1: GPS and Kalman filter position estimate while still

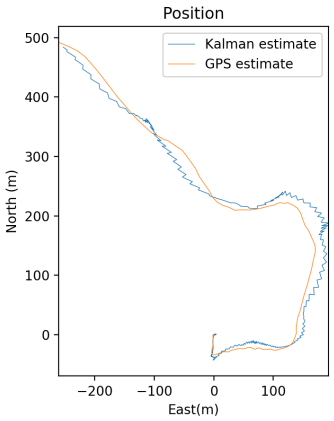


Figure 3.2: GPS and Kalman filter position estimate while moving

Item	Cost ($)
MPU9250	10
NEO6 GPS	12
Microncontroller	2
SD-Card and breakout	5
Proto-board and Cables	2
Total	31

This does not consider the time developing the project. This is merely the cost of the hardware components. This is much cheaper than the professional options described in section 1.3. However, the accuracy is not as desired. It is possible that the expensive GPS modules on these professional devices are justified.

Chapter 4

References

Bibliography

[19] *Filtro de Kalman*. url=https://es.wikipedia.org/wiki/Filtro$_de_Kalman$. 2019. (Last accessed: 07.01.2019).

[Ale19] Enrique Alejo. *All python code for my GoKart data logging project*. url=https://github.com/easox/TF 2019. (Last accessed: 07.18.2019).

[Alp19] Wolfram Alpha. *Cylindrical Equidistant Projection*. url=http://mathworld.wolfram.com/Cylindricall 2019. (Last accessed: 06.02.2019).

[Ber17] Manuel Berger. *How do magnetic inclination and declination differ?* url=https://www.quora.com/Ho do-magnetic-inclination-and-declination-differ. 2017. (Last accessed: 05.17.2019).

[BK16] Robert Bieda and Jaskot Krzysztof. *Determinig of an object orientation in 3D space using direction cosine matrix and non-stationary Kalman filter*. 26th ed. Archives of Control Sciences, 2016.

[Cha19] Elm Chan. *FatFs - Generic FAT Filesystem Module*. url=http://elm-chan.org/fsw/ff/00index_e.html. 2019. (Last accessed: 06.15.2019).

[Cur] Howard D. Curtis. *Orbital Mechanics for Engineering Students*. 3rd ed. Butterworth-Heinemann.

[Dec18] Magnetic Declination. *What is magnetic declination in Madrid, Spain?* url=http://www.magnetic-declination.com/Spain/Madrid/765754.html. 2018. (Last accessed: 06.18.2019).

[Dep12] Dale Depriest. *NMEA data*. url=https://www.gpsinformation.org/dale/nmea.htm#intro. 2012. (Last accessed: 05.02.2019).

[Far12] Ramsey Faragher. *Understanding the Basis of the Kalman Filter Via a Simple and Intuitive Derivation*. 1st ed. IEEE Signal Processing Magazine, 2012.

[FGM11] A. Fakharian, Thomas Gustafsson, and M. Mehrfam. *Adaptive Kalman Filtering Based Navigation: An IMU/GPS Integration Approach*. 1st ed. International Conference on Networking, Sensing and Control, 2011.

[Inv15] Invesense. *MPU-9250 Register Map and Descriptions*. 1.6. 2015.

[Inv16] Invesense. *MPU-9250 Product Specification*. 1.1. 2016.

[Mac11] Leonel Machava. *Rotating 3D Cube using Python and Pygame*. url=http://codentronix.com/2011/05/3d-cube-using-python-and-pygame/. 2011. (Last accessed: 07.11.2019).

[Mad10a] Sebastian Madgwick. *An efficient orientation filter for inertial and inertial/magnetic sensor arrays*. 1st ed. 2010.

[Mad10b] Sebastian Madgwick. *An efficient orientation filter for inertial and inertial/magnetic sensor arrays*. 1st ed. 2010.

[Mad11] Sebastian Madgwick. *Quaternions*. 1st ed. 2011.

[MIC14] Alexandra Malyugina, Konstantin Igudesman, and Dmitry Chickrin. *Least-Squares Fitting of a Three-Dimensional Ellipsoid to Noisy Data*. 8th ed. Kazan Federal University, 2014.

[Nyc17] Douglas Nychka. *The Likelihood, the prior and Bayes Theorem*. url=https://www.image.ucar.edu/pul 2017. (Last accessed: 06.20.2019).

[Ozy15] Talat Ozyagcilar. *Calibrating an eCompass in the Presence of Hard- and Soft-Iron Interference*. 4th ed. Freescale Semiconductor, 2015.

[Pen15] Wade Penson. *sd-spi-communications-library*. url=https://github.com/wpenson/sd-spi-communications-library. 2015. (Last accessed: 06.13.2019).

[Rac19a] RaceCloud. *Innovative 3-Component Live Telemetry Solution For Go Karting*. 2019. (Last accessed: 06.02.2019).

[Rac19b] RaceTime. *GPS Stopwatch and Telemetry for Android*. 2019. (Last accessed: 06.27.2019).

[SE18] Grant Sanderson and Ben Eater. *Visualizing quaternions*. url=https://eater.net/quaternions. 2018. (Last accessed: 07.02.2019).

[Sem18] Mark Semple. *Python adaptation of Yury Petrov's MATLAB ellipsoid_fit function*. url=https://github.com/marksemple/pyEllipsoid_Fit. 2018.

[Shi09] N Shimkin. *Estimation and Identification in Dynamical Systems*. 1st ed. 2009.

[STM15] STMicroelectronics. *Medium-density performance line ARM®-based 32-bit MCU with 64 or 128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 com. interfaces*. 17th ed. STMicroelectronics, 2015.

[STM17] STMicroelectronics. *Description of STM32F1 HAL and Low-layer drivers*. 2nd ed. STMicroelectronics, 2017.

[Tec] Kingston Technology. *microSDHC memory card Flash Storage Media*. 1st ed. Kingston.

[Teo18] James Teow. *Understanding Kalman Filters with Python*. url=https://medium.com/@jaems33/unders
kalman-filters-with-python-2310e87b8f48. 2018. (Last accessed: 06.03.2019).

[Ubl11] Ublox. *NEO-6 u-blox 6 GPS Modules*. 1st ed. Ublox, 2011.

[Vec08] Vectornav. *Magnetometer*. url=https://www.vectornav.com/support/library/magnetometer.
2008. (Last accessed: 06.19.2019).

Chapter 5

Appendix

5.A Serial

```c
#include "serial.h"


void my_putchar(uint16_t c)
{
  uint8_t buffer[2];
  buffer[0]=c;
  buffer[1]=(c>>8);
  HAL_UART_Transmit(&huart2, buffer, 2, 1);
}
void putstring(char *str)
{
    while (*str)
    {
        my_putchar(*str);
        str++;
    }
}

void vprint(const char *fmt, va_list argp)
{
    char string[200];
    if (0 < vsprintf(string, fmt, argp)) // build string
    {
        putstring(string);
    }
```

```
}

void my_printf(const char *fmt, ...) // custom printf() function outputs throuhg pin A2 (uart2)
{
    va_list argp;
    va_start(argp, fmt);
    vprint(fmt, argp);
    va_end(argp);
}
```

5.B MPU9250

```

#include "mpu9250.h"
#include "serial.h"

//TODO: Calibrate accel, read temp, FIFO and motion tracker on board.

#define MPU_I2C_ADDRESS 0b11010000
#define MAG_I2C_ADDRESS 0x18

extern uint32_t uwTick;

void writeReg(uint8_t address,uint8_t data){

  uint8_t recieved[2];
  uint8_t sent[2];
  sent[1]=address;
  sent[0]=data;
  sel(2);
  HAL_SPI_TransmitReceive(&hspi2,sent,recieved,1,100);
  deselect(2);

}

uint8_t readReg(uint8_t address){
  uint8_t data;

  uint8_t recieved[2]={0,0};
  uint8_t sent[2];
  sent[1]=(1<<7)|address;
  sent[0]=0x00;
  sel(2);
```

```
  HAL_SPI_TransmitReceive(&hspi2,sent,recieved,1,10);
  deselect(2);
  data= recieved[0];


  return data;

}

uint8_t read_mag(uint8_t address){
  writeReg(MPUREG_I2C_SLV0_REG,address);
  writeReg(MPUREG_I2C_SLV0_CTRL,0x81);
  volatile int i=0;
  for(i=0;i<4000;i++){

  }
   uint8_t value=readReg(MPUREG_EXT_SENS_DATA_00);
   return value;
}

void write_mag ( uint8_t address,uint8_t data){
  writeReg(MPUREG_I2C_SLV0_ADDR,0x0C);
  HAL_Delay(1);
  writeReg(MPUREG_I2C_SLV0_REG,address);
  HAL_Delay(1);
  writeReg(MPUREG_I2C_SLV0_DO,data);
  HAL_Delay(1);
  writeReg(MPUREG_I2C_SLV0_CTRL,0x81);
  HAL_Delay(1);
  writeReg(MPUREG_I2C_SLV0_ADDR,0x8C);
  HAL_Delay(1);
}



void mag_init_spi(void){
  //allow imu to communicate with  salves through i2c
  writeReg(MPUREG_USER_CTRL,0b00110000);
  writeReg(MPUREG_I2C_MST_CTRL,0b01010000);
  writeReg(MPUREG_INT_PIN_CFG,0x22);

  //set magnetometer as slave 0 to read
  writeReg(MPUREG_I2C_SLV0_ADDR,0x0C);
  HAL_Delay(10);
  //write to CNTL2 to reset device
```

```
77    write_mag(AK8963_CNTL2,0x01);
78    HAL_Delay(10);
79    write_mag(AK8963_CNTL1,0x16);
80    HAL_Delay(10);
81  }
82  void mag_init_i2c(void){
83    //allow imu to communicate with  salves through i2c
84    writeReg(MPUREG_USER_CTRL,0b00100000);
85    writeReg(MPUREG_I2C_MST_CTRL,0b01010000);
86    writeReg(MPUREG_INT_PIN_CFG,0x22);
87    //set magnetometer as slave 0 to read
88    writeReg(MPUREG_I2C_SLV0_ADDR,0x0C);
89    HAL_Delay(10);
90    //write to CNTL2 to reset device
91    write_mag(AK8963_CNTL2,0x01);
92    HAL_Delay(10);
93    write_mag(AK8963_CNTL1,0x12);
94    HAL_Delay(10);
95  }
96
97  void mpu9250_init(void){
98     writeReg( MPUREG_PWR_MGMT_1,0x80); // Reset Device
99     writeReg( MPUREG_PWR_MGMT_1,0x01);     // Clock Source
100    writeReg( MPUREG_PWR_MGMT_2,0x00);     // Enable Acc & Gyro
101    writeReg( MPUREG_CONFIG,0x00); // Use DLPF set Gyroscope bandwidth
       184Hz, temperature bandwidth 188Hz
102    writeReg( MPUREG_GYRO_CONFIG,0x18);    // +-2000dps
103    writeReg( MPUREG_ACCEL_CONFIG,0x08);   // +-4G
104    writeReg( MPUREG_ACCEL_CONFIG_2,0x09); // Set Acc Data Rates, Enable
       Acc LPF , Bandwidth 184Hz
105    writeReg( MPUREG_INT_PIN_CFG,0x22);  //
106    set_acc_scale(BITS_FS_4G);
107    set_gyro_scale(BITS_FS_500DPS);
108    HAL_Delay(100);
109    return;
110 }
111
112
113 void sel(int i){
114    if (i==2){
115       HAL_GPIO_WritePin(GPIOB,GPIO_PIN_12,GPIO_PIN_RESET);
116    }
117
118 }
119 void deselect(int i){
```

```
120     if(i==2){
121         HAL_GPIO_WritePin(GPIOB,GPIO_PIN_12,GPIO_PIN_SET);
122     }
123 }
124 
125 
126 
127 /*
    ------------------------------------------------------------------------------

128                                 ACCELEROMETER SCALE
129 usage: call this function at startup, after initialization, to set the
    right range for the
130 accelerometers. Suitable ranges are:
131 BITS_FS_2G
132 BITS_FS_4G
133 BITS_FS_8G
134 BITS_FS_16G
135 returns the range multiplier
136 ------------------------------------------------------------------------------
    */
137 float set_acc_scale( uint16_t scale){
138 
139 
140     writeReg(MPUREG_ACCEL_CONFIG, scale);
141 
142     switch(scale){
143       case BITS_FS_2G: return MPU9250A_2g;
144             case BITS_FS_4G: return MPU9250A_4g;
145             case BITS_FS_8G: return MPU9250A_8g;
146             case BITS_FS_16G: return MPU9250A_16g;
147             default: return -1;
148         }
149 
150 }
151 
152 
153 
154 /*
    ------------------------------------------------------------------------------

155                                 GYROSCOPE SCALE
156 usage: call this function at startup, after initialization, to set the
    right range for the
157 gyroscopes. Suitable ranges are:
```

```
158 BITS_FS_250DPS
159 BITS_FS_500DPS
160 BITS_FS_1000DPS
161 BITS_FS_2000DPS
162 returns the range multiplier
163 ------------------------------------------------------------------------------
    */
164 float set_gyro_scale(uint16_t scale){
165
166         uint16_t write_val=scale;
167    writeReg(MPUREG_GYRO_CONFIG,write_val );
168
169         switch(scale){
170             case BITS_FS_250DPS: return MPU9250G_250dps;
171             case BITS_FS_500DPS: return MPU9250G_500dps;
172             case BITS_FS_1000DPS: return MPU9250G_1000dps;
173             case BITS_FS_2000DPS: return MPU9250G_2000dps;
174             default: return -1;
175         }
176
177 }
178
179
180 void set_accel_filter(uint8_t state, uint8_t filter){
181
182  if(state==DLPF_OFF){
183    writeReg(MPUREG_ACCEL_CONFIG_2, 0b1000);
184  }
185  if(state==DLPF_ON && filter<8){
186      writeReg(MPUREG_ACCEL_CONFIG_2, filter);
187  }
188
189 }
190
191 void set_gyrotemp_filter(uint8_t state, uint8_t filter){
192  uint16_t gyro_scale= readReg(MPUREG_GYRO_CONFIG);
193  if(state==DLPF_OFF){
194
195    writeReg(MPUREG_GYRO_CONFIG, (gyro_scale & 0b11111100 )|01);
196  }
197  if(state==DLPF_ON && filter<8){
198      writeReg(MPUREG_GYRO_CONFIG, (gyro_scale & 0b11111100 )|10);
199      uint16_t config= readReg(MPUREG_CONFIG);
200      writeReg(MPUREG_CONFIG, (config & 0b11111000 )|filter);
201  }
```

```
 if(state==DLPF_ONON && filter<8){
     writeReg(MPUREG_GYRO_CONFIG, (gyro_scale & 0b11111100 ));
     uint16_t config= readReg(MPUREG_CONFIG);
     writeReg(MPUREG_CONFIG, (config & 0b11111000 )|filter);
 }


}


/*
    ------------------------------------------------------------------------------

                                WHO AM I?
usage: call this function to know if SPI is working correctly.
------------------------------------------------------------------------------
    */
int whoami(void){
  int response=0;
  response=readReg(MPUREG_WHOAMI);

  if(response == 0x71){
    return 1;
  }

  return 0;
}

//check if magnetometer works
int whoami_mag(void){

  int response=0;
  response=read_mag(AK8963_WIA);

  if(response == 0x48){
    return 1;
  }

  return 0;
}


int16_t read_X_accel (void){
  uint16_t aux=readReg(MPUREG_ACCEL_XOUT_H);
  uint16_t value=readReg(MPUREG_ACCEL_XOUT_L);
```

```
244   int16_t xaccel=(aux<<8)|(value);
245   return xaccel;
246 
247 }
248 
249 
250 
251 int16_t read_Y_accel (void){
252   uint16_t aux=readReg(MPUREG_ACCEL_YOUT_H);
253   uint16_t value=readReg(MPUREG_ACCEL_YOUT_L);
254   int16_t yaccel=(aux<<8)|(value);
255   return yaccel;
256 
257 }
258 
259 int16_t read_Z_accel (void){
260   uint16_t aux=readReg(MPUREG_ACCEL_ZOUT_H);
261   uint16_t value=readReg(MPUREG_ACCEL_ZOUT_L);
262   int16_t zaccel=(aux<<8)|(value);
263   return zaccel;
264 
265 }
266 
267 int16_t read_X_gyro(void){
268   uint16_t aux=readReg(MPUREG_GYRO_XOUT_H);
269   uint16_t value=readReg(MPUREG_GYRO_XOUT_L);
270   int16_t xgyro=(aux<<8)|(value);
271   return xgyro;
272 
273 }
274 
275 int16_t read_Y_gyro(void){
276   uint16_t aux=readReg(MPUREG_GYRO_YOUT_H);
277   uint16_t value=readReg(MPUREG_GYRO_YOUT_L);
278   int16_t ygyro=(aux<<8)|(value);
279   return ygyro;
280 
281 }
282 
283 int16_t read_Z_gyro(void){
284   uint16_t aux=readReg(MPUREG_GYRO_ZOUT_H);
285   uint16_t value=readReg(MPUREG_GYRO_ZOUT_L);
286   int16_t zgyro=(aux<<8)|(value);
287   return zgyro;
288 
```

```
}

int16_t read_X_mag(void){

  uint16_t low=read_mag(AK8963_HXL);
  read_mag(AK8963_ST2);
  uint16_t high=read_mag(AK8963_HXH);
  read_mag(AK8963_ST2);
  int16_t value=(high<<8)|(low);
  return value;
}

int16_t read_Y_mag(void){

  uint16_t low=read_mag(AK8963_HYL);
  read_mag(AK8963_ST2);
  uint16_t high=read_mag(AK8963_HYH);
  read_mag(AK8963_ST2);
  int16_t value=(high<<8)|(low);
  return value;
}
int16_t read_Z_mag(void){

  uint16_t low=read_mag(AK8963_HZL);
  read_mag(AK8963_ST2);
  uint16_t high=read_mag(AK8963_HZH);
  read_mag(AK8963_ST2);
  int16_t value=(high<<8)|(low);
  return value;
}



void accel_x_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_XA_OFFSET_H,high);
  writeReg(MPUREG_XA_OFFSET_L,low);
}
void accel_y_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_YA_OFFSET_H,high);
  writeReg(MPUREG_YA_OFFSET_L,low);

```

```
}
void accel_z_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_ZA_OFFSET_H,high);
  writeReg(MPUREG_ZA_OFFSET_L,low);

}
void gyro_x_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_XG_OFFSET_H,high);
  writeReg(MPUREG_XG_OFFSET_L,low);

}
void gyro_y_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_YG_OFFSET_H,high);
  writeReg(MPUREG_YG_OFFSET_L,low);

}
void gyro_z_offset( uint16_t offset){
  uint8_t high= offset>>8;
  uint8_t low= offset;
  writeReg(MPUREG_ZG_OFFSET_H,high);
  writeReg(MPUREG_ZG_OFFSET_L,low);
}


void calib_gyro(void){
  int gx,gy,gz;
  uint16_t status;
   status=readReg(MPUREG_GYRO_CONFIG);
   set_gyro_scale(BITS_FS_1000DPS);
   gx= read_X_gyro();
   gy= read_Y_gyro();
   gz= read_Z_gyro();
   gyro_x_offset(-gx);
   gyro_y_offset(-gy);
   gyro_z_offset(-gz);
   writeReg(MPUREG_GYRO_CONFIG,status);
}


```

```
379
380
381
382 void delay(int i){
383   while(i-->0){
384     asm("nop");
385   }
386 }
387
388
389 int mpu_init(void){
390   int attempt=0;
391   while(1){
392     mpu9250_init();
393     mag_init_spi();
394
395
396
397     set_acc_scale(BITS_FS_4G);
398     set_gyro_scale(BITS_FS_500DPS);
399     set_accel_filter(DLPF_OFF,0);
400     set_gyrotemp_filter(DLPF_OFF,0);
401
402     if(whoami_mag()&& whoami()){
403       break;
404     }
405
406     HAL_SPI_DeInit(&hspi2);
407     HAL_Delay(1);
408     HAL_SPI_Init(&hspi2);
409     attempt++;
410     if(attempt==100){
411       return 1;
412     }
413   }
414   return 0;
415 }
416
417
418 void euler(void){
419
420   float ax= read_X_accel()*MPU9250A_4g;//*a_factor;
421   float ay= read_Y_accel()*MPU9250A_4g;//*a_factor;
422   float az= read_Z_accel()*MPU9250A_4g;//*a_factor;
423
```

```
424   float gx= read_X_gyro()*MPU9250G_500dps;//*g_factor;
425   float gy= read_Y_gyro()*MPU9250G_500dps;//*g_factor;
426   float gz= read_Z_gyro()*MPU9250G_500dps;//*g_factor;
427
428   float mx=read_X_mag();//*Magnetometer_Sensitivity_Scale_Factor;
429   float my=read_Y_mag();//*Magnetometer_Sensitivity_Scale_Factor;
430   float mz=read_Z_mag();//*Magnetometer_Sensitivity_Scale_Factor;
431
432   my_printf("%d:%d,%d,%d\r\n",uwTick,mx,my,mz);
433
434
435
436   update(gx,gy,gz,ax,ay,az,mx,my,mz);
437   computeAngles();
438 }
```

5.C diskIO

This is a wrapper on the SD Device Manager created by [Pen15] so that it can be integrated with the FatFs module.

```
1  /*
       -----------------------------------------------------------------------
       */
2  /* Low level disk I/O module skeleton for FatFs     (C)ChaN, 2014
        */
3
4  /* Includes
       ------------------------------------------------------------------*/
5  #include "diskio.h"
6  #include "header.h"
7  #include "sd_spi.h"
8
9
10 DSTATUS status;
11
12
13 DSTATUS disk_initialize (
14   BYTE pdrv       /* Physical drive nmuber to identify the drive */
15 )
16 {
17   if(sd_spi_init(GPIO_PIN_4)==SD_ERR_OK){
18     status=0;
19
```

```
  }else{
    status=STA_NOINIT;
  }
  return status;
}

/**
  * @brief  Reads Sector(s)
  * @param  pdrv: Physical drive number (0..)
  * @param  *buff: Data buffer to store read data
  * @param  sector: Sector address (LBA)
  * @param  count: Number of sectors to read (1..128)
  * @retval DRESULT: Operation result
  */
DRESULT disk_read (
  BYTE pdrv,    /* Physical drive nmuber to identify the drive */
  BYTE *buff,   /* Data buffer to store read data */
  DWORD sector,         /* Sector address in LBA */
  UINT count    /* Number of sectors to read */
)
{
  DRESULT res;

  if(sd_spi_read(sector, buff,512*count,0)==SD_ERR_OK){
    res=RES_OK;
  }else{
    res=RES_ERROR;
  }
  return res;
}

/**
  * @brief  Writes Sector(s)
  * @param  pdrv: Physical drive number (0..)
  * @param  *buff: Data to be written
  * @param  sector: Sector address (LBA)
  * @param  count: Number of sectors to write (1..128)
  * @retval DRESULT: Operation result
  */
#if _USE_WRITE == 1
DRESULT disk_write (
  BYTE pdrv,    /* Physical drive nmuber to identify the drive */
  const BYTE *buff, /* Data to be written */
  DWORD sector,   /* Sector address in LBA */
  UINT count          /* Number of sectors to write */
```

```
)
{
    sd_spi_write_continuous_start(sector, count);
    uint32_t i;
    for(i=0;i<count;i++){
      sd_spi_write_continuous(buff+i*512, 512, 0);
      sd_spi_write_continuous_next(); // Advance to the next block in the sequence.

    }
    sd_spi_write_continuous_stop(); // Stop the writing process. This implicitly flushes the buffer to the card.
    return RES_OK;
}

#endif /* _USE_WRITE == 1 */
```

5.D Timer Interrupt: Output data logging

```
void TIM2_IRQHandler(void)
{

  //Handle interrupt
  HAL_TIM_IRQHandler(&htim2);

  //send start sequence
  sd_buffer_append(0x59);
  sd_buffer_append(0x59);



   //get current time

  sd_buffer_append(uwTick);
  sd_buffer_append(uwTick>>8);
  sd_buffer_append(uwTick>>16);
  sd_buffer_append(uwTick>>24);




//Get accel and gyro and send it

```

```
25
26  sd_buffer_append(readReg(MPUREG_ACCEL_XOUT_L));
27  sd_buffer_append(readReg(MPUREG_ACCEL_XOUT_H));
28  sd_buffer_append(readReg(MPUREG_ACCEL_YOUT_L));
29  sd_buffer_append(readReg(MPUREG_ACCEL_YOUT_H));
30  sd_buffer_append(readReg(MPUREG_ACCEL_ZOUT_L));
31  sd_buffer_append(readReg(MPUREG_ACCEL_ZOUT_H));
32
33  sd_buffer_append(readReg(MPUREG_GYRO_XOUT_L));
34  sd_buffer_append(readReg(MPUREG_GYRO_XOUT_H));
35  sd_buffer_append(readReg(MPUREG_GYRO_YOUT_L));
36  sd_buffer_append(readReg(MPUREG_GYRO_YOUT_H));
37  sd_buffer_append(readReg(MPUREG_GYRO_ZOUT_L));
38  sd_buffer_append(readReg(MPUREG_GYRO_ZOUT_H));
39
40
41
42  if(send_GPS==0){    //send an end bit=0
43
44
45
46  }else{  //send an end bit to signal to send more data
47  send_GPS=0;
48    sd_buffer_append(0x71);
49
50    //Get and send mag data
51    sd_buffer_append(read_mag(AK8963_HXL));
52    read_mag(AK8963_ST2);
53    sd_buffer_append(read_mag(AK8963_HXH));
54    read_mag(AK8963_ST2);
55    sd_buffer_append(read_mag(AK8963_HYL));
56    read_mag(AK8963_ST2);
57    sd_buffer_append(read_mag(AK8963_HYH));
58    read_mag(AK8963_ST2);
59    sd_buffer_append(read_mag(AK8963_HZL));
60    read_mag(AK8963_ST2);
61    sd_buffer_append(read_mag(AK8963_HZH));
62    read_mag(AK8963_ST2);
63
64    //send GPS data
65    sd_buffer_send();
66    sd_gps_send();
67  }
68  //Get end time of whole proccess
69  sd_buffer_append(uwTick);
```

```
  sd_buffer_append(uwTick>>8);
  sd_buffer_append(uwTick>>16);
  sd_buffer_append(uwTick>>24);

}
```

5.E Read data files from SD card

```
def read_mag_file(file="magcalib.txt"):
  data = np.empty((0,3), float) #x,y,z mag values
  in_file=open(file,"rb")
  line=in_file.read(2)
  if line==b'YY':
    while True:
      line=in_file.read(6)
      if check_eof(line,6):
        in_file.close()
        break

      my,mx,mz_neg=np.asarray(struct.unpack('<hhh',line))
      mag=[mx,my,-mz_neg]
      data=np.append(data,[mag],axis=0)
    return data

#reads out file into 2 arrays, one with IMU, one with GPS
def read_data_file(file="out.txt"):
    data1 = np.empty((0,8), float) # timestamp, accelx,y,z,gyrox,y,z
    data2 = np.empty((0,8),float) #timestamp,gpstime,lon,lat,vel,
   track_angle,mag_var,magx,y,z
    in_file = open(file, "rb") # opening for [r]eading as [b]inary
    while True:
      if check_start(in_file):
        break

      line=in_file.read(17)
      if check_eof(line,17):
        in_file.close()
        break


      timestamp,ax,ay,az,gx,gy,gz,end_byte=struct.unpack('<IhhhhhhB',
   line)
      temp_data=np.array([[timestamp,ax*G4ACCEL,ay*G4ACCEL,az*G4ACCEL,gx
   *DPS500,gy*DPS500,gz*DPS500,end_byte]])
```

```
34
35
36
37      data1=np.append(data1,temp_data,0)
38      if end_byte==0x71:
39          line=in_file.read(6)
40          if check_eof(line,6):
41            break
42          my,mx,mz_neg=np.asarray(struct.unpack('<hhh',line))
43          mag=[mx,my,-mz_neg]
44          gps_line=in_file.read(GPS_BUFFER_SIZE)
45          if check_eof(gps_line,GPS_BUFFER_SIZE):
46            in_file.close()
47            break
48          gps_data=gps_line.decode('utf-8','ignore')
49          gpstime,lon,lat,vel=gps_unpack(gps_data.split('\r')[0])
50          temp_data=np.array([[timestamp,gpstime,lon,lat,vel,mag[0],mag
    [1],mag[2]]])
51
52          data2=np.append(data2,temp_data,0)
53
54
55
56      line=in_file.read(4)
57      if check_eof(line,4):
58        in_file.close()
59        break
60
61      end_time=struct.unpack('<I',line)
62
63
64
65    return data1,data2
66
67
68 #checks that there are still bytes to be read, if not, ends the search
69 def check_eof(string,read_b):
70   if len(string)<read_b:
71     print("EOF file reached correctly")
72     return 1
73
74   else:
75     return 0
76
77
```

```
#checks start sequence YY
def check_start(in_file):
    while True:
            a=in_file.read(1)
            if check_eof(a,1):
              in_file.close()
              return 1

            if a==b'Y':
                a=in_file.read(1)
                if check_eof(a,1):
                  in_file.close()
                  return 1
                if a==b'Y':
                    return 0
```

5.F Parse GPS data

```
def gps_unpack(line):
  if line:
    msg=pynmea2.parse(line)
    if msg.is_valid:
      lat=msg.latitude
      lon=msg.longitude
      speed=float(msg.data[6])*1.852
      time=float(msg.data[0])

      return time,lon,lat,speed

    else:
      print("GPRMC data not valid")

      return 200,300,400,500

  else:
    print("Nothing to read")
    return 200,300,400,500
```

5.G Attitude

```
#Initializes Madgwick object
mad_filter=madgwickahrs.MadgwickAHRS(1/50,beta=.13,alpha=.04)
```

```
3
4  #reads data from 'out.txt'
5  data1,data2=read_data_file(file='out.txt')
6
7  #calibrate gyro
8  data1=calib_gyro(data1)
9
10 #calibrates magnetometer
11 data2=calibration_mag(data2)
12
13 #converts lon and lat to meters
14 data2=coord_2_meters(data2)
15
16 #centers GPS data around (0,0)
17 data2=center_meters(data2)
18
19 #calculates rotation matrix to return to
20 R=data_2_NWU(data1,data2)
21
22 #refers all accelerometer data to NWU axes
23 for a in range(0,len(data1)):
24   data1[a,1:4]=np.matmul(R,np.transpose(data1[a,1:4]))
25   data1[a,4:7]=np.matmul(R,np.transpose(data1[a,4:7]))
26
27 #refers all magnetometer data to NWU axes (not necessary, but useful for
       checking)
28 for a in range(0,len(data2)):
29   data2[a,5:8]=np.matmul(R,np.transpose(data2[a,5:8]))
30
31 #init np array with orientation values
32 orientation=np.empty((0,3),float)
33
34 #init array with NWU refered acceleration (even during dynamic
     conditions)
35 a_NWU_aux=np.empty((0,3),float)
36 data2_index=0
37
38 #for all the data points collected
39 for a in range(0, len(data1)):
40
41
42   if data1[a,7]==0: #if we do not have magnetometer data
43     #calculate euler angles with Madgwick filter
44     theta=updateEuler(data1[a,1:4],data1[a,4:7],data1[a,7],[0,0,0],
     mad_filter)
```

```
    #Euler angles to inverse rotation matrix
    R_tot=allign_axis.eulerAnglesToRotationMatrix([theta[2],theta[1],theta[0]])
    #apply rotation to acceleration to refer to NWU axes
    a_NWU=R_tot@(data1[a,1:4])

  else: #if we have magnetometer data
    #same as before
    theta=updateEuler(data1[a,1:4],data1[a,4:7],data1[a,7],data2[data2_index,5:8],mad_filter)
    R_tot=allign_axis.eulerAnglesToRotationMatrix([theta[2],theta[1],theta[0]])
    a_NWU=R_tot@(data1[a,1:4])
    data2_index+=1

#appends to the np array
  a_NWU_aux=np.append(a_NWU_aux,np.array([a_NWU]),0)
```

5.H Cube rotation in Pygame

```
def updateEuler(): #recives data packet and calculates new euler angles

  data=reciever.read_packet()
  gx=data[3]-offset[0]
  gy=data[4]-offset[1]
  gz=data[5]-offset[2]
  if len(data)==6:

    mad_filter.update_imu([gx*math.pi/180,gy*math.pi/180,gz*math.pi/180],[data[0],data[1],data[2]])
  else:
    mx,my,mz=correct_magnetometer(np.array([[data[6]],[data[7]],[data[8]]]))
    mad_filter.update([gx*math.pi/180,gy*math.pi/180,gz*math.pi/180],[data[0],data[1],data[2]],[mx,my,mz])
    print(gx*math.pi/180,gy*math.pi/180,gz*math.pi/180)
    print(data[0],data[1],data[2])
    print(mx,my,mz)

  roll,pitch,yaw=quat_to_degree(mad_filter.quaternion)
  print(time.time())
  print()
  return roll,pitch,yaw

```

```
def quat_to_degree(q):
  return Quaternion_naive.getEulerAngles(q)
def correct_magnetometer(m):
  global cx,cy,cz

  return m[0]-cx, m[1]-cy,m[2]-cz

def calibration_gyro():
  calib_reciver=uart_recieve.uart_reciever()
  gx=0
  gy=0
  gz=0
  for i in range(0,100):
    data=calib_reciver.read_packet()
    gx+=data[3]
    gy+=data[4]
    gz+=data[5]

  return np.array([[gx/100],[gy/100],[gz/100]])

def calibration_mag():
  data = sd_recieve.read_mag_file("magcalib.txt")
  cx=0
  cy=0
  cz=0
  for item in data:
    cx+=item[0]
    cy+=item[1]
    cz+=item[2]
  cx=cx/len(data)
  cy=cy/len(data)
  cz=cz/len(data)

  return cx,cy,cz




if __name__ == "__main__":
  global cx,cy,cz

  global offset
  #calibrate gyro and mag
  offset=calibration_gyro()
  cx,cy,cz=calibration_mag()
```

```
67
68  #initialize madg filter
69  mad_filter=madgwickahrs.MadgwickAHRS(1/50,beta=.1)
70
71  #intialize serial connection
72  reciever=uart_recieve.uart_reciever()
73
74  #simulate cube (borrowed code)
75  cube=Simulation()
76  cube.run(0,0,0)
```