Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Caixa Preta para Veículos Automotivos

Gabriela da Silva Lopes

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Ricardo Zelenovsky

Brasília
2018

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Engenharia da Computação

Coordenador: Prof. Dr. Ricardo Pezzuol Jacobi

Banca examinadora composta por:

Prof. Dr. Ricardo Zelenovsky (Orientador) — ENE/UnB
Prof. Dr. Alexandre Romariz — ENE/UnB
Mestre Vinícius de Oliveira Lima — IC/PCDF

CIP — Catalogação Internacional na Publicação

Lopes, Gabriela da Silva.

Caixa Preta para Veículos Automotivos / Gabriela da Silva Lopes. Brasília : UnB, 2018.
117 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2018.

1. IMU, 2. perícia, 3. acidentes de trânsito, 4. simulação 3D, 5. filtro de orientação

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Caixa Preta para Veículos Automotivos

Gabriela da Silva Lopes

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Ricardo Zelenovsky (Orientador)
ENE/UnB

Prof. Dr. Alexandre Romariz
ENE/UnB

Mestre Vinícius de Oliveira Lima
IC/PCDF

Prof. Dr. Ricardo Pezzuol Jacobi
Coordenador do Curso de Engenharia da Computação

Brasília, 05 de Julho de 2018

Dedicatória

Dedico este trabalho aos meus pais, Marilene e Francisco, e ao meu irmão, Guilherme.

Agradecimentos

Agradeço aos meus pais, Marilene e Francisco, por todo o apoio, amor, confiança, por sempre cuidarem tão bem de mim e me motivarem em todas as etapas da minha vida. E ao meu irmão, Guilherme, por todo o companheirismo, amizade e ajuda com o que fosse preciso.

Agradeço à minha família pelo apoio e carinho. E aos meus amigos por todo o incentivo, ensinamentos e por proporcionarem tantos momentos incríveis durante estes anos.

Agradeço também ao meu orientador, Ricardo Zelenovsky, pela dedicação e apoio na realização deste projeto. E aos meus professores por todo o conhecimento transmitido.

Resumo

A detecção e o acompanhamento de movimentos desempenha um papel muito importante em uma grande variedade de aplicações. Um sensor de medição inercial (IMU) é um dispositivo, composto de sensores como acelerômetros, giroscópios e magnetômetros, utilizado para esta finalidade. Neste trabalho, um IMU e um receptor GPS foram utilizados em um equipamento desenvolvido para obter informações de um automóvel no momento de um acidente. Os dados obtidos são armazenados em uma memória não volátil, o que permite uma análise após o evento e serve como ferramenta para a perícia técnica do acidente. A análise dos dados obtidos é realizada utilizando gráficos e também um *software*, elaborado com a plataforma de desenvolvimento de jogos Unity 3D, capaz de realizar uma simulação tridimensional para reconstruir a dinâmica de um acidente de trânsito. A fusão dos dados obtidos com os sensores foi realizada utilizado um filtro, conhecido como filtro de Madgwick, que utiliza quaternions para representar orientações. O objetivo é realizar vários ensaios com o *hardware* e o *software* desenvolvidos para verificar os dados obtidos.

Palavras-chave: IMU, perícia, acidentes de trânsito, simulação 3D, filtro de orientação

Abstract

Motion detection and tracking play a critical role in a wide range of fields. An inertial measurement unit (IMU) is a device, made up of sensors such as accelerometers, gyroscopes and magnetometers, used for this purpose. In this project, an IMU and a GPS receiver were used in a device designed to obtain car information at the time of an accident. The data obtained are stored in a non-volatile memory, which allows a post-event analysis and it serves as a tool for the forensic analysis of the accident. The analysis of the obtained data is performed using graphs and also a software developed with the Unity 3D game development platform to perform a three-dimensional simulation to reconstruct the dynamics of the accident. The data fusion was performed using a filter, known as a Madgwick filter, which uses quaternions to represent orientations. The goal is to perform several tests with the hardware and software developed to verify the data obtained.

Keywords: IMU, forensic analysis, traffic accidents, 3D simulation, orientation filter

Sumário

Lista de Figuras

Lista de Abreviaturas e Siglas

API *Application Programming Interface.*

CS *Chip Select.*

DPVAT Danos Pessoais Causados por Veículos Automotores de Via Terrestre.

EEPROM *Eletrically Erasable Programmable Read Only Memory.*

GPS *Global Positioning System.*

I2C *Inter-IC Bus.*

IDE *Integrated Development Environment.*

IMU *Inertial Measurement Units.*

LCD *Liquid Crystal Display.*

LED *Light Emitting Diode.*

MEMS *Micro Electro Mechanical System.*

MPU *Motion Processing Unit.*

SCK *Serial Clock Pin.*

SCL *Serial Clock.*

SDA *Serial Data.*

SI *Serial Input.*

SO *Serial Output.*

SPI *Serial Peripheral Interface.*

SRAM *Static Random Access Memory.*

USB *Universal Serial Bus.*

UTC *Universal Time Coordinated.*

Capítulo 1

Introdução

Dados do Sistema de Informação sobre Mortalidade [7] do Ministério da Saúde, e do Seguro DPVAT [8], mostram as estatísticas nacionais de acidentes de trânsito [9]. No Brasil, aproximadamente, 45 mil pessoas morrem vítimas de acidentes de trânsito a cada ano [10]. Em muitos casos, é possível que ocorram processos judiciais, e nessas situações um laudo pericial desempenha um papel muito importante.

Acidentes são acontecimentos possivelmente evitáveis causados por uma somatória de fatores relacionados ao ser humano, ao ambiente e ao veículo [11]. Logo, a reconstrução de um acidente investiga todos esses fatores para determinar como cada um deles contribuiu para causar ou agravar um acidente.

A perícia realizada após um acidente de trânsito consiste na realização de uma investigação técnica dos componentes da cena, como o automóvel, o ambiente e marcas de frenagem deixadas, com o objetivo de identificar as causas e a dinâmica do evento. Porém, a forma com que a perícia no Brasil é realizada atualmente pode tornar esse processo demorado e impreciso.

O avanço da tecnologia possibilita a implantação de medidas que aumentam a segurança nas estradas. O desenvolvimento de dispositivos e o aprimoramento de algumas peças dos automóveis buscam diminuir a incidência de algumas das possíveis causas de acidentes de trânsito, tornando os automóveis mais seguros.

1.1 Motivação

O laudo pericial tem o intuito de fornecer elementos necessários para apontar a materialidade da infração e possui caráter científico. Porém, esse laudo pode sofrer influência do trabalho de isolamento e de preservação do local. Isso acontece pois os vestígios deixados no local, como marcas de frenagem, fragmentos desprendidos e deformação do veículo, são determinantes para a análise.

Os automóveis possuem um módulo chamado *ACM* (*Airbag Control Module*) responsável pelo controle dos *airbags*. Esse módulo é um dispositivo do tipo *EDR* (*Event Data Recorder*). *EDR*s são capazes de guardar informações do veículo como, por exemplo, a utilização ou não de cintos de segurança, o acionamento do freio e o giro do veículo no momento da colisão. Isso é possível devido a leitura de sensores espalhados pelo veículo e à presença de uma memória não volátil para armazenar os dados obtidos. Porém, a forma de armazenamento não é padronizada o que dificulta, e na maioria dos casos impossibilita, a utilização dos dados obtidos para análise do evento.

A presença, em veículos automotivos, de um hardware capaz de obter dados do veículo como aceleração, giro, localização e velocidade de forma padronizada combinados com um software que, através dos parâmetros obtidos, seja capaz de realizar uma simulação 3D, podem agilizar a elaboração de um laudo técnico e o entendimento da dinâmica do acidente ocorrido.

Entre os objetivos da perícia, o de identificar as trajetórias dos veículos, calcular as velocidades de cada um deles antes e após o acidente e examinar o local para determinar a dinâmica do acidente pode ser obtido de forma mais simples e confiável se os dados do veículo antes mesmo do acidente ocorrer forem armazenados em um dispositivo.

A análise dos dados obtidos em um acidente de trânsito, além de ajudar na determinação da causa específica do acidente ocorrido, também pode ser comparada com outros dados de acidentes que ocorreram no mesmo local ou em uma dinâmica parecida, e então ser utilizados para prevenir e mitigar os riscos desse tipo de acidente de trânsito. Esses dados podem ser utilizados para melhorar a segurança dos veículos e das estradas por meio da proposição de medidas que diminuam os acontecimentos, como melhorar sinalizações ou redefinir velocidades de vias.

1.2 Objetivos

Diante do problema especificado, este trabalho tem como objetivo principal o desenvolvimento de um *hardware* capaz de obter parâmetros de um veículo em movimento, e, em caso de acidente, armazená-los para uma análise posterior. Será estudado um componente de baixo custo disponível para a obtenção desses dados. E, além do *hardware*, para uma análise mais completa, o desenvolvimento de um *software* para visualização dos dados após processamento.

Este trabalho é uma continuação da dissertação de mestrado de Vinícius de Oliveira Lima, *Proposta de Plataforma Inercial para Auxiliar na Perícia de Acidentes de Trânsito* [12], e dos trabalhos de graduação de Hudson Pereira Ramos e Vanessa Oliveira Lucena,

Proposta de Plataforma Inercial e Simulador 3D para Periciar Acidentes de Trânsito [13], e João Victor Romualdo e Daniel Serra, *Caixa Preta para Veículos Automotivos* [5].

Como objetivo específico, este trabalho busca realizar vários ensaios para que os dados sejam observados e validados com a utilização do *hardware* e *software* desenvolvidos. A ideia é abranger algumas das situações possíveis e propor melhorias para a obtenção e tratamento dos dados.

1.3 Estrutura do Trabalho

O trabalho está dividido em 5 capítulos. Neste Capítulo 1 encontra-se o contexto, motivação e objetivos do trabalho.

O Capítulo 2 apresentará a fundamentação teórica contendo os principais conceitos sobre os elementos que compõem a plataforma desenvolvida.

O Capítulo 3 explicará o desenvolvimento do hardware e do software utilizados para aquisição dos dados e simulação do acidente.

O Capítulo 4 apresentará os resultados obtidos com todos os testes de campo realizados e uma análise dos dados resultantes.

E, por fim, o Capítulo 5 apresentará as principais conclusões e recomendações para pesquisas futuras.

Capítulo 2

Fundamentação Teórica

Este capítulo abordará os principais conceitos dos elementos que compõem a plataforma. Serão descritos os componentes utilizados e suas funcionalidades em cada uma das seguintes etapas: aquisição e armazenamento dos dados, comunicação entre os dispositivos e processamento dos dados. E, por fim, serão explicados conceitos de representações de rotações em três dimensões e filtragem dos dados obtidos.

2.1 Aquisição dos Dados

A detecção e o acompanhamento de movimentos desempenha um papel muito importante em uma grande variedade de aplicações. O rastreamento da posição e orientação de um objeto em relação a um ponto de partida é muito utilizado em áreas como navegação, aviação e robótica.

Para este propósito, vários sensores podem ser empregados para aquisição de dados utilizados na detecção e acompanhamento de movimentos. Neste trabalho foram utilizados dois tipos de sensores, um módulo inercial de medição de movimento composto de um conjunto de sensores inerciais e um receptor GPS.

A combinação dos dados obtidos por esses dois sensores fornece um resultado satisfatório para a posição, orientação e localização de objetos. A quantidade de parâmetros obtidos tem como finalidade possibilitar a eliminação de alguns erros e ruídos.

2.1.1 Unidade de Medição Inercial

Unidades de Medição Inercial, ou IMU (*Inertial Measurement Units*), são dispositivos utilizados para a detecção e acompanhamento de movimentos por meio de dados obtidos de um conjunto de sensores inerciais contidos neles. O IMU utilizado neste trabalho foi o MPU-9250 [6], um dispositivo de rastreamento de movimentos de baixo custo.

O MPU-9250, representado em uma placa (MPU-92/65) na Figura 2.1, é um módulo composto de um acelerômetro, um giroscópio e um magnetômetro, o AK8963 [14], todos de 3 eixos (x, y, z). Ele possui então 9 graus de liberdade, o que possibilita uma melhor eliminação de erros e ruídos provenientes dos sinais de cada um dos sensores.

Figura 2.1: Módulo MPU-9250.

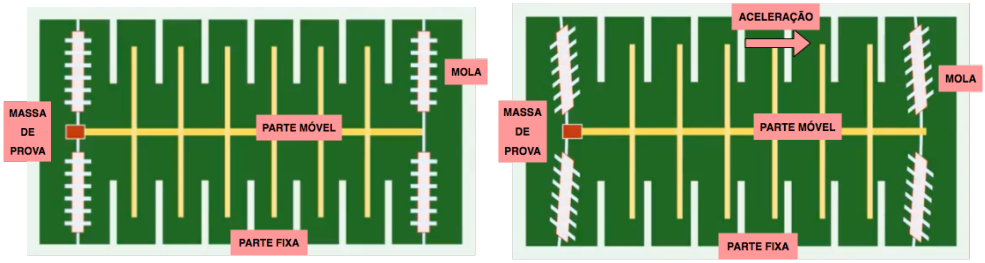
Os sensores inerciais contidos no MPU são baseadas na tecnologia de Sistemas Micro Eletromecânicos, ou MEMS (*Micro Electro Mechanical System*). Sensores MEMS podem ser definidos de forma geral como elementos mecânicos e eletromecânicos miniaturizados que são feitos utilizando técnicas de microfabricação [15].

Acelerômetro

Acelerômetros de 3 eixos são dispositivos capazes de obter a aceleração de um corpo em três eixos (**x**, **y** e **z**). Essa aceleração é medida em relação ao campo gravitacional da Terra, expressa em g's. Os acelerômetros são úteis para detectar vibrações em sistemas ou para aplicações de orientação.

Os valores obtidos podem ser integrados para se obter velocidade e posição de um objeto. Porém, em geral, estimativas de posição e velocidade baseadas em acelerômetros de baixo custo não são precisas e apresentam vários erros. Neste trabalho, os valores de aceleração obtidos nos três eixos serão usados em combinação com dados obtidos por outros sensores para gerar um resultado satisfatório.

A Figura 2.2a ilustra um acelerômetro MEMS. Esse tipo de sensor é composto de uma massa de prova, molas e uma parte fixa e uma móvel formando pares de capacitores. O princípio de funcionamento desse acelerômetro pode ser observado na Figura 2.2b. Quando uma aceleração é aplicada para a direita as placas sofrem um deslocamento também para a direita e a capacitância entre elas muda. A partir dessas mudanças na capacitância, a aceleração pode ser determinada.



(a) Acelerômetro em repouso (b) Acelerômetro MEMS acelerado

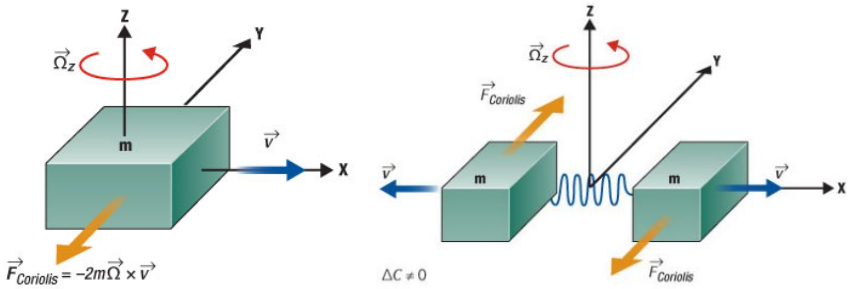
Figura 2.2: Funcionamento de um acelerômetro MEMS [1]

Giroscópio

Giroscópios são dispositivos capazes de medir velocidades angulares de um corpo. O giroscópio MEMS utiliza um princípio da física chamado Efeito de Coriolis, que ajuda a medir rotações ou velocidades angulares.

O Efeito de Coriolis, ilustrado na Figura 2.3a, define que quando um corpo de massa m movendo em uma direção com uma certa velocidade v sofre uma velocidade angular Ω, uma força de Coriolis vai ser sentida pelo corpo. Essa força é determinada em função da massa, velocidade e velocidade angular.

Na Figura 2.3b podemos observar um giroscópio MEMS formado por duas massas de prova. Essas massas estão se movendo em direções opostas continuamente. Quando o conjunto sofre uma rotação angular, as duas massas experienciam uma força de Coriolis, porém em sentidos opostos, fazendo que as massas se movam, provocando um deslocamento que altera as capacitâncias, da mesma forma que nos acelerômetros MEMS.



(a) Efeito de Coriolis (b) Giroscópio MEMS

Figura 2.3: Funcionamento de um giroscópio MEMS [2]

O giroscópio presente no IMU mede, então, velocidades angulares, as quais, sendo a posição inicial conhecida, podem ser integradas ao longo do tempo para computar a orientação do sensor. Porém essa integração deve ser feita com ressalvas pois é imprecisa e sujeita a vários erros.

Magnetômetro

O magnetômetro MEMS é utilizado para detectar e medir os campos magnéticos presentes próximo ao dispositivo. O magnetômetro de 3 eixos usa tecnologia de sensor Hall altamente sensível como abordagem para detecção magnética.

A Figura 2.4 ajuda a ilustrar o efeito de Hall. Uma corrente flui pela placa condutora em uma linha reta de uma borda para outra da placa. A presença de um campo magnético próximo a placa, faz com que o fluxo reto das cargas seja alterado devido a uma força chamada de força de Lorentz. O que causa uma polarização, as cargas positivas de um lado da placa e as cargas negativas no lado oposto, e assim o surgimento de uma tensão. A obtenção de uma tensão que pode ser medida é conhecida como efeito de Hall.

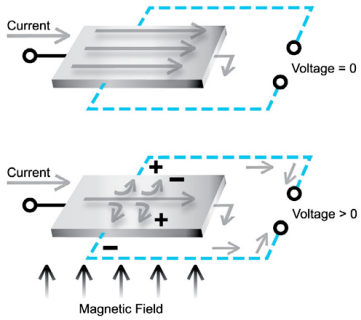


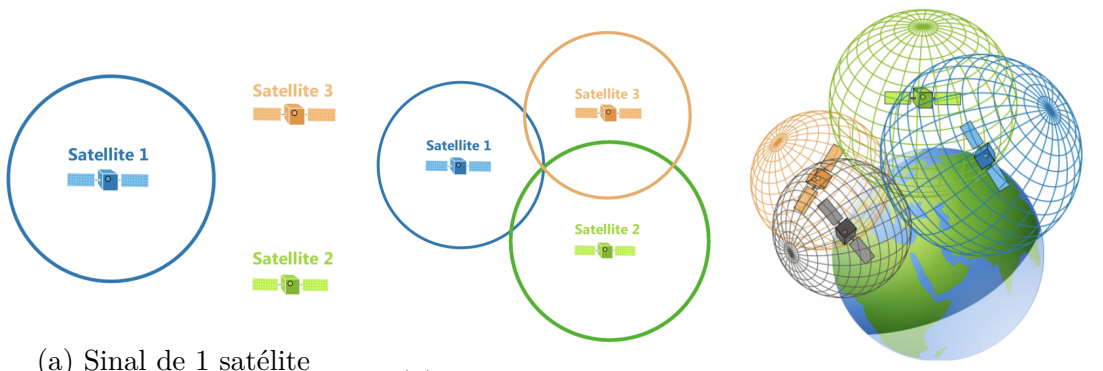
Figura 2.4: Efeito de Hall.

2.1.2 Sistema de Posicionamento Global

O Sistema de Posicionamento Global, ou GPS (*Global Positioning System*) é um sistema de navegação por satélites. Existem vários sistemas desse tipo, mas o GPS, desenvolvido pelos Estados Unidos na década de 1970, e formado por mais de 20 satélites, é o mais conhecido e utilizado. Nesses sistemas é possível obter informações de localização, data e hora por meio de sinais provenientes de satélites.

Um receptor GPS é um dispositivo capaz de receber os sinais de satélites e realizar cálculos para obter as informações de geolocalização, data e hora. Isto depende da localização dos satélites, já que a posição precisa do receptor é determinada a partir da distância dele até algumas posições conhecidas no espaço [16].

A técnica utilizada por esses receptores é chamada trilateração. Todos os satélites transmitem um sinal e quando um desses sinais atinge o receptor GPS, a distância é conhecida e forma-se um círculo com ela, como é mostrado na Figura 2.5a, o que significa que a posição do GPS pode estar em qualquer lugar no círculo formado por este raio.



(a) Sinal de 1 satélite

(b) Sinais de 3 satélites

(c) Sinais formam esferas

Figura 2.5: Trilateração usada por receptores GPS [3]

O mesmo se repete para o sinal recebido de um segundo satélite e de um terceiro, que é quando é possível obter apenas uma localização. Como podemos ver na Figura 2.5b, cada satélite está no centro de uma esfera e a interseção dessas esferas define a localização. Porém, em geral quatro satélites devem ser usados para determinar uma posição precisa [16].

O receptor GPS utilizado foi o módulo GY-NEO6MV2 que combina um alto nível de capacidade de integração com opções de conectividade flexíveis em um pacote pequeno [17]. Além disso, possui muitos recursos e configurações disponíveis permitindo as informações obtidas à finalidade desejada.

2.2 Armazenamento dos Dados

O funcionamento do projeto é baseado inicialmente na obtenção de vários dados e em seguida na utilização deles para a análise, logo, esses dados devem ser armazenados, e constantemente atualizados, durante o funcionamento da "caixa preta", e por fim, armazenados permanentemente após evento.

Duas memórias foram utilizadas para armazenar os dados necessários. Uma delas, volátil, responsável por guardar todos os dados enquanto ainda não ocorreu um acidente, e a outra, não volátil, na qual os dados vão ser gravados apenas quando um acidente ocorrer. Dessa última, os dados serão recuperados posteriormente para reconstrução do acidente.

2.2.1 SRAM

A SRAM (*Static Random Access Memory*) é um tipo de memória volátil, a qual retém os dados apenas enquanto está conectada à alimentação. Outras características da SRAM

são o acesso direto a endereços específicos, ou seja, não é preciso percorrer toda a memória, e o suporte a um número ilimitado de leituras e escritas no dispositivo.

A memória SRAM utilizada foi a 23LC1024 [18], com a qual é possível se comunicar utilizando uma interface SPI, descrita na seção seguinte. Ela possui um espaço para armazenamento de 128 Kbytes e trabalha com páginas de 32 bytes. Os modos de acesso disponíveis que podem ser selecionados através do registrador de modo são byte, página e sequencial.

Os sinais necessários são uma entrada de relógio (SCK), uma linha de entrada (SI) e uma linha de saída de dados (SO). O acesso ao dispositivo é controlado por meio de uma entrada Chip Select (CS). O CS deve estar em nível baixo para que leituras e escritas sejam feitas.

2.2.2 EEPROM

A EEPROM (*Eletrically Erasable Programmable Read Only Memory*) é uma memória não-volátil, ou seja, que retém os dados mesmo quando a alimentação é removida. O acesso aos dados pode ser aleatório ou sequencial, sendo o acesso sequencial mais rápido. Suporta em média um milhão de escritas antes de ser danificada.

A memória EEPROM utilizada foi a 24LC1025 [19], que possui um espaço para armazenamento de 128 Kbytes. Essa memória possui uma interface serial, logo a comunicação foi realizada utilizando I^2C, um tipo de comunicação serial que será explicado no próximo tópico.

2.3 Processamento

O Arduino Mega 2560, baseado no microcontrolador ATmega2560, foi escolhido para realizar o processamento dos dados obtidos pelos sensores já descritos. Esse dispositivo possui 54 pinos digitais de entrada/saída, 16 entradas analógicas, 4 portas de comunicação serial, uma conexão USB, conector de alimentação e um botão de reset [20]. A programação foi realizada utilizando a IDE própria do Arduino.

O microcontrolador possui as seguintes especificações de memória: 256KB de memória de programa, 8KB de memória SRAM e 4KB de memória EEPROM. Cada leitura de dados de aceleração, giro e campo magnético, contém 18 bytes. Considerando cerca de 100 leituras por segundo, a capacidade da memória não-volátil esgotaria em menos de 3 segundos. Dessa forma, torna-se necessário a utilização de memórias externas com maior capacidade.

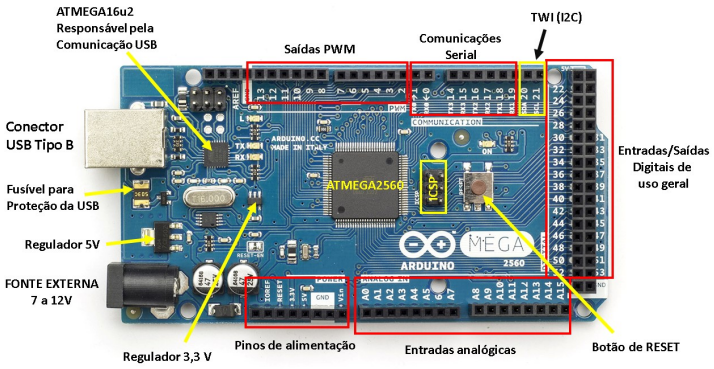


Figura 2.6: Arduino Mega 2560.

2.4 Comunicação

Os diversos componentes do projeto precisam se comunicar e para isso foram utilizados alguns protocolos de comunicação. Existem vários tipos de protocolos e os usados neste projeto e que foram "impostos" pelos periféricos selecionados são: I^{2}C e SPI, que serão explicados a seguir.

2.4.1 I^{2}C

I^{2}C (*Inter-IC Bus*) é um protocolo do tipo mestre/escravo, desenvolvido pela Philips, e que necessita de apenas dois fios para comunicação bidirecional, uma linha de dados serial SDA (*Serial Data*) e uma linha de clock serial SCL (*Serial Clock*). Um dispositivo mestre pode se comunicar com vários escravos e é ele que controla a linha de clock.

A comunicação do Arduino com o MPU e com a memória EEPROM foi realizada utilizando I^{2}C. Cada um desses dispositivos possui um endereço de escrita e um de leitura pré-definidos que podem ser encontrados no manual. A comunicação é feita com apenas um dos dispositivos por vez.

O endereço dos escravos é utilizado pelo mestre, o Arduino, que deve enviá-lo ao iniciar uma transação, para que o dispositivo desejado seja selecionado. O mestre é o responsável por iniciar e finalizar transações. O dispositivo que recebe dados responde com um ACK *(acknowledgement)* a cada dados recebido.

2.4.2 SPI

SPI (*Serial Peripheral Interface*) é uma interface de comunicação composta por quatro linhas, sendo elas: MOSI *(Master Output Slave Input)*, ou seja, saída do mestre e entrada do escravo; MISO *(Master Input Slave Output)*, entrada do mestre e saída do escravo;

SCLK, linha com o relógio gerado pelo mestre e SS, linha de seleção para escolher o dispositivo com o qual você deseja conversar, a qual deve ser conectada a entrada CS (ou SS) do escravo. A Figura 2.7 ilustra um exemplo de configuração de um barramento SPI.

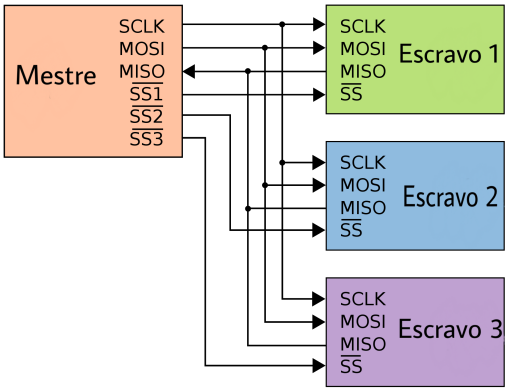


Figura 2.7: Exemplo de barramento SPI com 2 escravos.

2.5 Rotação de Objetos em três dimensões

A rotação de objetos em relação a uma posição inicial pode ser representada de algumas formas. Duas dessas alternativas serão descritas aqui, a de ângulos de Euler e a de quaternions.

2.5.1 Ângulos de Euler

Ângulos de Euler é uma representação de rotações tridimensionais de objetos com relação a uma origem inicial no espaço. Consiste na utilização de um conjunto de três ângulos que definem, cada um, uma rotação com relação a cada um dos eixos, x, y e z. Por utilizar ângulos, essa é uma representação mais intuitiva do que a de quaternions, que foi descrita na seção anterior.

A convenção utilizada para os ângulos de Euler deve ser observada, pois o conhecimento dos ângulos sem a informação da convenção utilizada pode não levar ao resultado correto. Isso acontece pois rotações tridimensionais não são comutativas, ou seja, a ordem dos operandos altera o resultado final. Aplicar uma rotação primeiro sobre o eixo x produzirá um resultado diferente ao de uma rotação iniciando pelo eixo y.

Existem várias convenções para os ângulos de Euler que interferem na interpretação da rotação realizada. A convenção de *Tait-Bryan* será a utilizada. Nessa convenção, cada um dos ângulos representa uma rotação em torno de um eixo diferente. Rotações sucessivas

são realizadas sobre os eixos resultantes de rotações anterioes, e não em relação aos eixos do sistema de coordenadas origial.

Os ângulos são chamados *roll*, *picth* e *yaw*. A convenção *yaw-picth-roll* pode ser visualizada como a mudança na orientação de uma aeronave da perspectiva do piloto, como podemos observar na Figura 2.8. Inicialmente, o sistema de coordenadas da aeronave está alinhado com o sistema de coordenadas da Terra.

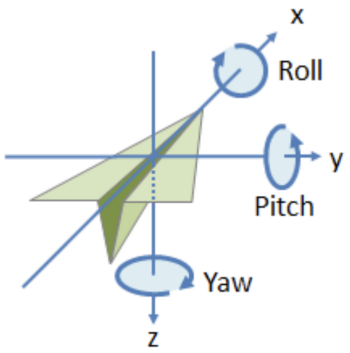


Figura 2.8: Ângulos de Euler.

2.5.2 Quaternions

Quaternions é uma forma de representação de rotações em um espaço tridimensional concebida por William Rowan Hamilton, em 1843 [21]. Hamilton inventou a ideia de um sistema imaginário de 3 partes que se transformou na álgebra de Quaternions [22].

Quaternions são definidos como vetores de quatro dimensões e representados pela Equação 2.1. Eles são compostos por uma parte escalar q_0 e uma parte vetorial $\mathbf{q}$. Esses números são utilizados para representar a orientação de um corpo rígido no espaço tridimensional.

$$q = (q_0, q_1, q_2, q_3) = (q_0, \mathbf{q}) \tag{2.1}$$

A Equação 2.2, se assemelha à de números complexos, na qual i, j e k são unidades imaginárias e representam a parte imaginária do quaternion.

$$i^2 = j^2 = k^2 = ijk = -1 \tag{2.2}$$

Alguns conceitos importantes para o entendimento de Quaternions serão mostrados nas equações a seguir:

- Representação de um vetor em 4 dimensões:

$$\mathbf{u} = (w, x, y, z) \tag{2.3}$$

- Cálculo do tamanho de um vetor:

$$\|\mathbf{u}\| = \sqrt{w^2 + x^2 + y^2 + z^2} \tag{2.4}$$

- Vetores unitários, obtidos pela divisão de um vetor pelo seu tamanho:

$$\hat{u} = \frac{\mathbf{u}}{\|\mathbf{u}\|} \tag{2.5}$$

A Figura 2.9 ilustra os três eixos mutuamente ortogonais de dois *frames*, A e B, e um eixo $\hat{A}_r$ definido no *frame* A. Um quaternion descrevendo uma orientação no *frame* B em relação ao *frame* A, obtido através de uma rotação (de θ graus) em torno do eixo $\hat{A}_r$, é definido pela equação Equação 2.6. $\hat{A}_r$ é um vetor unitário cujos componentes são r_x, r_y e r_z, nos eixos x, y e z do *frame* A.

$$\overset{A}{B}\hat{q} = [q_0, q_1, q_2, q_3] = [cos\frac{\theta}{2}\ ,\ \ -r_x sin\frac{\theta}{2}\ ,\ \ -r_y sin\frac{\theta}{2}\ ,\ \ -r_z sin\frac{\theta}{2}] \tag{2.6}$$

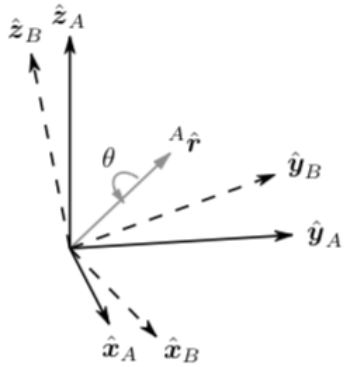


Figura 2.9: Orientação obtida por uma rotação de um ângulo θ em torno do eixo $\hat{A}_r$ [4].

Logo, no formato $\overset{A}{B}\hat{q}$ da representação de uma orientação, o valor subscrito representa o frame que está sendo descrito e o valor sobrescrito denota o frame de referência. O conjugado de um quaternion dado pela Equação 2.7 é utilizado para trocar os frames descrito e de referência.

$$\overset{A}{B}\hat{q}^* = \overset{B}{A}\ \hat{q} = [q_0,\ -q_1,\ -q_2,\ -q_3] \tag{2.7}$$

Dados dois quaternions de orientação, o do frame C relativo ao frame B ($^{B}_{C}\hat{q}$) e o do frame B relativo ao frame A ($^{A}_{B}\hat{q}$), a orientação composta do frame C relativo ao frame A é definida pelo produto vetorial dos dois quaternions (Equação 2.8). O produto vetorial não é comutativo e um exemplo da operação, considerando dois quaternions $q = [q_0\ q_1\ q_2\ q_3]$ e r $= [r_0\ r_1\ r_2\ r_3]$, é dado pela Equação 2.9.

$$^{A}_{C}\hat{q} \ = \ ^{B}_{C}\hat{q} \ \otimes \ ^{A}_{B}\hat{q} \tag{2.8}$$

$$\begin{aligned} q \otimes r &= (q_0, q_1, q_2, q_3) \otimes (r_0, r_1, r_2, r_3) \\ q \otimes r &= \begin{bmatrix} q_0 r_0 - q_1 r_1 - q_2 r_2 - q_3 r_3 \\ q_0 r_1 + q_1 r_0 + q_2 r_3 - q_3 r_2 \\ q_0 r_2 - q_1 r_3 + q_2 r_0 + q_3 r_1 \\ q_0 r_3 + q_1 r_2 - q_2 r_1 + q_3 r_0 \end{bmatrix}^T \end{aligned} \tag{2.9}$$

Após todas essas explicações sobre a álgebra de quaternions, finalmente a rotação, utilizando um quaternion, de um vetor de 3 dimensões pode ser obtida pela Equação 2.10, onde A_v e B_v representam o mesmo vetor descrito nos frames A e B, e orientações compostas são aplicadas.

$$B_v = \ ^{A}_{B}\hat{q} \ \otimes \ A_v \ \otimes \ ^{A}_{B}\hat{q}^{*} \tag{2.10}$$

Porém, alega-se que pouco importa a facilidade em extrair algum significado físico dos quaternions, pois eles podem ser facilmente convertidos para ângulos de Euler e que, estes sim, possuem a vantagem de permitirem o imediato entendimento físico. As Equações 2.11 a 2.13 definem a conversão para a representação por ângulos de Euler.

$$\psi = Atan2(2q_2q_3 - 2q_1q_4, 2{q_1}^2 + 2{q_2}^2 - 1) \tag{2.11}$$

$$\theta = -sin^{-1}(2q_2q_4 + 2q_1q_3) \tag{2.12}$$

$$\phi = Atan2(2q_3q_4 - 2q_1q_2, 2{q_1}^2 + 2{q_4}^2 - 1) \tag{2.13}$$

2.6 Filtro de Madgwick

Madgwick desenvolveu, em 2010, um filtro de orientação aplicável a IMUs compostos de giroscópios e acelerômetros de três eixos e também para IMUs que possuem um magnetô-

metro de três eixos. Um filtro de orientação é capaz de computar a orientação de um dispositivo através da fusão dos dados medidos pelos sensores.

O filtro de Madgwick utiliza uma representação por quaternions, necessita de pouco processamento computacional, funciona em baixas taxas de amostragem e contém um parâmetro ajustável, beta (β), de acordo com a observação de características do sistema [4]. Ele propõe soluções para algumas desvantagens do filtro de Kalman, que é base para a grande maioria de algoritmos de filtros de orientação.

Neste filtro, uma estimativa de orientação, $\hat{q}_{est,t}$, do frame do sensor em relação ao frame da Terra pode ser obtida por meio da fusão dos cálculos de orientação ${}^{S}_{E}q_{w,t}$ e ${}^{S}_{E}q_{\nabla,t}$. Essa fusão é calculada utilizando a Equação 2.14, onde γ_t e $(1-\gamma_t)$ são pesos aplicados a cada cálculo de orientação.

$$ {}^{S}_{E}q_{est,t} = \gamma_t \, {}^{S}_{E}q_{\nabla,t} + (1-\gamma_t) \, {}^{S}_{E}q_{w,t}, \; 0 \leq \gamma_t \leq 1 \tag{2.14} $$

O primeiro cálculo de orientação, ${}^{S}_{E}q_{w,t}$, corresponde a orientação no frame da Terra em relação ao frame do sensor no instante t e pode ser calculada integrando-se numericamente a derivada do quaternion, ${}^{S}_{E}\dot{q}_{w,t}$, desde que as condições iniciais sejam conhecidas. A Equação 2.16 e a Equação 2.15, onde ${}^{S}w_t = [0 \; w_x \; w_y \; w_z]$ é um vetor contendo as taxas angulares medidas pelo giroscópio no instante t, mostram como é feito esse cálculo. Os outros parâmetros são: Δt, que é o período de amostragem e ${}^{E}_{S}\hat{q}_{est,t-1}$, que corresponde a estimativa anterior de orientação.

$$ {}^{S}_{E}\dot{q}_{w,t} = \frac{1}{2} \, {}^{S}_{E}\hat{q}_{est,t-1} \otimes {}^{S}w_t \tag{2.15} $$

$$ {}^{S}_{E}q_{w,t} = {}^{S}_{E}\hat{q}_{est,t-1} + {}^{S}_{E}\dot{q}_{w,t}\Delta t \tag{2.16} $$

O segundo cálculo de orientação, ${}^{S}_{E}q_{\nabla,t}$, é realizado com a Equação 2.17, calculado em um instante t e baseado em uma estimativa anterior de orientação ${}^{S}_{E}\hat{q}_{est,t-1}$ e pela função gradiente ∇f definida por medições do acelerômetro ${}^{S}\hat{a}_t$, sendo ${}^{S}\hat{a} = [0 \;\; a_x \;\; a_y \;\; a_z]$, amostradas no tempo t (Equação 2.18).

$$ {}^{S}_{E}q_{\nabla,t} = {}^{S}_{E}\hat{q}_{est,t-1} - \mu_t \frac{\nabla f}{\|\nabla f\|} \tag{2.17} $$

$$ \nabla f = J_g^T({}^{S}_{E}\hat{q}_{est,t-1}) f_g({}^{S}_{E}\hat{q}_{est,t-1}, {}^{S}\hat{a}_t) \tag{2.18} $$

A representação indica que o quaternion é calculado usando um algoritmo de descida por gradiente. Esse algoritmo é utilizado em problemas de otimização e consiste na

realização de múltiplas iterações para a obtenção de uma estimativa de orientação baseada em um palpite inicial e um tamanho de passo μ_t.

O quaternion pode ser obtido por um problema de otimização pois a orientação do sensor, ${}^{S}_{E}\hat{q}$, é aquela que alinha uma direção de referência pré-definida do campo no frame da Terra, ${}^{E}\hat{d}$, com a direção medida do campo no frame do sensor, ${}^{S}\hat{s}$, usando a operação de rotação descrita na Equação 2.10.

Assim, o quaternion ${}^{S}_{E}\hat{q}$ pode ser obtido com a Equação 2.19. A função f é calculada com a Equação 2.20, onde cada um dos parâmetros é definido com as Equações 2.21 a 2.23.

$$\min_{{}^{S}_{E}\hat{q}\epsilon\mathbb{R}^4} f({}^{S}_{E}\hat{q},\ {}^{E}\hat{d},\ {}^{S}\hat{s}) \tag{2.19}$$

$$f({}^{S}_{E}\hat{q},\ {}^{E}\hat{d},\ {}^{S}\hat{s}) = {}^{S}_{E}\hat{q}^{*} \otimes {}^{E}\hat{d} \otimes {}^{S}_{E}\hat{q} - {}^{S}\hat{s} \tag{2.20}$$

$${}^{S}_{E}\hat{q} = [q_1\ q_2\ q_3\ q_4] \tag{2.21}$$

$${}^{E}\hat{d} = [0\ d_x\ d_y\ d_z] \tag{2.22}$$

$${}^{S}\hat{s} = [0\ s_x\ s_y\ s_z] \tag{2.23}$$

Logo, o algoritmo de descida por gradiente para esta aplicação, considerando uma convenção que assume que a direção da gravidade define o eixo vertical z, pode ser descrito pelas Equações 2.24 a 2.27, onde ${}^{E}\hat{d}$ e ${}^{S}\hat{s}$ foram substituídos por ${}^{E}\hat{g}$ e ${}^{S}\hat{a}$, respectivamente.

$${}^{E}\hat{g} = [0\ 0\ 0\ 1] \tag{2.24}$$

$${}^{S}\hat{a} = [0\ a_x\ a_y\ a_z] \tag{2.25}$$

$$f_g({}^{S}_{E}\hat{q}, {}^{S}\hat{a}) = \begin{bmatrix} 2(q_2q_4 - q_1q_3) - a_x \\ 2(q_1q_2 - q_3q_4) - a_y \\ 2(\frac{1}{2} - q_2^2 - q_3^2) - a_z \end{bmatrix} \tag{2.26}$$

$$J_g({}^{S}_{E}\hat{q}) = \begin{bmatrix} -2q_3 & 2q_4 & -2q_1 & 2q_2 \\ 2q_2 & 2q_1 & 2q_4 & 2q_3 \\ 0 & -4q_2 & -4q_3 & 0 \end{bmatrix} \tag{2.27}$$

O diagrama de blocos deste filtro está representado na Figura 2.10.

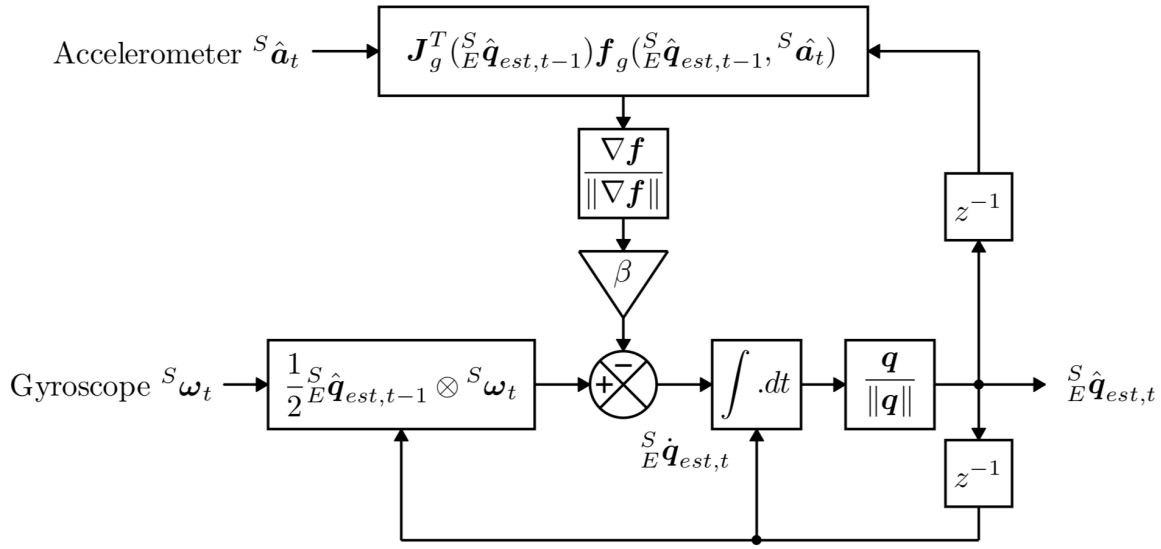


Figura 2.10: Representação em diagrama de blocos do filtro de orientação para um IMU.

Capítulo 3

Hardware e Software

Este capítulo explica o desenvolvimento e o funcionamento do hardware responsável por toda a coleta, processamento e armazenamento dos dados, e do software, um simulador desenvolvido utilizando a plataforma de jogos Unity 3D.

3.1 Hardware

A Figura 3.1 mostra o protótipo construído para esse projeto e os esquemáticos estão disponíveis no Apêndice B. Além dos componentes já explicados: IMU e receptor GPS para aquisição de dados, Arduino Mega 2560 para o processamento e memórias para armazenar temporariamente (SRAM) e permanentemente (EEPROM) os dados obtidos, também foram utilizados alguns componentes para facilitar o desenvolvimento e a utilização do protótipo. Esses componentes são um display LCD, três LEDs (vermelho, amarelo e verde) e 6 botões.

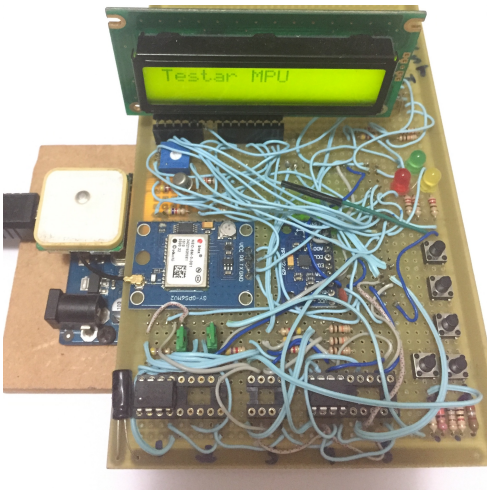


Figura 3.1: Protótipo.

Inicialmente, ao conectar o protótipo, é realizada uma fase de configuração, onde são iniciadas as comunicações seriais, o IMU tem sua comunicação testada, calibrações e testes realizados. Em seguida, o display LCD exibe as funcionalidades disponíveis, e a seleção é realizada com auxílio dos botões. Eles foram utilizados da seguinte forma: dois para navegar entre as funções existentes, um para selecionar a função desejada e um para reiniciar. As funcionalidades implementadas são:

- **Testar MPU:** Exibição dos dados brutos ou já ajustados para as resoluções adequadas, para que seja possível verificar a comunicação com o MPU e a coerência dos dados obtidos. Essa função também foi usada para, durante o desenvolvimento, gerar dados nas saídas em um formato adequado para os testes realizados, discutidos no próximo capítulo. O interrompimento desta função ocorre quando qualquer botão do protótipo é pressionado. A Figura [] representa os dados brutos obtidos com essa função.

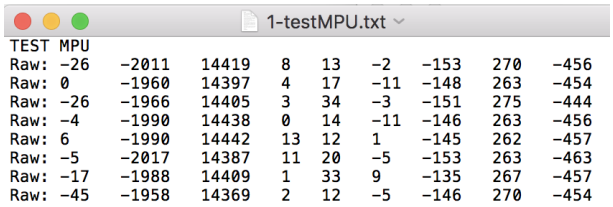


Figura 3.2: Saída da função Testar MPU.

- **Testar GPS:** Exibição em laço de dados do GPS obtidos no momento. Os dados exibidos, como na Figura 3.3, são os mesmos gravados na EEPROM quando um evento ocorre: dados de localização (latitude e longitude), velocidade, data e hora.

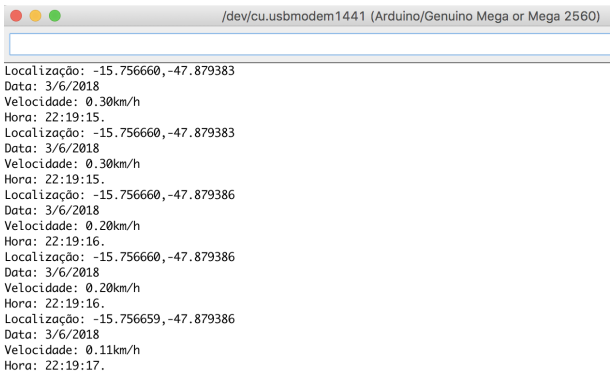


Figura 3.3: Saída da função Testar GPS.

- **Rodar rotina:** Leitura em loop dos dados do MPU e do GPS. Todos os dados são gravados na SRAM, ciclicamente, e a transferência para a EEPROM acontece

quando um evento ocorre. O funcionamento mais detalhado desta função será descrito na seção 3.1.4.

- **Ler dados EEPROM:** Exibição de todo o conteúdo da memória EEPROM que consiste nos dados transferidos da memória SRAM para a EEPROM pela rotina principal quando um evento ocorre. Os dados exibidos aqui são brutos, sem correção de resolução ou passagem de filtros. A saída pode ser vista na Figura 3.4.

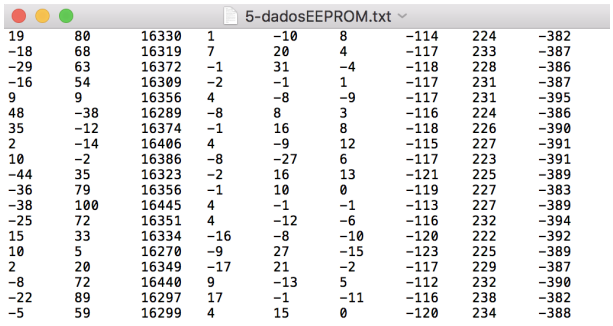


Figura 3.4: Saída da função Ler dados EEPROM.

- **Ler dados GPS:** Exibição dos dados de localização, data, hora e velocidade obtidos durante a execução da rotina principal.

- **Tratar dados:** Nesta função os dados gravados na EEPROM são lidos e aqui é aplicado o filtro de Madgwick, que faz a fusão dos dados de aceleração e giro e gera quaternions que representam as rotações. Os dados são então exibidos no monitor serial na ordem quatro quaternions (q_0, q_1, q_2, q_3) e três acelerações (a_x, a_y, a_z).

3.1.1 Configuração do MPU-9250

O MPU-9250 fornece dados de 16 bits para a aceleração, velocidade angular e campo magnético nas direções x, y e z. As medições do acelerômetro são em relação a aceleração da gravidade (g = 9,8m/s^2) e podem ser obtidas nas escalas de $\pm 2g$, $\pm 4g$, $\pm 8g$ e $\pm 16g$. Para o giroscópio as escalas possíveis são $\pm 250 graus/s$, $\pm 500 graus/s$, $\pm 1000 graus/s$ e $\pm 2000 graus/s$.

O IMU é configurado a partir de vários registradores. A comunicação com o Arduino foi realizada utilizando uma interface I^2C. Com o auxílio de um *driver* fornecido pelo fabricante *Open Hardwares* Sparkfun para operações com o MPU9250, uma sequência de etapas foi realizada. O código dessa biblioteca está disponível no Apêndice A.3. Foram utilizadas as funções de leituras e escritas de bytes, self-test, calibração do giroscópio e do magnetômetro. Os passos realizados foram:

1. **Testar comunicação:** Para testar a comunicação com o IMU, o conteúdo do registrador WHOAMI é lido e deve retornar, no caso do MPU9250, o valor 0x73.

2. **Self test:** Esta etapa é realizada para testar as porções elétricas e mecânicas dos sensores. A porção eletrônica faz com que os sensores sejam acionados e produzam um sinal de saída que é então usado para observar a resposta do self-test. O dispositivo possui vários sensores de self-test e seus valores indicam a saída do self-test gerada durante os testes de fabricação. Quando o valor do self-test está nos limites apropriados, significa que a parte passou no teste.

3. **Calibração acelerômetro e giroscópio:** É acumulada uma sequência de leituras do acelerômetro e depois a média dessas leituras é calculada. O mesmo acontece para os dados do giroscópio. Esses offsets são carregados em variáveis para serem descontados das leituras realizadas. As variáveis são dois vetores **gyroBias[3]** e **accelBias[3]**. A calibração do acelerômetro deve ser feita quando não há movimento.

4. **Inicialização do MPU:** Nesta etapa vários registradores de configuração são ajustados de forma a atender aos requisitos de faixa de operação e sensibilidade exigidas pelo projeto.

3.1.2 Configuração do GPS

A utilização do GPS foi realizada através da interface Serial3 do Arduino e feita com o auxílio da biblioteca TinyGPS++. Essa biblioteca, desenvolvida por Mikal Hart, auxilia na obtenção dos dados GPS [23]. A biblioteca possui vários recursos que facilitam a obtenção dos dados desejados. Algumas das opções disponíveis são: localização, data, hora, velocidade, curso, altitude e número de satélites. Os dados obtidos foram ajustados utilizando o Tempo Universal Coordenado, ou UTC (*Universal Time Coordinated*), do local.

3.1.3 Acesso às memórias

O esquemático contido no Apêndice B mostra como foram feitas as conexões com as memórias. A comunicação com a memória SRAM é feita utilizando a interface SPI. A SRAM possui os seguintes modos de operação: **Byte**(0x00), **Page**(0x80) e **Sequencial**(0x40), sendo este último o utilizado para as leituras e escritas. As funções utilizadas para definição do modo de operação, e para leituras e escritas foram incluídas no Apêndice A.2. A comunicação com a memória EEPROM foi realizada com o auxílio do drive EEPROM Microchip 24XX1025 I^2C para Arduino. Foram utilizadas as funções para leituras e escritas sequenciais, de bytes, floats e inteiros.

3.1.4 Rotina principal de coleta de dados

A ideia principal do projeto é a coleta de vários dados que possam auxiliar no entendimento de um acidente automotivo. Logo, a rotina da "caixa-preta", que deve estar rodando no momento do acidente, para que os dados que o antecedem e o sucedem sejam captados, é o módulo que une grande parte dos componentes e das funcionalidades.

Funcionando em um laço, que é interrompido quando um acidente ocorre, essa rotina lê continuamente dados do MPU e do GPS e funciona da seguinte forma: a cada 32 leituras de 18 bytes (2 bytes para cada um dos três eixos, x, y e z, do acelerômetro, do giroscópio e do magnetômetro), o que totaliza 576 bytes, esses dados são armazenados na memória volátil. E a cada 14 dessas escritas na SRAM, uma amostra do GPS é lida e armazenada em uma lista de estruturas, contendo dados de GPS, que são mantidos em memória e armazenados apenas no fim da EEPROM quando o acidente ocorre. Esse processo fica se repetindo, preenchendo a SRAM ciclicamente.

Quando um evento ocorre, o endereço desse evento é guardado e utilizado para definir o começo dos dados na memória SRAM para que a ordem dos dados copiados para a memória EEPROM seja correta. A quantidade de dados anteriores ao acidente que é armazenada é de 113 leituras do MPU de 576 bytes cada, logo, 65088 bytes. Assim, tendo o endereço do evento, é subtraído deste valor 65088 bytes de dados pré-evento para se obter o endereço de início dos dados na SRAM. Observando para que o valor obtido seja um endereço válido, e caso contrário, basta somar o tamanho em bytes da memória para que o endereço obtido seja correto.

Por ocasião de um evento, após o endereço da SRAM de início do evento ser amarmazenado, a rotina ainda completa 114 leituras de 576 bytes de dados do MPU, totalizando 65664 bytes de dados pós-evento, e mais 8 amostras do GPS. Os dados anteriores e posteriores ao evento são de 65088 bytes e 65664 bytes, respectivamente. A soma desses valores com mais 320 bytes de dados do GPS representa o tamanho total das duas memórias.

Por fim, a partir do endereço de início de dados na SRAM, os dados são copiados para a memória não-volátil. A leitura e escrita é feita em blocos de 128 bytes, que é o maior bloco que a EEPROM aceita para escrita.

3.1.5 Tratamento dos dados

Os dados armazenados na EEPROM são lidos e utilizados como entrada para a função que implementa o filtro de Madgwick (Apêndice A.4). Essa função recebe os dados na ordem aceleração (a_x, a_y, a_z) e giro (g_x, g_y, g_z). A cada iteração é gerado o quaternion referente à rotação realizada.

Essa função exibe no monitor serial os quatro dados do quaternion obtido, seguidos dos dados de aceleração (Figura 3.5). Esse é o formato de entrada para o *software* desenvolvido no Unity.

quatAccel.txt

```
0.99818887 -0.04366906 -0.03985164 0.01112638 -0.07800293 -0.08489991 1.00250244
0.99756698 -0.04384007 -0.05316938 0.01054390 -0.08752442 -0.09088135 1.02258300
0.99838905 -0.03502860 -0.04337853 0.01050971 -0.08172608 -0.06695557 0.99029541
0.99786434 -0.04825768 -0.04259757 0.01110807 -0.08337403 -0.09545899 0.98480224
0.99854135 -0.03548179 -0.03924949 0.01075642 -0.08239747 -0.08770752 1.00665283
0.99809846 -0.04847810 -0.03633126 0.01138259 -0.07623291 -0.07989502 1.00000000
0.99799871 -0.04078269 -0.04715079 0.01059191 -0.08544922 -0.09112549 1.01458740
0.99859666 -0.03903663 -0.03403557 0.01106134 -0.07818603 -0.08154297 1.01202392
0.99783344 -0.04730068 -0.04440124 0.01093363 -0.08496094 -0.09289551 0.97949218
0.99764461 -0.03947515 -0.05518217 0.01008464 -0.09436036 -0.09277344 1.00555419
0.99843034 -0.03428750 -0.04307221 0.01029709 -0.07714844 -0.06646728 1.01562500
0.99804611 -0.04707689 -0.03958443 0.01098104 -0.08197022 -0.09057618 1.01885986
0.99828948 -0.03523235 -0.04551997 0.01023671 -0.08514405 -0.08648682 1.01544189
```

Figura 3.5: Saída da função Tratar Dados.

3.2 Software

Utilizando a plataforma de desenvolvimento de jogos Unity 3D foi preparado um *software* capaz de exibir uma simulação 3D do acidente. O software desenvolvido no trabalho anterior [5] possui funcionalidades como controle de velocidade, pause na simulação e uma divisão de tela em três câmeras (Figura 3.6).

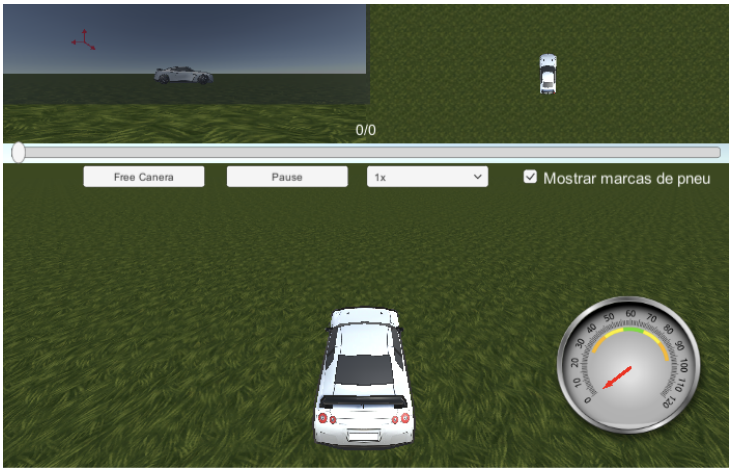


Figura 3.6: Simulador [5].

Os dados obtidos com o GPS foram adicionados ao simulador em forma de mapa em conjunto com a visualização da câmera superior. A Figura 3.7 mostra um exemplo simulado dessa visualização. Para a visualização em mapa foi utilizada uma API do Google Maps, que permite a adição de um mapa do google com marcadores baseados nas localizações obtidas no formato latitude e longitude. Essa API permite fazer requisições

que são definidas de acordo com os parâmetros utilizados. Uma das opções disponíveis é a de adição de marcadores, que vão determinar as localizações obtidas pelo GPS. Foi adicionado um checkbox para controlar a exibição do mapa.

Figura 3.7: Visualização no simulador dos dados de localização obtidos [5].

Capítulo 4

Análise dos Dados Obtidos

Com o objetivo de analisar os dados obtidos com a plataforma descrita e corrigir possíveis erros, alguns ensaios foram realizados e serão descritos neste capítulo. O conhecimento das orientações dos eixos para acelerômetro e giroscópio é fundamental para entendimento dos resultados e está representado na Figura 4.1.

Inicialmente, foram realizados ensaios com a plataforma fora do carro para checar se os dados obtidos com algumas rotações simples eram coerentes. Nesses ensaios controlados a vantagem é a possibilidade de eliminação de vários ruídos obtidos com os testes em campo, decorrentes do estado das vias e outras possíveis condições naturais.

Em seguida, foram realizados três ensaios com a plataforma já acoplada ao veículo, nos quais a plataforma saiu inicialmente do repouso. Esses ensaios em campo buscaram comparar o percurso real, rotações e acelerações realizadas, com o obtido a partir dos quaternions calculados utilizando os dados coletados.

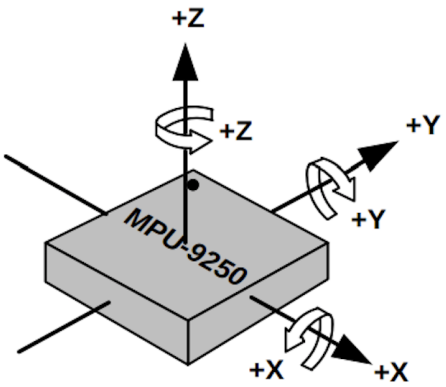


Figura 4.1: Orientação do eixo de sensibilidade e polaridade de rotação para acelerômetro e giroscópio [6].

4.1 Ensaio fora do veículo

Os três testes inicias fora do veículo e que duraram 20 segundos foram:

1. Rotação de 90º em torno do eixo Z e retorno para a posição inicial
2. Rotação de 90º em torno do eixo X e retorno para a posição inicial
3. Rotação de 90º em torno do eixo Y e retorno para a posição inicial

O giroscópio mede velocidade angular que pode ser integrada ao longo do tempo para se obter a orientação angular da plataforma. Porém a integração dos erros de medição do giroscópio leva a uma acumulação de erros no resultado final. O acelerômetro está continuamente sujeito ao campo gravitacional da Terra, o que é conveniente pois oferece uma referência absoluta. Porém as acelerações devido ao movimento vão corromper a direção medida da gravidade.

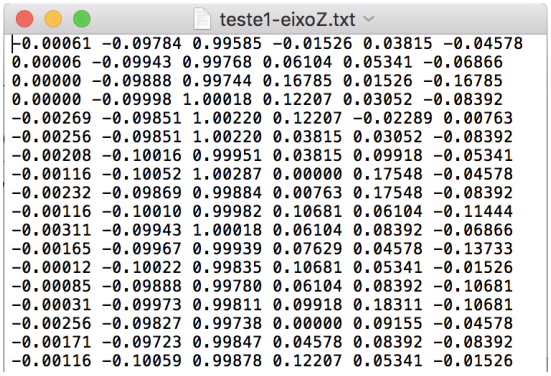
Teste 1

No primeiro teste foi realizada uma rotação em torno do eixo Z. Foram plotados quatro gráficos com os valores de aceleração, giro, ângulo de rotação em graus obtido através de integração e quaternions. Os dados obtidos podem ser observados na Figura 4.2a, onde a ordem em que aparecem é de aceleração e giro nos eixos x, y e z, respectivamente (a_x, a_y, a_z, g_x, g_y, g_z).

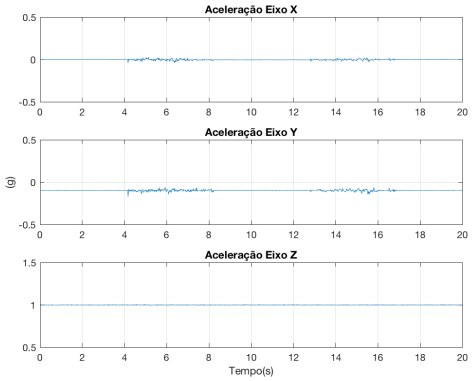
Os gráficos que foram plotados com a utilização do *Matlab* representam: os valores de aceleração (Figura 4.2b) e giro (Figura 4.2c), e ângulo de rotação em graus (Figura 4.2d), obtido a partir da integração dos dados do giroscópio. E os quaternions obtidos com a utilização do filtro do Madgwick podem ser observados na Figura 4.3a.

Por meio da Figura 4.2c podemos observar dois picos no giro obtido no eixo Z, que se refere as duas rotações de 90º realizadas (a primeira rotação de 90º seguida do retorno à posição inicial). A Figura 4.2d, contendo os dados integrados, comprova essa rotação até os 90º e retorno a 0.

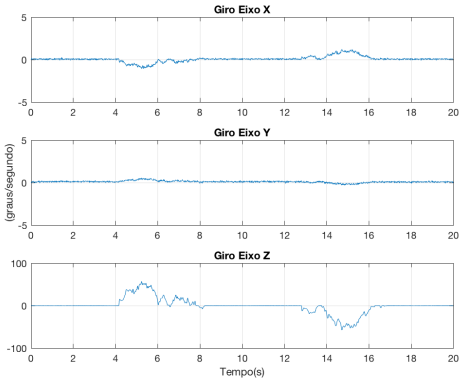
Vamos agora considerar os quaternions formados pelos quatro componentes q_w, q_x, q_y, q_z. A representação dos quaternions (4.3a) mostra inicialmente os valores em uma situação sem rotação: $q_w = 1$, $q_x = 0$, $q_y = 0$, $q_z = 0$. Esses valores mudam rapidamente pois a rotação foi realizada logo em seguida para caber no intervalo de 20 segundos. Podemos observar o valor de q_w diminuindo e q_z aumentando. Os quaternions são unitários, logo a soma dos quadrados é sempre 1. Foi realizada também uma rotação de 180º em torno do eixo Z, que está representada na Figura 4.3b, na qual podemos observar que a componente q_w continua diminuindo até o 0, enquanto q_z aumenta até atingir o valor 1. E logo em seguida acontece o inverso e o quaternion retorma para a posição inicial.



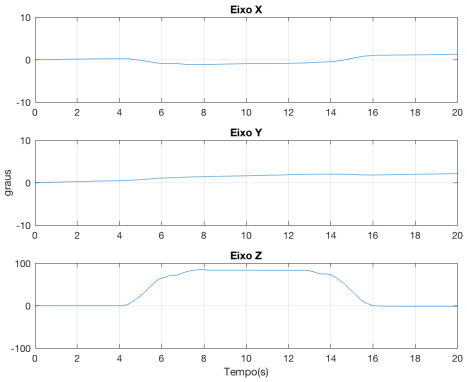
(a) Dados de aceleração e giro



(b) Aceleração

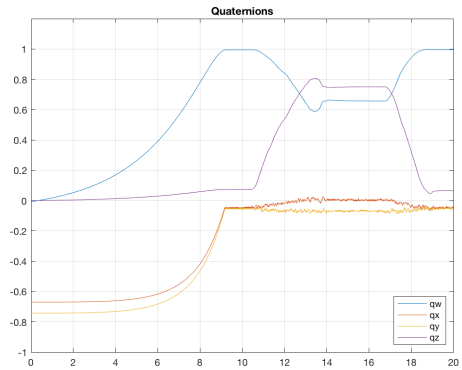


(c) Giroscópio

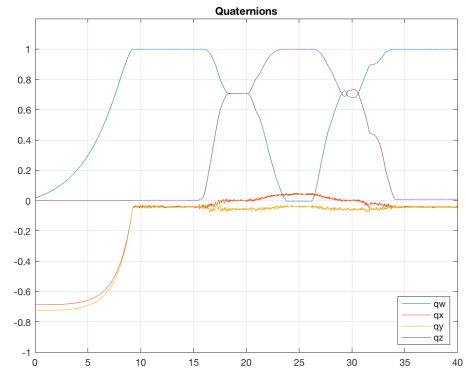


(d) Ângulo de rotação

Figura 4.2: Rotação 90º em torno do eixo Z



(a) Rotação de 90º e retorno à posição inicial



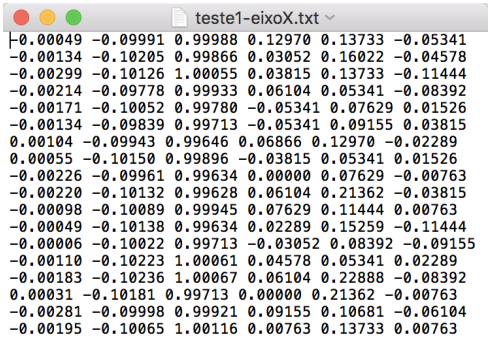
(b) Rotação de 180º e retorno à posição inicial

Figura 4.3: Visualização dos quaternions obtidos com a rotações no eixo Z

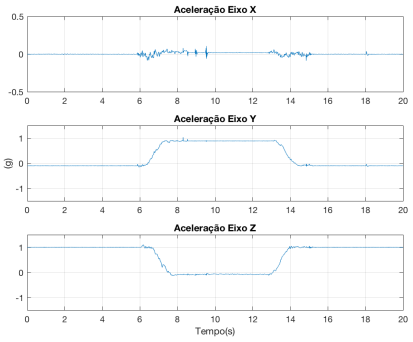
Teste 2

No segundo teste foi realizada uma rotação em torno do eixo X. Os dados obtidos podem ser observados na Figura 4.4a, a ordem em que aparecem é de aceleração eixos x, y e z, e giro, eixos x, y, e z. Foram plotados com a utilização do *Matlab* os valores de aceleração (Figura 4.4b), giro (Figura 4.4c) e ângulo de rotação em graus (Figura 4.4d), obtido a partir da integração dos dados do giroscópio.

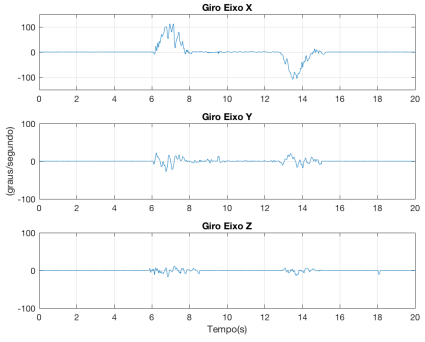
Os quaternions, obtidos com a utilização do filtro do Madgwick, em uma rotação em torno do eixo X plotados na Figura 4.5a, separando as quatro componentes do quaternion q_w, q_x, q_y e q_z, mostram a componente q_w diminuindo e a componente q_x aumentando. Novamente, podemos observar quaternions unitários. E da mesma forma que foi feito no teste anterior, em seguida foi realizada uma rotação de 180º e retorno à posição inicial (Figura 4.5b), no qual se pode ver que a componente q_w diminui até atingir o valor 0 enquanto a componente q_x aumenta até atingir 1. Os testes foram realizados em um intervalo de 20 segundos. Apenas o ensaio com a rotação de 180º durou 40 segundos para ser possível a convergência inicial e a execução dos movimentos necessários.



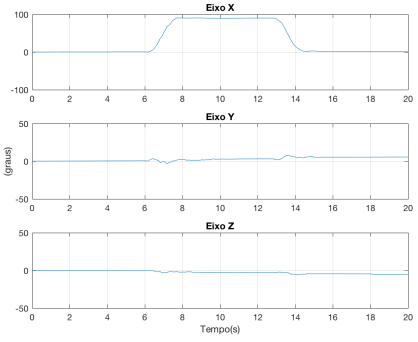
(a) Dados de aceleração e giro



(b) Aceleração

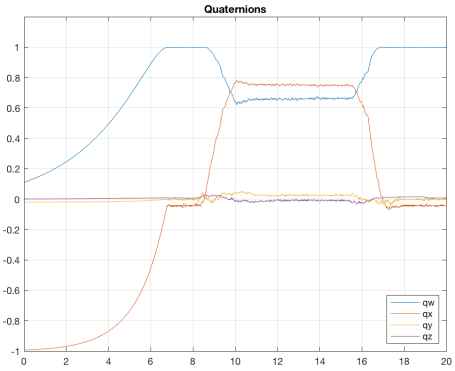


(c) Giroscópio

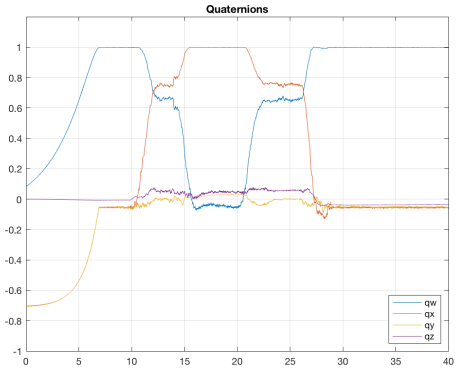


(d) Ângulo de rotação

Figura 4.4: Rotação 90º em torno do eixo X



(a) Rotação de 90º e retorno à posição inicial



(b) Rotação de 180º e retorno à posição inicial

Figura 4.5: Visualização dos quaternions obtidos com a rotações no eixo X

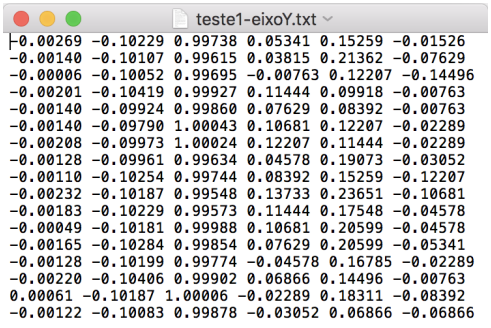
Teste 3

No terceiro teste foi realizada uma rotação em torno do eixo Y. Os dados obtidos podem ser observados na Figura 4.6a, a ordem em que aparecem é de aceleração eixos x, y e z, e giro, eixos x, y, e z. Os gráficos obtidos e as interpretações são semelhantes ao que foi explicado anteriormente para os testes 1 e 2, de rotações em torno dos eixos Z e X.

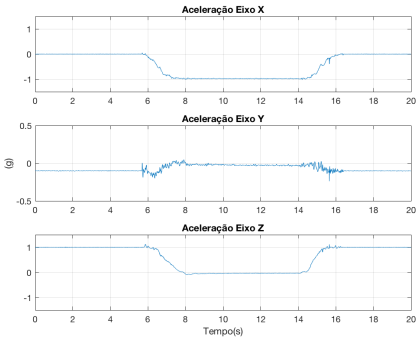
Foram plotados com a utilização do *Matlab* os valores de aceleração (Figura 4.6b), giro (Figura 4.6c) e ângulo de rotação em graus (Figura 4.6d), obtido a partir da integração dos dados do giroscópio. E as Figuras 4.7a e 4.7b contêm os quaternions obtidos com a utilização do filtro do Madgwick. O teste utilizado para plotar os gráficos de aceleração, giro e quaternions (rotação 90º) tiveram um intervalo de 20 segundos. E o que obteve a rotação de 180º, um intervalo de 40 segundos.

4.2 Ensaios no veículo

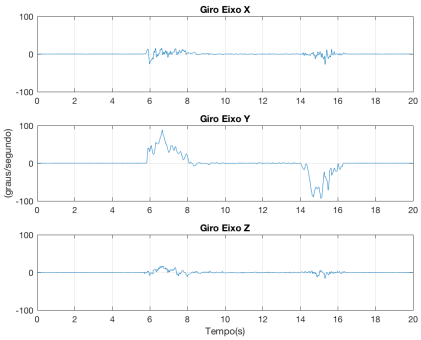
A Figura 4.8 mostra a plataforma posicionada no veículo para os testes realizados em campo. Foram realizados dois testes (Figuras 4.9a e 4.9b) para observar algumas características da plataforma. A duração de cada um dos testes foi de 90 segundos. Para cada um dos testes realizados, o veículo saiu do repouso, os dados foram armazenados com a rotina principal da plataforma. Em seguida foi preciso conectar o protótipo a um computador e extrair os dados salvos na memória. Só após esse processo é realizado um novo teste, para que os dados obtidos não sejam sobrescritos pelos de um novo teste.



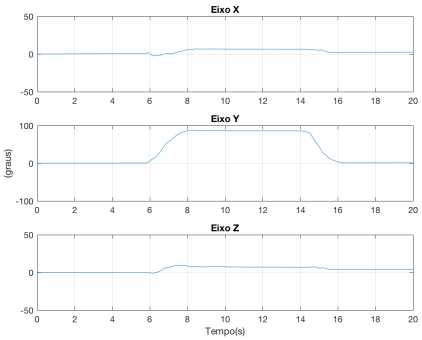
(a) Dados de aceleração e giro



(b) Aceleração

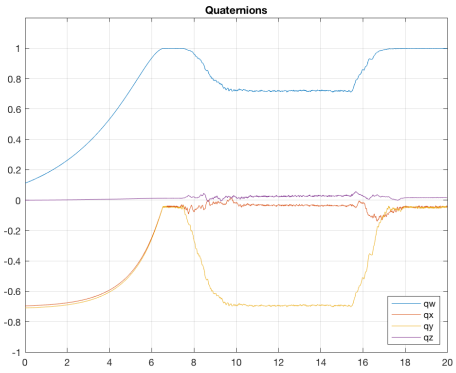


(c) Giroscópio

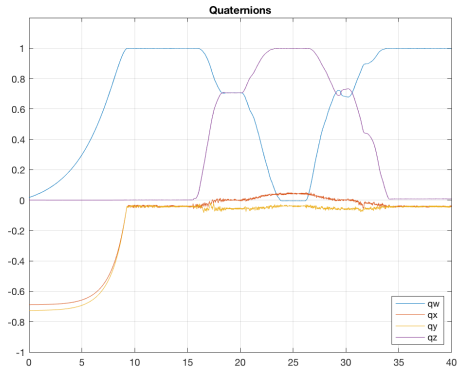


(d) Posição

Figura 4.6: Rotação 90º em torno do eixo Y



(a) Rotação de 90º e retorno à posição inicial



(b) Rotação de 180º e retorno à posição inicial

Figura 4.7: Visualização dos quaternions obtidos com a rotações no eixo Y

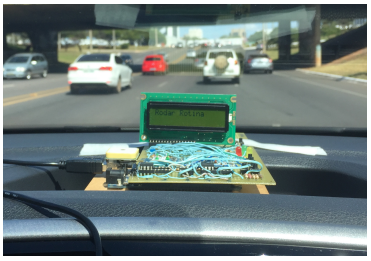
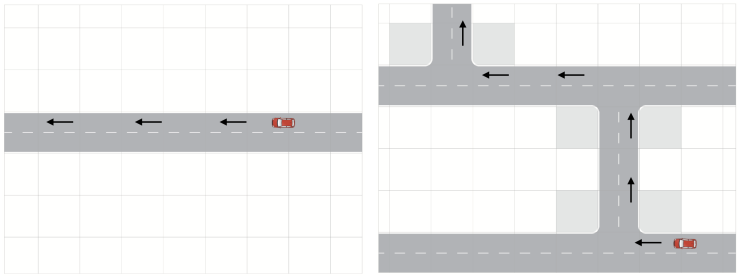


Figura 4.8: Protótipo desenvolvido acoplado ao veículo.



(a) Teste 1 (b) Teste 2

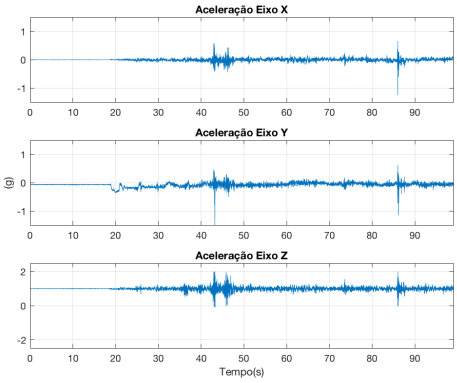
Figura 4.9: Percursos realizados nos testes fora do veículo

Teste 1

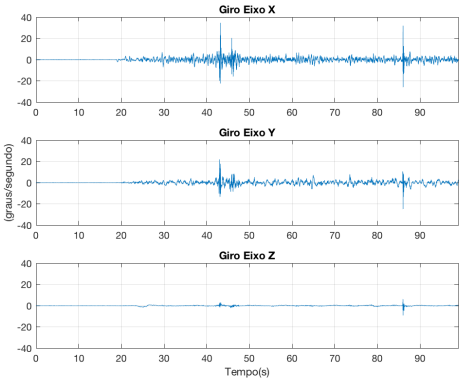
No primeiro teste realizado, o percurso desenvolvido foi apenas em linha reta, sem a execução de rotação em nenhum dos três eixos. O veículo partiu do repouso e foi aumentando a velocidade até atingir os 60km/h, porém aconteceram algumas desacelerações no período. O percurso possuía alguns buracos, o que se refletiu na quantidade de ruídos nos gráficos obtidos de aceleração e giro. O intervalo de tempo de obtenção dos dados foi de 90 segundos.

As Figuras 4.10a e 4.10b contêm os dados obtidos de aceleração nos eixos x, y e z e giro nos eixos x, y e z, respectivamente. A Figura 4.10c contém os dados de ângulo de rotação da plataforma em graus obtidos a partir da integração dos dados do giroscópio. Podemos observar que os valores nos três gráficos se mantém, porém há a presença de bastante ruído, devido às condições da via.

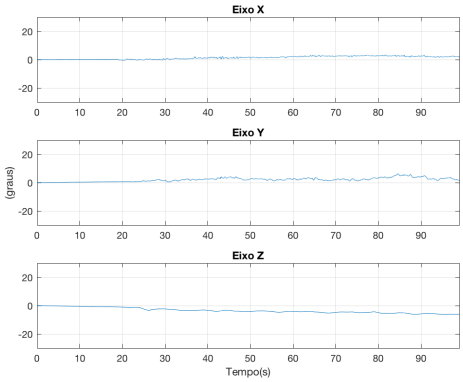
Os dados obtidos pela plataforma foram tratados com o auxílio do filtro de Madgwick, que é responsável pela fusão dos dados dos sensores e geração de quaternions representando as orientações em cada instante. A Figura 4.11 contém os quaternions obtidos. De maneira semelhante a que foi explicada no bloco de testes fora do veículo, podemos observar as componentes q_w, q_x, q_y e q_z mantendo os valores, o que é o esperado já que não há nenhuma rotação nesse percurso.



(a) Aceleração



(b) Giroscópio



(c) Posição

Figura 4.10: Dados obtidos com o teste 1

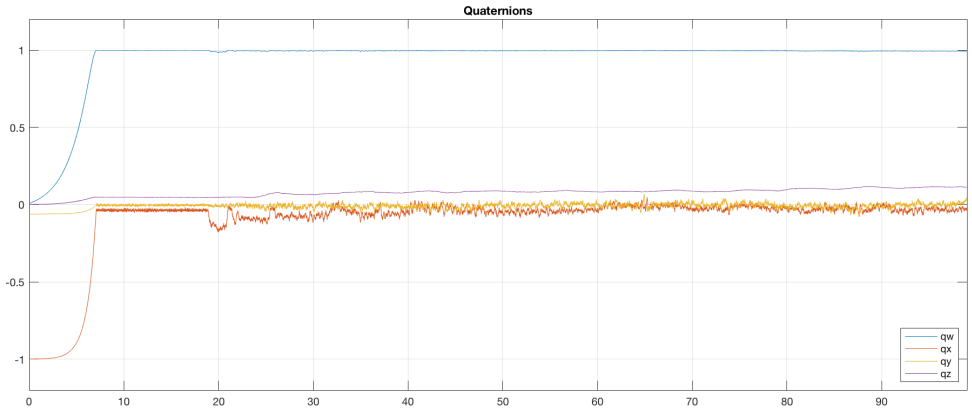
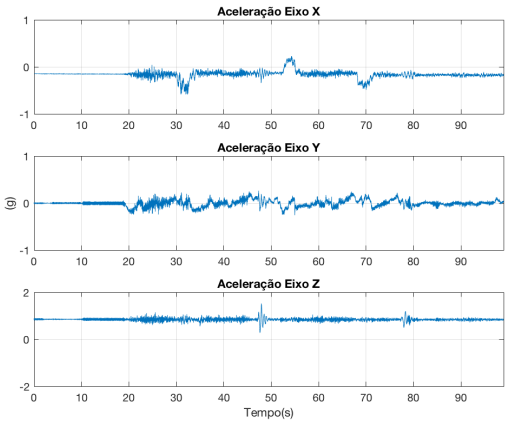


Figura 4.11: Quaternions obtidos com o Teste 1 realizado no veículo.

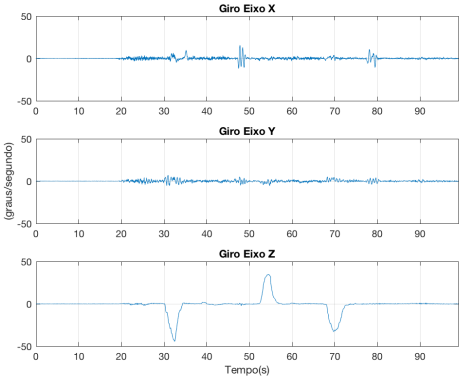
Teste 2

No segundo teste realizado (Figura 4.9b), foram realizadas três rotações, que podem ser observadas na Figura 4.12b, que mostra três picos nos valores de giro obtidos. Ao integrar esses dados (Figura 4.12c), a rotação em graus vai até os -90 graus, retorna a 0 e em seguida atinge aproximadamente -100. Os valores de aceleração (Figura 4.12a) possuem muito ruído.

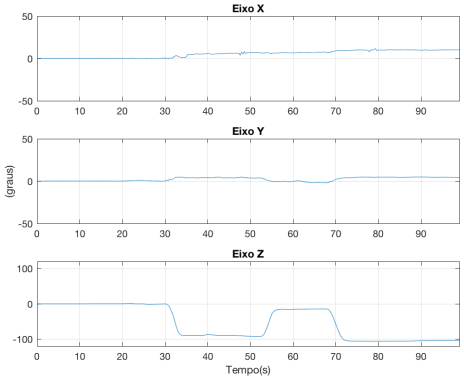
Os quaternions obtidos estão representados na Figura 4.13. Nos primeiros 30 segundos do percurso, o veículo não realizou nenhuma rotação, e as componentes do quaternion mantém o seu valor inicial $q_w = 1$, $q_x = 0$, $q_y = 0$ e $q_z = 0$. A primeira rotação de 90º no eixo Z (curva à direita), pode ser observada no instante 30, em que a componente q_w começa a diminuir à medida que a componente q_z aumenta.



(a) Aceleração



(b) Giroscópio



(c) Posição

Figura 4.12: Dados obtidos com o teste 2

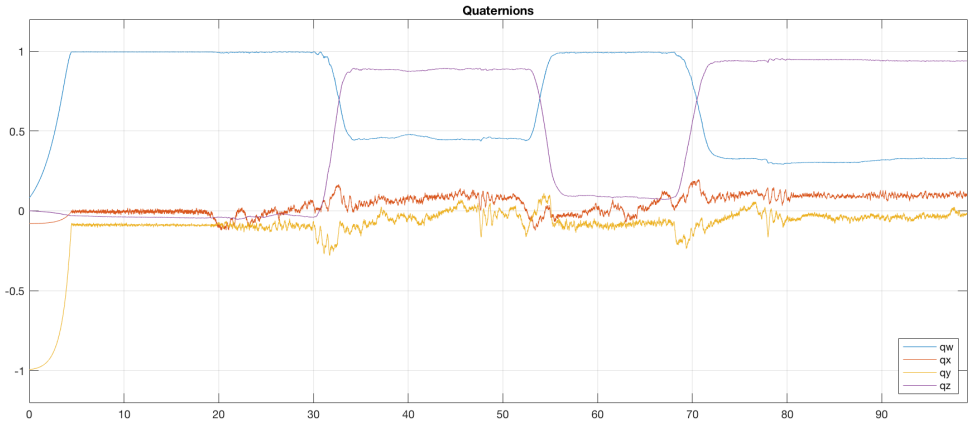


Figura 4.13: Quaternions obtidos com o Teste 2 realizado no veículo.

4.3 Simulações no Unity

As Figuras 4.14 a 4.16 são capturas de tela da simulação rodando no Unity. A Figura 4.14 representa o primeiro percurso realizado no ensaio em campo no qual foi desenvolvido um movimento apenas em linha reta. Podemos observar que apenas 4 localizações diferentes foram obtidas pelo GPS, o que acontece devido a acurácia do receptor GPS, porém como não houveram rotações, o resultado obtido foi satisfatório.

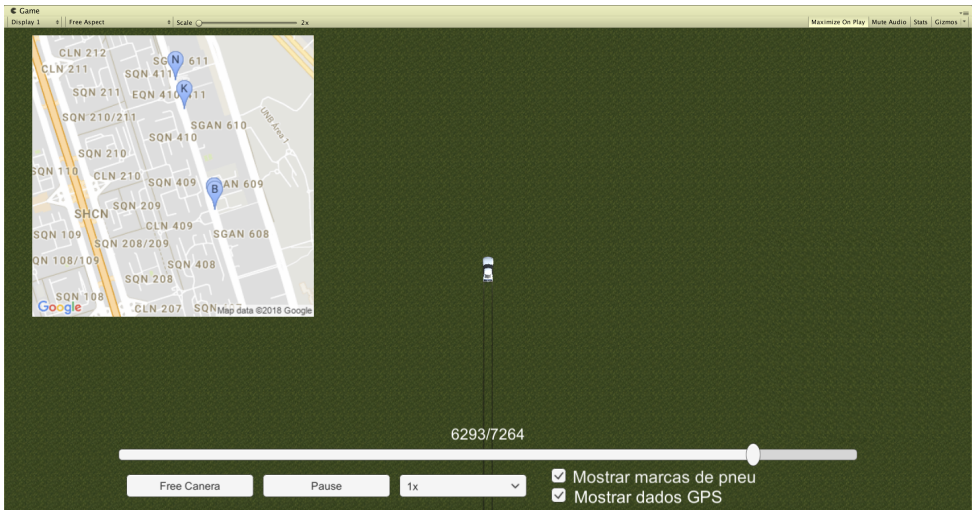


Figura 4.14: Simulação no Unity obtida para o primeiro percurso dos ensaios em campo.

A Figura 4.15 representa a primeira rotação de 90º no eixo Z (uma curva a direita) realizada no segundo percurso dos ensaios em campo. Podemos observar que a orientação angular final do veículo ultrapassa os 90º, logo, apresenta incoerências em relação ao movimento real.

A Figura 4.16 representa um instante mais próximo ao final da simulação, quando uma rotação de -90º no eixo Z (curva a esquerda) é realizada. Nessa instante, já houve uma maior acumulação de erros e o resultado não está de acordo com os gráficos obtidos para os quaternions.

O resultado visto pelo Unity ainda necessita de alguns ajustes para ser mais condizente com os resultados obtidos, já que o *software* é o responsável pela visualização da dinâmica do acidente.

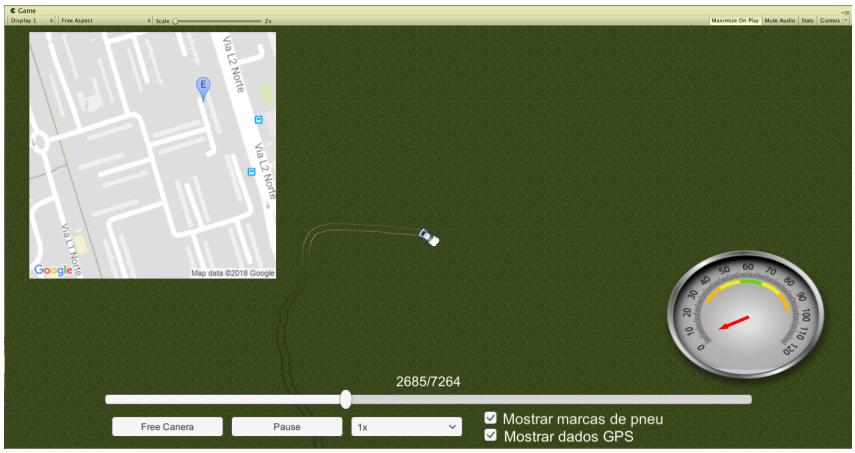


Figura 4.15: Simulação no Unity obtida para o segundos percursos dos ensaios em campo.

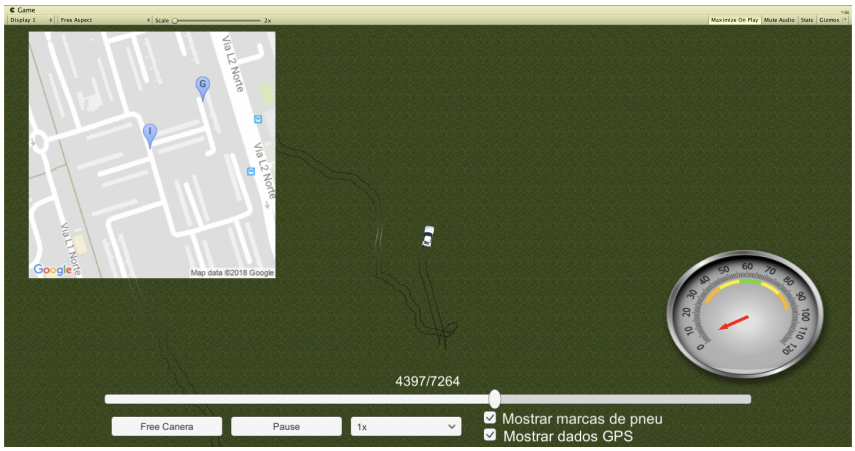


Figura 4.16: Simulação no Unity obtida para o segundos percursos dos ensaios em campo.

Capítulo 5

Conclusões

Este trabalho apresentou um sistema desenvolvido para auxiliar a obtenção de dados de um veículo e elaboração de um laudo técnico em perícias de acidentes de trânsito. O sistema é composto de um *hardware* que é acoplado ao veículo e fornece informações como aceleração e giro, e um *software* que é utilizado após o acidente e possibilita a reconstrução da dinâmica do evento.

5.1 Resultados Obtidos

Foram realizados alguns ensaios que buscaram testar as funcionalidades desenvolvidas e a utilização da representação por quaternions para a posição. Esses resultados foram avaliados por meio de gráficos plotados e apresentados no Capítulo 4. Os gráficos analisados foram os de aceleração, giro, orientação angular e quaternions.

Os gráficos de aceleração, principalmente para os ensaios em campo, apresentaram muito ruído, o que comprova a alta quantidade de ruídos nos dados obtidos pelo acelerômetro. Já os gráficos de orientação angular, obtidos a partir da integração numérica dos valores de giro, apresentaram imprecisões devido ao acúmulo de erros de medições do giroscópio no resultado final.

Os resultados obtidos com a utilização do filtro de orientação de Madgwick foram plotados em gráficos contendo as componentes q_w, q_x, q_y e q_z dos quaternions gerados. Os gráficos concordaram com o comportamento esperado para cada um dos percursos, logo, os resultados apresentados foram satisfatórios.

Foi acrescentado ao *software* elaborado com o Unity 3D os dados de GPS obtidos. Por meio de uma API do Google foi possível inserir uma visualização em mapa com as localizações, em latitude e longitude, destacadas, para facilitar a visão geral do percurso.

5.2 Trabalhos Futuros

Possíveis trabalhos futuros poderiam se beneficiar da utilização de circuitos impressos, o que reduziria o tamanho do *hardware* e facilitaria o acoplamento ao veículo para a utilização.

Além disso, o filtro de Madgwick possui duas versões, uma que utiliza os dados obtidos pelo magnetômetro e uma que realiza a fusão apenas dos dados de aceleração e giro, e, neste trabalho, foi utilizada a segunda opção devido a dificuldade de correta calibração do magnetômetro. O magnetômetro deve ser calibrado sempre que ocorre uma mudança em sua configuração de utilização, pois componentes eletrônicos e ferromagnéticos nas proximidades interferem nesse processo.

O simulador desenvolvido necessita de alguns ajustes para a obtenção de um percurso mais próximo ao realizado. Apesar dos quaternions obtidos estarem de acordo com as rotações realizadas, a visualização no simulador ainda não reproduz corretamente essas rotações.

Referências

[1] Ninjas, Practical: *How accelerometer works? | working of accelerometer in a smartphone | mems inside accelerometer.* https://www.youtube.com/watch?v=T_iXLNkkjFo, acesso em 2018-27-05. 10, 19

[2] ElectronicDesign: *Mems sensors help safeguard passengers.* http://www.electronicdesign.com/automotive/mems-sensors-help-safeguard-passengers, acesso em 2018-02-06. 10, 19

[3] GISGeography: *Trilateration vs triangulation – how gps receivers work.* https://gisgeography.com/trilateration-triangulation-gps/, acesso em 2018-27-05. 10, 21

[4] Madgwick, Sebastian O. H.: *An efficient orientation filter for inetial and inertial/-magnetic sensor arrays.* Technical Report, University of Bristol, UK, abril 2010. 10, 26, 28

[5] Serra, Daniel; Romualdo, João Victor: *Caixa preta para veículos automotivos.* Trabalho de conclusão de curso (bacharelado em engenharia elétrica), Universidade de Brasília, Brasília, 2017. 10, 16, 36, 37

[6] InvenSense Inc.: *MPU-9250 Product Specification*, junho 2016. Revision: 1.1. 10, 17, 38

[7] Dados Abertos, Portal Brasileiro de: *Indicadores sobre sistema de informações de mortalidade - sim.* http://dados.gov.br/dataset/sistema-de-informacoes-de-mortalidade-sim, acesso em 2018-04-06. 14

[8] Líder, Seguradora: *Seguro dpvat - sobre o seguro dpvat.* https://www.seguradoralider.com.br/Seguro-DPVAT/Sobre-o-Seguro-DPVAT, acesso em 2018-04-06. 14

[9] Seguras, Vias: *Estatísticas nacionais de acidentes de trânsito.* http://vias-seguras.com/os_acidentes/estatisticas/estatisticas_nacionais/estatisticas_do_ministerio_da_saude, acesso em 2018-04-06. 14

[10] TrânsitoBr: *O portal do trânsito brasileiro - acidentes números.* http://www.transitobr.com.br/index2.php?id_conteudo=9, acesso em 2018-04-06. 14

[11] Viana, Rubens Moreira: *Perícia física de acidente de trânsito*, 2009. 14

[12] LIMA, Vinícius De Oliveira: *Proposta de plataforma inercial para auxiliar na perícia de acidentes de trânsito.* Tese de Mestrado, Universidade de Brasília, Departamento de Engenharia Elétrica, Brasília, 2016. 15

[13] RAMOS, Hudson Pereira; LUCENA, Vanessa Oliveira: *Proposta de plataforma inercial e simulador 3d para periciar acidentes de trânsito.* Trabalho de conclusão de curso (bacharelado em engenharia mecatrônica), Universidade de Brasília, Brasília, 2017. 16

[14] AsahiKASEI: *AK8963 - 3-axis Electronic Compass*, outubro 2013. 18

[15] MEMS e Nanotechnology Exchange: *What is mems technology?* `https://www.mems-exchange.org/MEMS/what-is.html`, acesso em 2018-24-05. 18

[16] Tsui, James Bao Yen: *Fundamentals of Global Positioning System Receivers: A Software Approach.* John Wiley Sons, Inc., 2000. 20, 21

[17] ublox: *Neo-6 u-blox 6 gps modules data sheet*, 2011. 21

[18] Inc., Microchip Technology: *1mbit spi serial sram with sdi and sqi interface*, 2015. 22

[19] Inc., Microchip Technology: *1024k i2ctm cmos serial eeprom*, 2015. 22

[20] Arduino: *Documentation arduino mega 2560.* `https://store.arduino.cc/usa/arduino-mega-2560-rev3`, acesso em 2018-01-06. 22

[21] Hamilton, William Rowan: *On quaternions, or on a new system of imaginaries in algebra.* The London, Edinburgh and Dublin Philosophical Magazine and Journal of Science, xxv–xxxvi(3), 1844-1850. 25

[22] Hanson, Andrew J.: *Visualizing Quaternions Series in Interactive 3D Technology*, volume 1993. Elsevier, 2005. 25

[23] Hart, Mikal: *Tinygps++.* `http://arduiniana.org/libraries/tinygpsplus/`, acesso em 2018-02-06. 34

Apêndice A

Códigos Utilizados

A.1 Código Principal

```
#include <Wire.h>
#include <SPI.h>
#include <MPU9250_t.h>
#include <EEPROM_24XX1025.h>
#include <LiquidCrystal.h>
#include <I2C16.h>
#include <SRAM.h>
#include <TinyGPS++.h>
#include <Utils.h>
#include <TimeLib.h>

/*---------------------- MPU9250 ---------------------*/
MPU9250 mpu;

/*---------------------- EEPROM ----------------------*/
EEPROM_24XX1025 eeprom(0,0);

/*----------------------- SRAM -----------------------*/
SRAM sram;

/*----------------------- GPS ------------------------*/
TinyGPSPlus gps;
const int UTC_offset = -3;

/*----------------------- LCD ------------------------*/
// Function: LiquidCrystal lcd(rs, enable, d4, d5, d6, d7)
LiquidCrystal lcd(36, 37, 26, 27, 28, 29);
const uint8_t LCD_RW = 35;
const uint8_t LCD_BL = 34;

/*---------------- PARAMETROS FILTRO ----------------*/
float ax, ay, az, gx, gy, gz, mx, my, mz;
float q[4] = {1.0f, 0.0f, 0.0f, 0.0f};
float sum = 0.0f;
float gyroMeasError = 3.14159265358979 * (40.0f / 180.0f);
float beta = sqrt(3.0f / 4.0f) * gyroMeasError;

#define printRawAccelGyroData false
```

```
39 #define debugSetup false
40 #define debug false
41
42 typedef struct {
43   float lat;
44   float lng;
45   float veloc;
46   uint8_t hora;
47   uint8_t minuto;
48   uint8_t segundo;
49   uint8_t dia;
50   uint8_t mes;
51   uint16_t ano;
52 } GPSdata;
53 GPSdata dataGPS[16];
54 int indexGPS = 0;
55
56 /*---------------------- SETUP ----------------------*/
57 void setup() {
58   // SRAM
59   pinMode(CSpin, OUTPUT);
60   digitalWrite(CSpin, HIGH);
61
62   // LCD
63   pinMode(LCD_RW, OUTPUT);
64   pinMode(LCD_BL, OUTPUT);
65   digitalWrite(LCD_RW, LOW);
66   digitalWrite(LCD_BL, LOW);
67   lcd.begin(16, 2);
68   lcd.write("Iniciando caixa-preta...");
69
70   // Comunicacao
71   Serial.begin(115200);
72   Serial3.begin(9600);
73   Wire.begin();
74   SPI.begin();
75
76   // LEDs
77   pinMode(LED_VM, OUTPUT);
78   pinMode(LED_AM, OUTPUT);
```

```
79    pinMode(LED_VD, OUTPUT);
80    digitalWrite(LED_VM, LOW);
81    digitalWrite(LED_AM, LOW);
82    digitalWrite(LED_VD, LOW);
83
84    // Rotina inicializacao RST - pisca cada um dos LEDs
85    blink_led(LED_VM); blink_led(LED_AM); blink_led(LED_VD);
86
87    lcd.clear();
88    setup_mpu();
89  }
90
91  /*---------------------- LOOP -----------------------*/
92  void loop() {
93    int chave;
94    static int option = 0;
95
96    lcd.clear();
97    lcd.setCursor(0,0);
98
99    switch(option) {
100     case 0: lcd.print(F("Testar MPU       1")); break;
101     case 1: lcd.print(F("Testar GPS       2")); break;
102     case 2: lcd.print(F("Rodar Rotina     3")); break;
103     case 3: lcd.print(F("Ler dados EEPROM 4")); break;
104     case 4: lcd.print(F("Ler dados GPS    5")); break;
105     case 5: lcd.print(F("Tratar dados     6")); break;
106     default: lcd.clear(); break;
107   }
108
109   chave = sw_wait_read();
110   if(chave == SW_BAIXO) {
111     option++;
112     if(option > 5) option &= 0x00;
113   }
114   else if(chave == SW_CIMA) {
115     option--;
116     if(option == -1) option = 5;
117   }
118   else if(chave == SW_SEL) {
```

```
119     lcd.setCursor(1, 1);
120     switch(option) {
121     case 0:lcd.print("Testar MPU 1"); test_mpu(); break;
122     case 1:lcd.print("Testar GPS 2"); test_gps(); break;
123     case 2:lcd.print("Rodar Rotina 3"); routine(); break;
124     case 3:lcd.print("Ler dados EEPROM 4");transEEPROM();break;
125     case 4:lcd.print("Ler dados GPS 5");readGPSdataEEPROM();break;
126     case 5:lcd.print("Tratar dados 6"); treat_data(); break;
127     default: lcd.clear(); break;
128    }
129  }
130 }
131
132 /*------------------- SETUP MPU --------------------*/
133 void setup_mpu() {
134   if(debugSetup) Serial.println("Testar comunicacao com o MPU");
135   // Testar comunicacao com o MPU - leitura do registrador
        WHO_AM_I
136   uint8_t c = mpu.readByte(MPU9250_ADDRESS, WHO_AM_I_MPU9250);
137
138   if(c == 0x73) {
139     if(debugSetup) {
140       Serial.println("Comunicacao ok");
141       Serial.println("SelfTest:");
142     }
143
144     mpu.MPU9250SelfTest(mpu.selfTest);
145     if(debugSetup) {
146       Serial.println("Aceleracao e giro self-test:");
147       Serial.print("Eixo x:"); Serial.print(mpu.selfTest[0], 1);
148       Serial.print("Eixo y:"); Serial.print(mpu.selfTest[1], 1);
149       Serial.print("Eixo z:"); Serial.print(mpu.selfTest[2], 1);
150       Serial.print("Eixo x:"); Serial.print(mpu.selfTest[3], 1);
151       Serial.print("Eixo y:"); Serial.print(mpu.selfTest[4], 1);
152       Serial.print("Eixo z:"); Serial.print(mpu.selfTest[5], 1);
153     }
154     mpu.getAres();
155     mpu.getGres();
156     mpu.getMres();
157
```

```
158     if(debugSetup) Serial.println("Calibrar...");
159     mpu.calibrateMPU9250(mpu.gyroBias, mpu.accelBias);
160
161     if(debugSetup) {
162       Serial.println("Acelerometro bias:");
163       Serial.print("Eixo x: "); Serial.println(mpu.accelBias[0]);
164       Serial.print("Eixo y: "); Serial.println(mpu.accelBias[1]);
165       Serial.print("Eixo z: "); Serial.println(mpu.accelBias[2]);
166       Serial.println("Giroscopio bias:");
167       Serial.print("Eixo x: "); Serial.println(mpu.gyroBias[0]);
168       Serial.println("Eixo y: "); Serial.print(mpu.gyroBias[1]);
169       Serial.print("Eixo z: "); Serial.println(mpu.gyroBias[2]);
170     }
171
172     if(debugSetup) Serial.println("Inicializar...");
173     mpu.initMPU9250();
174
175     // Magnetometro
176     c = mpu.readByte(AK8963_ADDRESS, WHO_AM_I_AK8963);
177
178     if(c != 0x48) {
179      if(debugSetup)Serial.println("Falha na comunicacao AK8963");
180      Serial.flush();
181      abort();
182     }
183
184     if(debugSetup) Serial.println("Inicializar AK8963...");
185     mpu.initAK8963(mpu.factoryMagCalibration);
186   } else {
187     Serial.println("Falha na comunicacao");
188     while(1);
189   }
190 }
191
192 /*------------------- ROUTINE --------------------*/
193 void routine() {
194   int16_t dados[288]; // Dados de 32 leituras de 18 bytes = 576
195   uint8_t endRoutine = 0, evento = 0;
196   int32_t addrSram = 0;
197   int32_t addrEvento, addrDataStart = 0;
```

```
198   uint32_t t_f;
199   uint16_t writes = 0;
200   int16_t dadosGPS[20]; // Dados GPS - 8 amostras de 40 bytes
201   int contLeituras = 0, contGPS = 0;
202
203   loading();
204   lcd.clear();
205
206   while(!endRoutine) {
207         // Para encerrar o loop pressione qualquer botao
208     if(analogRead(ADC0) < 1000) evento = 1;
209
210     do {
211       delayMicroseconds(10);
212     } while((micros() - t_f) < 10000);
213
214     // Le um conj de dados do MPU - ax ay az gx gy gz mx my mz
215     mpu.readMotionSensor((dados + (contLeituras*9 + 0)), (dados +
          (contLeituras*9 + 1)), (dados + (contLeituras*9 + 2)), (
         dados + (contLeituras*9 + 3)), (dados + (contLeituras*9 +
         4)), (dados + (contLeituras*9 + 5)), (dados + (contLeituras
         *9 + 6)), (dados + (contLeituras*9 + 7)), (dados + (
         contLeituras*9 + 8)));
216
217     if(contGPS == 14) {
218       getGPSdata(indexGPS);
219       indexGPS++;
220       if(indexGPS == 16) indexGPS = 0;
221       contGPS = 0;
222     }
223     contLeituras++;
224     t_f = micros();
225
226     // Aqui sao definidos os parametros para indicar a batida,
          por enquanto o evento e uma certa quantidade de dados
227     if((addrSram + contLeituras*18) == 65664) {
228       evento = 1;
229     }
230
231
```

```
232     // Quando contLeituras chegar a 32, armazenar na SRAM
233     if(contLeituras == 32) {
234       sram.writeSeq(addrSram, dados, 288);
235       contGPS++;
236 
237       if(evento) {
238         endRoutine = 1;
239         addrEvento = addrSram + contLeituras*18;
240 
241         if(debug) {
242           Serial.print("End evento: "); Serial.print(addrEvento);
243         }
244       }
245       addrSram += sizeof(dados);
246 
247       if(addrSram >= 131071) addrSram -= 131071;
248       contLeituras = 0;
249 
250       lcd.clear(); lcd.print(addrSram);
251       lcd.setCursor(0, 1); lcd.print(" escritas ");
252       lcd.setCursor(12, 1); lcd.print(writes + 1);
253       writes++;
254     }
255   }
256 
257   // Preencher o restante da memoria no pos evento
258   for(int i = 0; i < 113; i++) {
259     for(int j = 0; j < 32; j++) {
260       do {
261         delayMicroseconds(10);
262       } while((micros() - t_f) < 10000);
263 
264       mpu.readMotionSensor((dados + (contLeituras*9 + 0)), (dados
              + (contLeituras*9 + 1)), (dados + (contLeituras*9 + 2)),
              (dados + (contLeituras*9 + 3)), (dados + (contLeituras*9
              + 4)), (dados + (contLeituras*9 + 5)), (dados + (
             contLeituras*9 + 6)), (dados + (contLeituras*9 + 7)), (
             dados + (contLeituras*9 + 8)));
265 
266 
```

```
267         if(contGPS == 14) {
268           getGPSdata(indexGPS);
269           indexGPS++;
270           if(indexGPS == 16) indexGPS = 0;
271           contGPS = 0;
272         }
273         contLeituras++;
274         t_f = micros();
275 
276         if(contLeituras == 32) {
277           sram.writeSeq(addrSram, dados, 288);
278           addrSram += sizeof(dados);
279           if(addrSram >= 131071) addrSram -= 131071;
280           contGPS++;
281 
282           lcd.clear(); lcd.print(addrSram);
283           lcd.setCursor(0, 1); lcd.print(" escritas ");
284           lcd.setCursor(12, 1); lcd.print(writes + 1);
285           writes++;
286           contLeituras = 0;
287         }
288       }
289     }
290 
291     // Dados GPS - 8 amostras de 40 bytes
292     for(int g = 0; g < 20; g++) dadosGPS[g] = 0;
293     for(int g = 0; g < 8; g++) {
294       sram.writeSeq(addrSram, dadosGPS, 20);
295       addrSram += sizeof(dadosGPS);
296     }
297 
298     // Definir endereco do inicio dos dados
299     uint32_t dataPreEvent = 65664; // 32*18*114bytes pre evento
300     addrDataStart = (addrEvento - dataPreEvent);
301     if(addrDataStart < 0) addrDataStart += 131071;
302 
303     saveDataToEEPROM(addrDataStart);
304     return;
305 }
306 
```

```
307 /*---------------- EEPROM ----------------*/
308 void saveDataToEEPROM(uint32_t addrDataStart) {
309   int16_t dataSRAMtoEEPROM[64], dataSRAMtoEEPROM1[32],
        gps_dataSRAMtoEEPROM[40];
310   uint32_t cont, cont1, addrEscritaEEPROM, addrData = 0;
311
312   lcd.clear();
313   lcd.setCursor(0, 1); lcd.write("Save data to EEPROM");
314   delay(50);
315   loading();
316   lcd.clear();
317
318   // Escrever na EEPROM de 128 em 128 bytes
319   for(cont = 0; cont < 1021; cont++) {
320    addrData = (addrDataStart + cont*128);
321    if(addrData > 131071) addrData -= 131071;
322    sram.readSeq(addrData, dataSRAMtoEEPROM, sizeof(
        dataSRAMtoEEPROM));
323    eeprom.write(cont*128, dataSRAMtoEEPROM, sizeof(
        dataSRAMtoEEPROM));
324   }
325
326   addrData = addrDataStart + cont*128;
327   if(addrData > 131071) addrData -= 131071;
328   sram.readSeq(addrData, dataSRAMtoEEPROM1, 64);
329   eeprom.write(cont*128, dataSRAMtoEEPROM1, sizeof(
        dataSRAMtoEEPROM1));
330
331   // Ultimos dados - GPS - 320 bytes
332   addrEscritaEEPROM = cont*128 + 64;
333   saveGPSdataToEEPROM(addrEscritaEEPROM);
334   Serial.println("Dados salvos na EEPROM");
335 }
336
337 void transEEPROM() {
338   uint8_t leitura[18];
339   uint32_t j;
340
341   loading();
342   for(j = 0; j < 130752; j+=18) {
```

```
343       eeprom.read(j, leitura, sizeof(leitura));
344       for(uint8_t i = 0; i < 18; i += 2) {
345         Serial.print(((int16_t)leitura[i+1] << 8 | leitura[i]));
346         Serial.print("\t");
347       }
348       Serial.println();
349     }
350     readGPSdataEEPROM();
351   }
352
353   /*---------------- GPS ----------------*/
354   void saveGPSdataToEEPROM(uint32_t addr) {
355     int16_t zeros[8];
356     for(int i = 0; i < 16; i++) {
357       eeprom.setPosition(addr); eeprom.writeFloat(dataGPS[i].lat);
358       addr += sizeof(float); eeprom.setPosition(addr);
359       eeprom.writeFloat(dataGPS[i].lng);
360       addr += sizeof(float); eeprom.setPosition(addr);
361       eeprom.writeFloat(dataGPS[i].veloc);
362       addr += sizeof(float); eeprom.setPosition(addr);
363       eeprom.writeByte(dataGPS[i].hora);
364       addr += sizeof(byte); eeprom.setPosition(addr);
365       eeprom.writeByte(dataGPS[i].minuto);
366       addr += sizeof(byte); eeprom.setPosition(addr);
367       eeprom.writeByte(dataGPS[i].segundo);
368       addr += sizeof(byte); eeprom.setPosition(addr);
369       eeprom.writeByte(dataGPS[i].dia);
370       addr += sizeof(byte); eeprom.setPosition(addr);
371       eeprom.writeByte(dataGPS[i].mes);
372       addr += sizeof(byte); eeprom.setPosition(addr);
373       eeprom.writeInt(dataGPS[i].ano);
374       addr += sizeof(int); eeprom.setPosition(addr);
375     }
376     for(int j = 0; j < 8; j++) zeros[j] = 0;
377     eeprom.write(addr, zeros, sizeof(zeros));
378   }
379
380   void readGPSdataEEPROM() {
381     float latitude, longitude, velocidade;
382     byte dia, mes, hora, minuto, segundo;
```

```
383   int ano;
384   uint32_t addr = 130752;
385   eeprom.setPosition(addr);
386   for(int i = 0; i < 16; i++) {
387     latitude = eeprom.readFloat();
388     addr += sizeof(float); eeprom.setPosition(addr);
389     longitude = eeprom.readFloat();
390     addr += sizeof(float); eeprom.setPosition(addr);
391     velocidade = eeprom.readFloat();
392     addr += sizeof(float); eeprom.setPosition(addr);
393     hora = eeprom.readByte();
394     addr += sizeof(byte); eeprom.setPosition(addr);
395     minuto = eeprom.readByte();
396     addr += sizeof(byte); eeprom.setPosition(addr);
397     segundo = eeprom.readByte();
398     addr += sizeof(byte); eeprom.setPosition(addr);
399     dia = eeprom.readByte();
400     addr += sizeof(byte); eeprom.setPosition(addr);
401     mes = eeprom.readByte();
402     addr += sizeof(byte); eeprom.setPosition(addr);
403     ano = eeprom.readInt();
404     addr += sizeof(int); eeprom.setPosition(addr);
405
406     setTime(hora, minuto, segundo, dia, mes, ano);
407     adjustTime(UTC_offset * SECS_PER_HOUR);
408
409     Serial.print("Localizacao: ");
410     Serial.print(latitude, 6); Serial.print(" ");
411     Serial.print(longitude, 6); Serial.println(" ");
412     Serial.print("Velocidade: ");
413     Serial.println(velocidade, 2); Serial.println("km/h");
414     Serial.print("Data: "); Serial.print(day());
415     Serial.print("/"); Serial.print(month());
416     Serial.print("/"); Serial.println(year());
417     Serial.print("Hora: "); Serial.print(hour());
418     Serial.print(":"); Serial.print(minute());
419     Serial.print(":"); Serial.println(second());
420   }
421 }
422
```

```
423 void getGPSdata(int index) {
424   bool ready = false;
425 
426   while(!ready) {
427     if(Serial3.available()) {
428       char c = Serial3.read();
429       if(gps.encode(c)) {
430         if(gps.time.isValid() && gps.location.isValid() && gps.
              speed.isValid() && gps.date.isValid()) {
431           ready = true;
432           dataGPS[index].lat = gps.location.lat();
433           dataGPS[index].lng = gps.location.lng();
434           dataGPS[index].veloc = gps.speed.kmph();
435           dataGPS[index].hora = gps.time.hour();
436           dataGPS[index].minuto = gps.time.minute();
437           dataGPS[index].segundo = gps.time.second();
438           dataGPS[index].dia = gps.date.day();
439           dataGPS[index].mes = gps.date.month();
440           dataGPS[index].ano = gps.date.year();
441         }
442       }
443     }
444   }
445 }
446 
447 void test_gps() {
448   bool ready = false;
449   loading();
450 
451   while(1) {
452     if(analogRead(ADC0) < 1000) {
453       lcd.clear();
454       return;
455     }
456     while(Serial3.available()) {
457       char c = Serial3.read();
458 
459       if(gps.encode(c)) {
460         if(gps.location.isValid() && gps.date.isValid() && gps.
              time.isValid() && gps.speed.isValid()) {
```

```
461             Serial.print(F("Localizacao: "));
462             Serial.println(gps.location.lat(), 6);
463             Serial.println(gps.location.lng(), 6);
464             Serial.print(F("Velocidade: "));
465             Serial.print(gps.speed.kmph()); Serial.println("km/h");
466             setTime(gps.time.hour(), gps.time.minute(),
467             gps.time.second(), gps.date.day(), gps.date.month(),
468             gps.date.year());
469             adjustTime(UTC_offset * SECS_PER_HOUR);
470
471             Serial.print(F("Data: "));
472             Serial.print(day()); Serial.print("/");
473             Serial.print(month()); Serial.print("/");
474             Serial.println(year());
475             Serial.print(F("Hora: "));
476             Serial.print(hour()); Serial.print(":");
477             Serial.print(minute()); Serial.print(":");
478             Serial.println(second());
479           }
480           delay(2000);
481         }
482       }
483       if(millis() > 5000 && gps.charsProcessed() < 10) {
484         Serial.println(F("GPS nao detectado."));
485         while(true);
486       }
487     }
488 }
489
490 /*------------------- TEST MPU --------------------*/
491 void test_mpu() {
492   int16_t dados[9]; //9 dados 16 bits ax ay az gx gy gz mx my mz
493   uint32_t start;
494
495   loading();
496   while(1) {
497     if(analogRead(ADC0) < 1000) {
498       lcd.clear();
499       return;
500     }
```

```
501
502     unsigned long t_now = millis();
503     start = micros();
504
505     mpu.getAres();
506     mpu.getGres();
507     mpu.getMres();
508
509     mpu.readMotionSensor((dados + 0), (dados + 1), (dados + 2),
510                          (dados + 3), (dados + 4), (dados + 5),
511                          (dados + 6), (dados + 7), (dados + 8));
512
513     if(debug) {
514       Serial.print("Ares: "); Serial.println(mpu.aRes, 5);
515       Serial.print("Gres: "); Serial.println(mpu.gRes, 5);
516       Serial.print("Mres: "); Serial.println(mpu.mRes, 5);
517     }
518
519     // Calcular valor da aceleracao em g's
520     ax = (float)dados[0] * (mpu.aRes);
521     ay = (float)dados[1] * (mpu.aRes);
522     az = (float)dados[2] * (mpu.aRes);
523
524     // Calcular o valor do gyro em graus/segundo
525     gx = (float)dados[3] * (mpu.gRes);
526     gy = (float)dados[4] * (mpu.gRes);
527     gz = (float)dados[5] * (mpu.gRes);
528
529     // Calcular valores magnetometro
530     mx = (float)dados[6] * (mpu.mRes) * (mpu.
           factoryMagCalibration[0]) - mpu.magBias[0];
531     my = (float)dados[7] * (mpu.mRes) * (mpu.
           factoryMagCalibration[1]) - mpu.magBias[1];
532     mz = (float)dados[8] * (mpu.mRes) * (mpu.
           factoryMagCalibration[2]) - mpu.magBias[2];
533
534
535     Serial.print(ax, 5); Serial.print(" ");
536     Serial.print(ay, 5); Serial.print(" ");
537     Serial.print(az, 5); Serial.print(" ");
```

```
538       Serial.print(gx, 5); Serial.print(" ");
539       Serial.print(gy, 5); Serial.print(" ");
540       Serial.print(gz, 5); Serial.print(" ");
541       Serial.print(mx, 5); Serial.print(" ");
542       Serial.print(my, 5); Serial.print(" ");
543       Serial.println(mz, 5);
544     }
545   }
546
547   void treat_data() {
548     uint8_t leitura[18];
549     uint32_t now = 0, start;
550     int count = 0;
551
552     mpu.getAres();
553     mpu.getGres();
554     mpu.getMres();
555
556     loading();
557     lcd.clear();
558
559     for(uint32_t j = 0; j < 0x20000; j+=18) {
560       eeprom.read(j, leitura, sizeof(leitura));
561
562       unsigned long t_now = millis();
563       start = micros();
564
565       ax = (float)(((int16_t)leitura[1]<<8)|leitura[0])*(mpu.aRes);
566       ay = (float)(((int16_t)leitura[3]<<8)|leitura[2])*(mpu.aRes);
567       az = (float)(((int16_t)leitura[5]<<8)|leitura[4])*(mpu.aRes);
568       gx = (float)(((int16_t)leitura[7]<<8)|leitura[6])*(mpu.gRes);
569       gy = (float)(((int16_t)leitura[9]<<8)|leitura[8])*(mpu.gRes);
570       gz = (float)(((int16_t)leitura[11]<<8)|leitura[10])*(mpu.gRes
          );
571
572       mpu.Now = micros();
573       mpu.deltat = ((mpu.Now - mpu.lastUpdate)/1000000.0f);
574       mpu.lastUpdate = mpu.Now;
575       mpu.sum += mpu.deltat;
576
```

```
577       do {
578         delayMicroseconds(10);
579       } while((micros() - start) < 10000);
580 
581       MadgwickQuaternionUpdate(-ax, ay, az, gx*PI/180.0f, -gy*PI
              /180.0f, -gz*PI/180.0f);
582 
583       Serial.print(q[0], 8); Serial.print(" ");
584       Serial.print(q[1], 8); Serial.print(" ");
585       Serial.print(q[2], 8); Serial.print(" ");
586       Serial.print(q[3], 8); Serial.print(" ");
587       Serial.print(ax, 8); Serial.print(" ");
588       Serial.print(ay, 8); Serial.print(" ");
589       Serial.println(az, 8);
590     }
591 }
```

A.2 Código para acesso a memória SRAM

A.2.1 SRAM.h

```
#ifndef SRAM_h
#define SRAM_h

#include <Arduino.h>
#include <SPI.h>

#define WRMR 0x01        // Escrever no registrador de modo
#define WRTE 0x02        // Instrucao para escrita
#define READ 0X03        // Instrucao para leitura
#define RDMR 0x05        // Fornece acesso ao registrador de modo
#define BYMD 0x00        // Modo de operacao: Byte
#define PGMD 0x80        // Modo de operacao: Pagina
#define SQMD 0x40        // Modo de operacao: Sequencial

#define CSpin 48         // Pino de controle de acesso

class SRAM {
    public:
      void setMode(uint8_t mode);
      uint8_t getMode();
      void writeSeq(uint32_t addr, int16_t *data, uint16_t
         numBytes);
      void readSeq(uint32_t addr, int16_t *data, uint32_t
         numBytes);
};

#endif
```

A.2.2 SRAM.cpp

```
#include "SRAM.h"

uint8_t SRAM::getMode() {
    uint8_t mode;
    digitalWrite(CSpin, LOW);
    delay(5);
    SPI.transfer(RDMR);
    mode = SPI.transfer(0x00);
    digitalWrite(CSpin, HIGH);
    return mode;
}

void SRAM::setMode(uint8_t mode) {
    digitalWrite(CSpin, LOW);
    SPI.transfer(WRMR);
    SPI.transfer(mode);
    digitalWrite(CSpin, HIGH);
}

void SRAM::writeSeq(uint32_t addr, int16_t *data, uint16_t
   numBytes) {
    setMode(SQMD);
    digitalWrite(CSpin, LOW);
    SPI.transfer(WRTE);
    SPI.transfer((uint8_t)(addr >> 16));
    SPI.transfer((uint8_t)(addr >> 8));
    SPI.transfer((uint8_t)addr);

    for (uint16_t i = 0; i < numBytes; i++) {
        SPI.transfer((uint8_t)(data[i] >> 8));
        SPI.transfer((uint8_t)data[i]);
    }

    digitalWrite(CSpin, HIGH);
}

void SRAM::readSeq(uint32_t addr, int16_t *data, uint32_t
   numBytes) {
```

```
37      setMode(SQMD);
38      digitalWrite(CSpin, LOW);
39      SPI.transfer(READ);
40      SPI.transfer((uint8_t)(addr >> 16));
41      SPI.transfer((uint8_t)(addr >> 8));
42      SPI.transfer((uint8_t)addr);
43
44      for (uint16_t i = 0; i < numBytes; i++) {
45          data[i] = (int16_t)(SPI.transfer(0x00) << 8) | SPI.
                transfer(0x00);
46      }
47
48      digitalWrite(CSpin, HIGH);
49  }
```

A.3 *Driver* MPU9250

A.3.1 MPU9250.h

```
/*
Note: The MPU9250 is an I2C sensor and uses the Arduino Wire
   library. We are also using the 400 kHz fast I2C mode by setting
    the TWI_FREQ  to 400000L /twi.h utility file. */
#ifndef _MPU9250_H_
#define _MPU9250_H_

#include <SPI.h>
#include <Wire.h>
#include "util/crc16.h"
#include <EEPROM.h>

#define SERIAL_DEBUG true

/* See also MPU-9250 Register Map and Descriptions, Revision 4.0,
    RM-MPU-9250A-00, Rev. 1.4, 9/9/2013 for registers not listed
   in above document; the MPU9250 and MPU9150 are virtually
   identical but the latter has a different register map */

//Magnetometer Registers
#define AK8963_ADDRESS   0x0C
#define WHO_AM_I_AK8963  0x00
#define INFO             0x01
#define AK8963_ST1       0x02
#define AK8963_XOUT_L    0x03
#define AK8963_XOUT_H    0x04
#define AK8963_YOUT_L    0x05
#define AK8963_YOUT_H    0x06
#define AK8963_ZOUT_L    0x07
#define AK8963_ZOUT_H    0x08
#define AK8963_ST2       0x09
#define AK8963_CNTL      0x0A
#define AK8963_ASTC      0x0C  // Self test control
#define AK8963_I2CDIS    0x0F  // I2C disable
#define AK8963_ASAX      0x10
#define AK8963_ASAY      0x11
```

```
32 #define AK8963_ASAZ      0x12
33
34 #define SELF_TEST_X_GYRO 0x00
35 #define SELF_TEST_Y_GYRO 0x01
36 #define SELF_TEST_Z_GYRO 0x02
37
38 /*
39 #define X_FINE_GAIN      0x03 // [7:0] fine gain
40 #define Y_FINE_GAIN      0x04
41 #define Z_FINE_GAIN      0x05
42 #define XA_OFFSET_H      0x06 // User-defined trim values for
      accelerometer
43 #define XA_OFFSET_L_TC   0x07
44 #define YA_OFFSET_H      0x08
45 #define YA_OFFSET_L_TC   0x09
46 #define ZA_OFFSET_H      0x0A
47 #define ZA_OFFSET_L_TC   0x0B */
48
49 #define SELF_TEST_X_ACCEL 0x0D
50 #define SELF_TEST_Y_ACCEL 0x0E
51 #define SELF_TEST_Z_ACCEL 0x0F
52
53 #define SELF_TEST_A              0x10
54
55 #define XG_OFFSET_H     0x13  //User-defined trim values for
      gyroscope
56 #define XG_OFFSET_L     0x14
57 #define YG_OFFSET_H     0x15
58 #define YG_OFFSET_L     0x16
59 #define ZG_OFFSET_H     0x17
60 #define ZG_OFFSET_L     0x18
61 #define SMPLRT_DIV      0x19
62 #define CONFIG          0x1A
63 #define GYRO_CONFIG     0x1B
64 #define ACCEL_CONFIG    0x1C
65 #define ACCEL_CONFIG2   0x1D
66 #define LP_ACCEL_ODR    0x1E
67 #define WOM_THR         0x1F
68
```

```
69  // Duration counter threshold for motion interrupt generation, 1
        kHz rate,
70  // LSB = 1 ms
71  #define MOT_DUR           0x20
72  // Zero-motion detection threshold bits [7:0]
73  #define ZMOT_THR          0x21
74  // Duration counter threshold for zero motion interrupt
        generation, 16 Hz rate,
75  // LSB = 64 ms
76  #define ZRMOT_DUR         0x22
77
78  #define FIFO_EN            0x23
79  #define I2C_MST_CTRL       0x24
80  #define I2C_SLV0_ADDR      0x25
81  #define I2C_SLV0_REG       0x26
82  #define I2C_SLV0_CTRL      0x27
83  #define I2C_SLV1_ADDR      0x28
84  #define I2C_SLV1_REG       0x29
85  #define I2C_SLV1_CTRL      0x2A
86  #define I2C_SLV2_ADDR      0x2B
87  #define I2C_SLV2_REG       0x2C
88  #define I2C_SLV2_CTRL      0x2D
89  #define I2C_SLV3_ADDR      0x2E
90  #define I2C_SLV3_REG       0x2F
91  #define I2C_SLV3_CTRL      0x30
92  #define I2C_SLV4_ADDR      0x31
93  #define I2C_SLV4_REG       0x32
94  #define I2C_SLV4_DO        0x33
95  #define I2C_SLV4_CTRL      0x34
96  #define I2C_SLV4_DI        0x35
97  #define I2C_MST_STATUS     0x36
98  #define INT_PIN_CFG        0x37
99  #define INT_ENABLE         0x38
100 #define DMP_INT_STATUS     0x39  // Check DMP interrupt
101 #define INT_STATUS         0x3A
102 #define ACCEL_XOUT_H       0x3B
103 #define ACCEL_XOUT_L       0x3C
104 #define ACCEL_YOUT_H       0x3D
105 #define ACCEL_YOUT_L       0x3E
106 #define ACCEL_ZOUT_H       0x3F
```

```
107 #define ACCEL_ZOUT_L       0x40
108 #define TEMP_OUT_H         0x41
109 #define TEMP_OUT_L         0x42
110 #define GYRO_XOUT_H        0x43
111 #define GYRO_XOUT_L        0x44
112 #define GYRO_YOUT_H        0x45
113 #define GYRO_YOUT_L        0x46
114 #define GYRO_ZOUT_H        0x47
115 #define GYRO_ZOUT_L        0x48
116 #define EXT_SENS_DATA_00   0x49
117 #define EXT_SENS_DATA_01   0x4A
118 #define EXT_SENS_DATA_02   0x4B
119 #define EXT_SENS_DATA_03   0x4C
120 #define EXT_SENS_DATA_04   0x4D
121 #define EXT_SENS_DATA_05   0x4E
122 #define EXT_SENS_DATA_06   0x4F
123 #define EXT_SENS_DATA_07   0x50
124 #define EXT_SENS_DATA_08   0x51
125 #define EXT_SENS_DATA_09   0x52
126 #define EXT_SENS_DATA_10   0x53
127 #define EXT_SENS_DATA_11   0x54
128 #define EXT_SENS_DATA_12   0x55
129 #define EXT_SENS_DATA_13   0x56
130 #define EXT_SENS_DATA_14   0x57
131 #define EXT_SENS_DATA_15   0x58
132 #define EXT_SENS_DATA_16   0x59
133 #define EXT_SENS_DATA_17   0x5A
134 #define EXT_SENS_DATA_18   0x5B
135 #define EXT_SENS_DATA_19   0x5C
136 #define EXT_SENS_DATA_20   0x5D
137 #define EXT_SENS_DATA_21   0x5E
138 #define EXT_SENS_DATA_22   0x5F
139 #define EXT_SENS_DATA_23   0x60
140 #define MOT_DETECT_STATUS  0x61
141 #define I2C_SLV0_DO        0x63
142 #define I2C_SLV1_DO        0x64
143 #define I2C_SLV2_DO        0x65
144 #define I2C_SLV3_DO        0x66
145 #define I2C_MST_DELAY_CTRL 0x67
146 #define SIGNAL_PATH_RESET  0x68
```

```
147 #define MOT_DETECT_CTRL     0x69
148 #define USER_CTRL           0x6A  // Bit 7 enable DMP, bit 3 reset
        DMP
149 #define PWR_MGMT_1          0x6B // Device defaults to the SLEEP
        mode
150 #define PWR_MGMT_2          0x6C
151 #define DMP_BANK            0x6D  // Activates a specific bank in
        the DMP
152 #define DMP_RW_PNT          0x6E  // Set read/write pointer to a
        specific start address in specified DMP bank
153 #define DMP_REG             0x6F  // Register in DMP from which to
        read or to which to write
154 #define DMP_REG_1           0x70
155 #define DMP_REG_2           0x71
156 #define FIFO_COUNTH         0x72
157 #define FIFO_COUNTL         0x73
158 #define FIFO_R_W            0x74
159 #define WHO_AM_I_MPU9250    0x75 // Should return 0x71
160 #define XA_OFFSET_H         0x77
161 #define XA_OFFSET_L         0x78
162 #define YA_OFFSET_H         0x7A
163 #define YA_OFFSET_L         0x7B
164 #define ZA_OFFSET_H         0x7D
165 #define ZA_OFFSET_L         0x7E
166
167 // Using the MPU-9250 breakout board, ADO is set to 0
168 // Seven-bit device address is 110100 for ADO = 0 and 110101 for
        ADO = 1
169 #define ADO 0
170 #if ADO
171 #define MPU9250_ADDRESS 0x69  // Device address when ADO = 1
172 #else
173 #define MPU9250_ADDRESS 0x68  // Device address when ADO = 0
174 #define AK8963_ADDRESS  0x0C   // Address of magnetometer
175 #endif // ADO
176
177 #define READ_FLAG 0x80
178 #define NOT_SPI -1
179 #define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the MPU
        -9250
```

```
180 //#define SPI_DATA_RATE 1000000 // 1MHz is the max speed of the
    MPU-9250
181 #define SPI_MODE SPI_MODE3
182
183  #define MPU9250_CAL_SIZE 68
184  #define MPU9250_CAL_EEADDR 60
185
186 class MPU9250
187 {
188   protected:
189     // Set initial input parameters
190     enum Ascale
191     {
192       AFS_2G = 0,
193       AFS_4G,
194       AFS_8G,
195       AFS_16G
196     };
197
198     enum Gscale {
199       GFS_250DPS = 0,
200       GFS_500DPS,
201       GFS_1000DPS,
202       GFS_2000DPS
203     };
204
205     enum Mscale {
206       MFS_14BITS = 0, // 0.60 mG per LSB
207       MFS_16BITS      // 0.15 mG per LSB
208
209     };
210
211     enum M_MODE {
212       M_8HZ = 0x02,  // 8 Hz update
213       M_100HZ = 0x06 // 100 Hz continuous magnetometer
214     };
215
216     // TODO: Add setter methods for this hard coded stuff
217     // Specify sensor full scale
218     uint8_t Gscale = GFS_250DPS;
```

```
219     uint8_t Ascale = AFS_2G;
220     // Choose either 14-bit or 16-bit magnetometer resolution
221     uint8_t Mscale = MFS_16BITS;
222
223     // 2 for 8 Hz, 6 for 100 Hz continuous magnetometer data read
224     uint8_t Mmode = M_100HZ;
225
226     // SPI chip select pin
227     int8_t _csPin;
228
229     uint8_t writeByteWire(uint8_t, uint8_t, uint8_t);
230     uint8_t writeByteSPI(uint8_t, uint8_t);
231     uint8_t readByteSPI(uint8_t subAddress);
232     uint8_t readByteWire(uint8_t address, uint8_t subAddress);
233     bool magInit();
234     void kickHardware();
235     void select();
236     void deselect();
237 // TODO: Remove this next line
238 public:
239     uint8_t ak8963WhoAmI_SPI();
240
241   public:
242     float pitch, yaw, roll;
243     float temperature;   // Stores the real internal chip
          temperature in Celsius
244     int16_t tempCount;   // Temperature raw count output
245     uint32_t delt_t = 0; // Used to control display output rate
246
247     uint32_t count = 0, sumCount = 0; // used to control display
          output rate
248     float deltat = 0.0f, sum = 0.0f;  // integration interval for
           both filter schemes
249     uint32_t lastUpdate = 0, firstUpdate = 0; // used to
          calculate integration interval
250     uint32_t Now = 0;        // used to calculate integration
          interval
251
252     int16_t gyroCount[3];   // Stores the 16-bit signed gyro
          sensor output
```

```
253     int16_t magCount[3];    // Stores the 16-bit signed
           magnetometer sensor output
254     // Scale resolutions per LSB for the sensors
255     float aRes, gRes, mRes;
256     // Variables to hold latest sensor data values
257     //float ax, ay, az, gx, gy, gz, mx, my, mz, temp;
258     int16_t accel_temp_raw[4];
259     int16_t mag_raw[3];
260     int16_t gyro_raw[3];
261     uint8_t newdata;
262     float cal[16];
263
264     // Factory mag calibration and mag bias
265     float factoryMagCalibration[3] = {0, 0, 0}, factoryMagBias[3]
            = {0, 0, 0};
266     // Bias corrections for gyro, accelerometer, and magnetometer
267     float gyroBias[3]  = {0, 0, 0},
268           accelBias[3] = {0, 0, 0},
269           magBias[3]   = {0, 0, 0},
270           magScale[3]  = {0, 0, 0};
271     float selfTest[6];
272     // Stores the 16-bit signed accelerometer sensor output
273     int16_t accelCount[3];
274
275     // Public method declarations
276     MPU9250(int8_t csPin=NOT_SPI);
277     void getMres();
278     void getGres();
279     void getAres();
280     int readAccelData(int16_t *);
281     void readMotionSensor(int16_t *ax, int16_t *ay, int16_t *az,
           int16_t *gx, int16_t *gy, int16_t *gz, int16_t *mx, int16_t
            *my,int16_t *mz);
282     void update();
283     bool writeCalibration(const void *);
284     void getCalibration(float *, float *, float *);
285     bool available();
286     int readGyroData(int16_t *);
287     int readMagData(int16_t *);
288     void readTempData(int16_t *);
```

```
289     void updateTime();
290     void initAK8963(float *);
291     void initMPU9250();
292     void calibrateMPU9250(float * gyroBias, float * accelBias);
293     void MPU9250SelfTest(float * destination);
294     void magCalMPU9250(float * dest1, float * dest2);
295     uint8_t writeByte(uint8_t, uint8_t, uint8_t);
296     uint8_t readByte(uint8_t, uint8_t);
297     uint8_t readBytes(uint8_t, uint8_t, uint8_t, uint8_t *);
298     // TODO: make SPI/Wire private
299     uint8_t readBytesSPI(uint8_t, uint8_t, uint8_t *);
300     uint8_t readBytesWire(uint8_t, uint8_t, uint8_t, uint8_t *);
301     bool isInI2cMode() { return _csPin == -1; }
302     bool begin();
303 };  // class MPU9250
304
305 #endif // _MPU9250_H_
```

A.3.2 MPU9250.cpp

```
#include "MPU9250_t.h"

//=========================================================
//Set of useful function to access acceleration, gyroscope,
   magnetometer, and temperature data
//=========================================================

MPU9250::MPU9250(int8_t cspin /*=NOT_SPI*/)
// Uses I2C communication by default
{
  /* Use hardware SPI communication
  If used with sparkfun breakout board https://www.sparkfun.com/
     products/13762 , change the pre-soldered JP2 to enable SPI (
     solder middle and left instead of middle and right) pads are
     very small and re-soldering can be very tricky. I2C highly
     recommended. */
  if ((cspin > NOT_SPI) && (cspin < NUM_DIGITAL_PINS)) {
    _csPin = cspin;
    SPI.begin();
    pinMode(_csPin, OUTPUT);
    deselect();
  } else {
    _csPin = NOT_SPI;
    Wire.begin();
    Wire.setClock(400000);
  }
}

void MPU9250::readMotionSensor(int16_t *ax, int16_t *ay, int16_t
   *az, int16_t *gx, int16_t *gy, int16_t *gz, int16_t *mx,
   int16_t *my, int16_t *mz) {
  if (readByte(MPU9250_ADDRESS, INT_STATUS) & 0x01) update();
  *ax = accel_temp_raw[0];
  *ay = accel_temp_raw[1];
  *az = accel_temp_raw[2];
  *mx = mag_raw[0];
  *my = mag_raw[1];
  *mz = mag_raw[2];
```

```
32    *gx = gyro_raw[0];
33    *gy = gyro_raw[1];
34    *gz = gyro_raw[2];
35  }
36
37  bool MPU9250::available() {
38    update();
39    if (readByte(MPU9250_ADDRESS, INT_STATUS) & 0x01) return true;
40    return false;
41  }
42
43  void MPU9250::getMres() {
44    switch (Mscale) {
45      /*Possible magnetometer scales (and their register bit
          settings) are: 14 bit resolution(0) and 16 bit resolution
          (1)*/
46      case MFS_14BITS:
47        mRes = 10.0f * 4912.0f / 8190.0f; // Proper scale to return
              milliGauss
48        break;
49      case MFS_16BITS:
50        mRes = (10.0f * 4912.0f / 32760.0f) / 1.5; // Proper scale
            to return milliGauss
51        break;
52    }
53  }
54
55  void MPU9250::getGres() {
56    switch (Gscale) {
57      /* Possible gyro scales(and their register bit settings) are:
           250 DPS (00), 500 DPS (01), 1000 DPS (10), and 2000 DPS
          (11). Here's a bit of an algorith to calculate DPS/(ADC
          tick) based on that 2-bit value:*/
58      case GFS_250DPS:
59        gRes = 250.0f / 32768.0f;
60        break;
61      case GFS_500DPS:
62        gRes = 500.0f / 32768.0f;
63        break;
64      case GFS_1000DPS:
```

```
65        gRes = 1000.0f / 32768.0f;
66        break;
67      case GFS_2000DPS:
68        gRes = 2000.0f / 32768.0f;
69        break;
70    }
71  }
72
73  void MPU9250::getAres() {
74    switch (Ascale) {
75      /* Possible accelerometer scales (and their register bit
           settings) are: 2 Gs(00), 4 Gs(01), 8 Gs(10), and 16 Gs(11).
            Here's a bit of an algorith to calculate DPS/(ADC tick)
           based on that 2-bit value:*/
76      case AFS_2G:
77        aRes = 2.0f / 32768.0f; break;
78      case AFS_4G:
79        aRes = 4.0f / 32768.0f; break;
80      case AFS_8G:
81        aRes = 8.0f / 32768.0f; break;
82      case AFS_16G:
83        aRes = 16.0f / 32768.0f; break;
84    }
85  }
86
87  int MPU9250::readAccelData(int16_t *destination) {
88    uint8_t rawData[6];  // x/y/z accel register data stored here
89    // Read the six raw data registers into data array
90    readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
91
92    // Turn the MSB and LSB into a signed 16-bit value
93    destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ; //
        accel x
94    destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ; //
        accel y
95    destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ; //
        accel z
96    //destination[3] = ((int16_t)rawData[6] << 8) | rawData[7] ; //
        temperature
97
```

```
98    return 1;
99  }
100
101
102 int MPU9250::readGyroData(int16_t *destination) {
103   uint8_t rawData[6];  // x/y/z gyro register data stored here
104   // Read the six raw data registers sequentially into data array
105   readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
106
107   // Turn the MSB and LSB into a signed 16-bit value
108   destination[0] = ((int16_t)rawData[0] << 8) | rawData[1] ;
109   destination[1] = ((int16_t)rawData[2] << 8) | rawData[3] ;
110   destination[2] = ((int16_t)rawData[4] << 8) | rawData[5] ;
111
112   return 1;
113 }
114
115 int MPU9250::readMagData(int16_t *destination) {
116   // x/y/z gyro register data, ST2 register stored here, must
        read ST2 at end
117   // of data acquisition
118   uint8_t rawData[7];
119   // Wait for magnetometer data ready bit to be set
120   if (1/*readByte(AK8963_ADDRESS, AK8963_ST1) & 0x01*/) {
121     // Read the six raw data and ST2 registers sequentially into
          data array
122     readBytes(AK8963_ADDRESS, AK8963_XOUT_L, 7, &rawData[0]);
123     uint8_t c = rawData[6]; // End data read by reading ST2
          register
124     // Check if magnetic sensor overflow set, if not then report
          data
125     if (!(c & 0x08)) {
126       // Turn the MSB and LSB into a signed 16-bit value
127       destination[0] = ((int16_t)rawData[1] << 8) | rawData[0];
128       // Data stored as little Endian
129       destination[1] = ((int16_t)rawData[3] << 8) | rawData[2];
130       destination[2] = ((int16_t)rawData[5] << 8) | rawData[4];
131
132       return 1;
133     }
```

```
134   }
135   return 0;
136 }
137
138 void MPU9250::readTempData(int16_t * destination) {
139   uint8_t rawData[2]; // x/y/z gyro register data stored here
140   // Read the two raw data registers sequentially into data array
141   readBytes(MPU9250_ADDRESS, TEMP_OUT_H, 2, &rawData[0]);
142   // Turn the MSB and LSB into a 16-bit value
143   destination = ((int16_t)rawData[0] << 8) | rawData[1];
144 }
145
146 // Calculate the time the last update took for use in the
      quaternion filters
147 // TODO: This doesn't really belong in this class.
148 void MPU9250::updateTime() {
149   Now = micros();
150
151   // Set integration time by time elapsed since last filter
        update
152   deltat = ((Now - lastUpdate) / 1000000.0f);
153   lastUpdate = Now;
154
155   sum += deltat; // sum for averaging filter update rate
156   sumCount++;
157 }
158
159 void MPU9250::initAK8963(float * destination) {
160   // First extract the factory calibration for each magnetometer
        axis
161   uint8_t rawData[3];  // x/y/z gyro calibration data stored here
162   // TODO: Test this!! Likely doesn't work
163   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
        magnetometer
164   delay(10);
165   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x0F); // Enter Fuse ROM
        access mode
166   delay(10);
167
168   // Read the x-, y-, and z-axis calibration values
```

```
169   readBytes(AK8963_ADDRESS, AK8963_ASAX, 3, &rawData[0]);
170
171   // Return x-axis sensitivity adjustment values, etc.
172   destination[0] =  (float)(rawData[0] - 128) / 256. + 1.;
173   destination[1] =  (float)(rawData[1] - 128) / 256. + 1.;
174   destination[2] =  (float)(rawData[2] - 128) / 256. + 1.;
175   writeByte(AK8963_ADDRESS, AK8963_CNTL, 0x00); // Power down
        magnetometer
176   delay(10);
177
178   // Configure the magnetometer for continuous read and highest
        resolution.
179   // Set Mscale bit 4 to 1 (0) to enable 16 (14) bit resolution
        in CNTL
180   // register, and enable continuous mode data acquisition Mmode
        (bits [3:0]),
181   // 0010 for 8 Hz and 0110 for 100 Hz sample rates.
182
183   // Set magnetometer data resolution and sample ODR
184   writeByte(AK8963_ADDRESS, AK8963_CNTL, Mscale << 4 | Mmode);
185   delay(10);
186 }
187
188 void MPU9250::initMPU9250() {
189   // wake up device
190   // Clear sleep mode bit (6), enable all sensors
191   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
192   delay(100); // Wait for all registers to reset
193
194   // Get stable time source
195   // Auto select clock source to be PLL gyroscope reference if
        ready else
196   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
197   delay(200);
198
199   // Configure Gyro and Thermometer
200   // Disable FSYNC and set thermometer and gyro bandwidth to 41
        and 42 Hz,
201   // respectively;
```

```
202   // minimum delay time for this setting is 5.9 ms, which means
          sensor fusion
203   // update rates cannot be higher than 1 / 0.0059 = 170 Hz
204   // DLPF_CFG = bits 2:0 = 011; this limits the sample rate to
          1000 Hz for both
205   // With the MPU9250, it is possible to get gyro sample rates of
           32 kHz (!),
206   // 8 kHz, or 1 kHz
207   writeByte(MPU9250_ADDRESS, CONFIG, 0x03);
208
209   // Set sample rate = gyroscope output rate/(1 + SMPLRT_DIV)
210   // Use a 200 Hz rate; a rate consistent with the filter update
          rate
211   // determined inset in CONFIG above.
212   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x04);
213
214   // Set gyroscope full scale range
215   // Range selects FS_SEL and AFS_SEL are 0 - 3, so 2-bit values
          are
216   // left-shifted into positions 4:3
217
218   // get current GYRO_CONFIG register value
219   uint8_t c = readByte(MPU9250_ADDRESS, GYRO_CONFIG);
220   // c = c & ~0xE0; // Clear self-test bits [7:5]
221   c = c & ~0x02; // Clear Fchoice bits [1:0]
222   c = c & ~0x18; // Clear AFS bits [4:3]
223   c = c | Gscale << 3; // Set full scale range for the gyro
224   // Set Fchoice for the gyro to 11 by writing its inverse to
          bits 1:0 of
225   // GYRO_CONFIG
226   // c =| 0x00;
227   // Write new GYRO_CONFIG value to register
228   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, c );
229
230   // Set accelerometer full-scale range configuration
231   // Get current ACCEL_CONFIG register value
232   c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG);
233   // c = c & ~0xE0; // Clear self-test bits [7:5]
234   c = c & ~0x18;  // Clear AFS bits [4:3]
```

```
235   c = c | Ascale << 3; // Set full scale range for the
          accelerometer
236   // Write new ACCEL_CONFIG register value
237   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, c);
238
239   // Set accelerometer sample rate configuration
240   // It is possible to get a 4 kHz sample rate from the
          accelerometer by
241   // choosing 1 for accel_fchoice_b bit [3]; in this case the
          bandwidth is
242   // 1.13 kHz
243   // Get current ACCEL_CONFIG2 register value
244   c = readByte(MPU9250_ADDRESS, ACCEL_CONFIG2);
245   c = c & ~0x0F; // Clear accel_fchoice_b (bit 3) and A_DLPFG (
          bits [2:0])
246   c = c | 0x03;  // Set accelerometer rate to 1 kHz and bandwidth
           to 41 Hz
247   // Write new ACCEL_CONFIG2 register value
248   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, c);
249   // The accelerometer, gyro, and thermometer are set to 1 kHz
          sample rates,
250   // but all these rates are further reduced by a factor of 5 to
          200 Hz because
251   // of the SMPLRT_DIV setting
252
253   // Configure Interrupts and Bypass Enable
254   // Set interrupt pin active high, push-pull, hold interrupt pin
           level HIGH
255   // until interrupt cleared, clear on read of INT_STATUS, and
          enable
256   // I2C_BYPASS_EN so additional chips can join the I2C bus and
          all can be
257   // controlled by the Arduino as master.
258   writeByte(MPU9250_ADDRESS, INT_PIN_CFG, 0x22);
259   // Enable data ready (bit 0) interrupt
260   writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x01);
261   delay(100);
262 }
263
```

```
264 // Function which accumulates gyro and accelerometer data after
    device
265 // initialization. It calculates the average of the at-rest
    readings and then
266 // loads the resulting offsets into accelerometer and gyro bias
    registers.
267 void MPU9250::calibrateMPU9250(float * gyroBias, float *
    accelBias) {
268   uint8_t data[12]; // data array to hold accelerometer and gyro
      x, y, z, data
269   uint16_t ii, packet_count, fifo_count;
270   int32_t gyro_bias[3]  = {0, 0, 0}, accel_bias[3] = {0, 0, 0};
271
272   // reset device
273   // Write a one to bit 7 reset bit; toggle reset device
274   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, READ_FLAG);
275   delay(100);
276
277   // get stable time source; Auto select clock source to be PLL
      gyroscope
278   // reference if ready else use the internal oscillator, bits
      2:0 = 001
279   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x01);
280   writeByte(MPU9250_ADDRESS, PWR_MGMT_2, 0x00);
281   delay(200);
282
283   // Configure device for bias calculation
284   // Disable all interrupts
285   writeByte(MPU9250_ADDRESS, INT_ENABLE, 0x00);
286   // Disable FIFO
287   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
288   // Turn on internal clock source
289   writeByte(MPU9250_ADDRESS, PWR_MGMT_1, 0x00);
290   // Disable I2C master
291   writeByte(MPU9250_ADDRESS, I2C_MST_CTRL, 0x00);
292   // Disable FIFO and I2C master modes
293   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x00);
294   // Reset FIFO and DMP
295   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x0C);
296   delay(15);
```

```
297
298   // Configure MPU6050 gyro and accelerometer for bias
          calculation
299   // Set low-pass filter to 188 Hz
300   writeByte(MPU9250_ADDRESS, CONFIG, 0x01);
301   // Set sample rate to 1 kHz
302   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
303   // Set gyro full-scale to 250 degrees per second, maximum
          sensitivity
304   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 0x00);
305   // Set accelerometer full-scale to 2 g, maximum sensitivity
306   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
307
308   uint16_t  gyrosensitivity  = 131;   // = 131 LSB/degrees/sec
309   uint16_t  accelsensitivity = 16384; // = 16384 LSB/g
310
311   // Configure FIFO to capture accelerometer and gyro data for
          bias calculation
312   writeByte(MPU9250_ADDRESS, USER_CTRL, 0x40);  // Enable FIFO
313   // Enable gyro and accelerometer sensors for FIFO  (max size
          512 bytes in
314   // MPU-9150)
315   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x78);
316   delay(40);  // accumulate 40 samples in 40 milliseconds = 480
          bytes
317
318   // At end of sample accumulation, turn off FIFO sensor read
319   // Disable gyro and accelerometer sensors for FIFO
320   writeByte(MPU9250_ADDRESS, FIFO_EN, 0x00);
321   // Read FIFO sample count
322   readBytes(MPU9250_ADDRESS, FIFO_COUNTH, 2, &data[0]);
323   fifo_count = ((uint16_t)data[0] << 8) | data[1];
324   // How many sets of full gyro and accelerometer data for
          averaging
325   packet_count = fifo_count / 12;
326
327   for (ii = 0; ii < packet_count; ii++)
328   {
329     int16_t accel_temp[3] = {0, 0, 0}, gyro_temp[3] = {0, 0, 0};
330     // Read data for averaging
```

```
331     readBytes(MPU9250_ADDRESS, FIFO_R_W, 12, &data[0]);
332     // Form signed 16-bit integer for each sample in FIFO
333     accel_temp[0] = (int16_t) (((int16_t)data[0] << 8) | data[1]
            );
334     accel_temp[1] = (int16_t) (((int16_t)data[2] << 8) | data[3]
            );
335     accel_temp[2] = (int16_t) (((int16_t)data[4] << 8) | data[5]
            );
336     gyro_temp[0]  = (int16_t) (((int16_t)data[6] << 8) | data[7]
            );
337     gyro_temp[1]  = (int16_t) (((int16_t)data[8] << 8) | data[9]
            );
338     gyro_temp[2]  = (int16_t) (((int16_t)data[10] << 8) | data
           [11]);
339
340     // Sum individual signed 16-bit biases to get accumulated
           signed 32-bit
341     // biases.
342     accel_bias[0] += (int32_t) accel_temp[0];
343     accel_bias[1] += (int32_t) accel_temp[1];
344     accel_bias[2] += (int32_t) accel_temp[2];
345     gyro_bias[0]  += (int32_t) gyro_temp[0];
346     gyro_bias[1]  += (int32_t) gyro_temp[1];
347     gyro_bias[2]  += (int32_t) gyro_temp[2];
348   }
349   // Sum individual signed 16-bit biases to get accumulated
         signed 32-bit biases
350   accel_bias[0] /= (int32_t) packet_count;
351   accel_bias[1] /= (int32_t) packet_count;
352   accel_bias[2] /= (int32_t) packet_count;
353   gyro_bias[0]  /= (int32_t) packet_count;
354   gyro_bias[1]  /= (int32_t) packet_count;
355   gyro_bias[2]  /= (int32_t) packet_count;
356
357   // Sum individual signed 16-bit biases to get accumulated
         signed 32-bit biases
358   if (accel_bias[2] > 0L) {
359     accel_bias[2] -= (int32_t) accelsensitivity;
360   } else {
361     accel_bias[2] += (int32_t) accelsensitivity;
```

```
362   }
363
364   // Construct the gyro biases for push to the hardware gyro bias
          registers,
365   // which are reset to zero upon device startup.
366   // Divide by 4 to get 32.9 LSB per deg/s to conform to expected
          bias input
367   // format.
368   data[0] = (-gyro_bias[0] / 4  >> 8) & 0xFF;
369   // Biases are additive, so change sign on calculated average
         gyro biases
370   data[1] = (-gyro_bias[0] / 4)       & 0xFF;
371   data[2] = (-gyro_bias[1] / 4  >> 8) & 0xFF;
372   data[3] = (-gyro_bias[1] / 4)       & 0xFF;
373   data[4] = (-gyro_bias[2] / 4  >> 8) & 0xFF;
374   data[5] = (-gyro_bias[2] / 4)       & 0xFF;
375
376   // Push gyro biases to hardware registers
377   writeByte(MPU9250_ADDRESS, XG_OFFSET_H, data[0]);
378   writeByte(MPU9250_ADDRESS, XG_OFFSET_L, data[1]);
379   writeByte(MPU9250_ADDRESS, YG_OFFSET_H, data[2]);
380   writeByte(MPU9250_ADDRESS, YG_OFFSET_L, data[3]);
381   writeByte(MPU9250_ADDRESS, ZG_OFFSET_H, data[4]);
382   writeByte(MPU9250_ADDRESS, ZG_OFFSET_L, data[5]);
383
384   // Output scaled gyro biases for display in the main program
385   gyroBias[0] = (float) gyro_bias[0] / (float) gyrosensitivity;
386   gyroBias[1] = (float) gyro_bias[1] / (float) gyrosensitivity;
387   gyroBias[2] = (float) gyro_bias[2] / (float) gyrosensitivity;
388
389   // Construct the accelerometer biases for push to the hardware
         accelerometer
390   // bias registers. These registers contain factory trim values
         which must be
391   // added to the calculated accelerometer biases; on boot up
         these registers
392   // will hold non-zero values. In addition, bit 0 of the lower
         byte must be
393   // preserved since it is used for temperature compensation
         calculations.
```

```
394   // Accelerometer bias registers expect bias input as 2048 LSB
          per g, so that
395   // the accelerometer biases calculated above must be divided by
          8.
396
397   // A place to hold the factory accelerometer trim biases
398   int32_t accel_bias_reg[3] = {0, 0, 0};
399   // Read factory accelerometer trim values
400   readBytes(MPU9250_ADDRESS, XA_OFFSET_H, 2, &data[0]);
401   accel_bias_reg[0] = (int32_t) (((int16_t)data[0] << 8) | data
          [1]);
402   readBytes(MPU9250_ADDRESS, YA_OFFSET_H, 2, &data[0]);
403   accel_bias_reg[1] = (int32_t) (((int16_t)data[0] << 8) | data
          [1]);
404   readBytes(MPU9250_ADDRESS, ZA_OFFSET_H, 2, &data[0]);
405   accel_bias_reg[2] = (int32_t) (((int16_t)data[0] << 8) | data
          [1]);
406
407   // Define mask for temperature compensation bit 0 of lower byte
          of
408   // accelerometer bias registers
409   uint32_t mask = 1uL;
410   // Define array to hold mask bit for each accelerometer bias
          axis
411   uint8_t mask_bit[3] = {0, 0, 0};
412
413   for (ii = 0; ii < 3; ii++) {
414     // If temperature compensation bit is set, record that fact
            in mask_bit
415     if ((accel_bias_reg[ii] & mask)) {
416       mask_bit[ii] = 0x01;
417     }
418   }
419
420   // Construct total accelerometer bias, including calculated
          average
421   // accelerometer bias from above
422   // Subtract calculated averaged accelerometer bias scaled to
          2048 LSB/g
423   // (16 g full scale)
```

```
424   accel_bias_reg[0] -= (accel_bias[0] / 8);
425   accel_bias_reg[1] -= (accel_bias[1] / 8);
426   accel_bias_reg[2] -= (accel_bias[2] / 8);
427
428   data[0] = (accel_bias_reg[0] >> 8) & 0xFF;
429   data[1] = (accel_bias_reg[0])      & 0xFF;
430   // preserve temperature compensation bit when writing back to
        accelerometer
431   // bias registers
432   data[1] = data[1] | mask_bit[0];
433   data[2] = (accel_bias_reg[1] >> 8) & 0xFF;
434   data[3] = (accel_bias_reg[1])      & 0xFF;
435   // Preserve temperature compensation bit when writing back to
        accelerometer
436   // bias registers
437   data[3] = data[3] | mask_bit[1];
438   data[4] = (accel_bias_reg[2] >> 8) & 0xFF;
439   data[5] = (accel_bias_reg[2])      & 0xFF;
440   // Preserve temperature compensation bit when writing back to
        accelerometer
441   // bias registers
442   data[5] = data[5] | mask_bit[2];
443
444   // Apparently this is not working for the acceleration biases
        in the MPU-9250
445   // Are we handling the temperature correction bit properly?
446   // Push accelerometer biases to hardware registers
447   writeByte(MPU9250_ADDRESS, XA_OFFSET_H, data[0]);
448   writeByte(MPU9250_ADDRESS, XA_OFFSET_L, data[1]);
449   writeByte(MPU9250_ADDRESS, YA_OFFSET_H, data[2]);
450   writeByte(MPU9250_ADDRESS, YA_OFFSET_L, data[3]);
451   writeByte(MPU9250_ADDRESS, ZA_OFFSET_H, data[4]);
452   writeByte(MPU9250_ADDRESS, ZA_OFFSET_L, data[5]);
453
454   // Output scaled accelerometer biases for display in the main
        program
455   accelBias[0] = (float)accel_bias[0] / (float)accelsensitivity;
456   accelBias[1] = (float)accel_bias[1] / (float)accelsensitivity;
457   accelBias[2] = (float)accel_bias[2] / (float)accelsensitivity;
458 }
```

```
459
460
461 // Accelerometer and gyroscope self test; check calibration wrt
       factory settings
462 // Should return percent deviation from factory trim values, +/-
       14 or less
463 // deviation is a pass.
464 void MPU9250::MPU9250SelfTest(float * destination) {
465   uint8_t rawData[6] = {0, 0, 0, 0, 0, 0};
466   uint8_t selfTest[6];
467   int32_t gAvg[3] = {0}, aAvg[3] = {0}, aSTAvg[3] = {0}, gSTAvg
        [3] = {0};
468   float factoryTrim[6];
469   uint8_t FS = 0;
470
471   // Set gyro sample rate to 1 kHz
472   writeByte(MPU9250_ADDRESS, SMPLRT_DIV, 0x00);
473   // Set gyro sample rate to 1 kHz and DLPF to 92 Hz
474   writeByte(MPU9250_ADDRESS, CONFIG, 0x02);
475   // Set full scale range for the gyro to 250 dps
476   writeByte(MPU9250_ADDRESS, GYRO_CONFIG, 1 << FS);
477   // Set accelerometer rate to 1 kHz and bandwidth to 92 Hz
478   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG2, 0x02);
479   // Set full scale range for the accelerometer to 2 g
480   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 1 << FS);
481
482   // Get average current values of gyro and acclerometer
483   for (int ii = 0; ii < 200; ii++) {
484     //Serial.print("BHW::ii = ");
485     //Serial.println(ii);
486     // Read the six raw data registers into data array
487     readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
488     // Turn the MSB and LSB into a signed 16-bit value
489     aAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1])
          ;
490     aAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3])
          ;
491     aAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5])
          ;
492
```

```
493     // Read the six raw data registers sequentially into data
           array
494     readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
495     // Turn the MSB and LSB into a signed 16-bit value
496     gAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData[1])
            ;
497     gAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData[3])
            ;
498     gAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData[5])
            ;
499   }
500
501   // Get average of 200 values and store as average current
        readings
502   for (int ii = 0; ii < 3; ii++) {
503     aAvg[ii] /= 200;
504     gAvg[ii] /= 200;
505   }
506
507   // Configure the accelerometer for self-test
508   // Enable self test on all three axes and set accelerometer
        range to +/- 2 g
509   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0xE0);
510   // Enable self test on all three axes and set gyro range to +/-
        250 degrees/s
511   writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0xE0);
512   delay(25);  // Delay a while to let the device stabilize
513
514   // Get average self-test values of gyro and acclerometer
515   for (int ii = 0; ii < 200; ii++) {
516     // Read the six raw data registers into data array
517     readBytes(MPU9250_ADDRESS, ACCEL_XOUT_H, 6, &rawData[0]);
518     // Turn the MSB and LSB into a signed 16-bit value
519     aSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData
           [1]) ;
520     aSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData
           [3]) ;
521     aSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData
           [5]) ;
522
```

```
523     // Read the six raw data registers sequentially into data
            array
524     readBytes(MPU9250_ADDRESS, GYRO_XOUT_H, 6, &rawData[0]);
525     // Turn the MSB and LSB into a signed 16-bit value
526     gSTAvg[0] += (int16_t)(((int16_t)rawData[0] << 8) | rawData
            [1]) ;
527     gSTAvg[1] += (int16_t)(((int16_t)rawData[2] << 8) | rawData
            [3]) ;
528     gSTAvg[2] += (int16_t)(((int16_t)rawData[4] << 8) | rawData
            [5]) ;
529   }
530
531   // Get average of 200 values and store as average self-test
        readings
532   for (int ii = 0; ii < 3; ii++) {
533     aSTAvg[ii] /= 200;
534     gSTAvg[ii] /= 200;
535   }
536
537   // Configure the gyro and accelerometer for normal operation
538   writeByte(MPU9250_ADDRESS, ACCEL_CONFIG, 0x00);
539   writeByte(MPU9250_ADDRESS, GYRO_CONFIG,  0x00);
540   delay(25);  // Delay a while to let the device stabilize
541
542   // Retrieve accelerometer and gyro factory Self-Test Code from
        USR_Reg
543   // X-axis accel self-test results
544   selfTest[0] = readByte(MPU9250_ADDRESS, SELF_TEST_X_ACCEL);
545   // Y-axis accel self-test results
546   selfTest[1] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_ACCEL);
547   // Z-axis accel self-test results
548   selfTest[2] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_ACCEL);
549   // X-axis gyro self-test results
550   selfTest[3] = readByte(MPU9250_ADDRESS, SELF_TEST_X_GYRO);
551   // Y-axis gyro self-test results
552   selfTest[4] = readByte(MPU9250_ADDRESS, SELF_TEST_Y_GYRO);
553   // Z-axis gyro self-test results
554   selfTest[5] = readByte(MPU9250_ADDRESS, SELF_TEST_Z_GYRO);
555
556   // Retrieve factory self-test value from self-test code reads
```

```
557   // FT[Xa] factory trim calculation
558   factoryTrim[0] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[0] - 1.0) ));
559   // FT[Ya] factory trim calculation
560   factoryTrim[1] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[1] - 1.0) ));
561   // FT[Za] factory trim calculation
562   factoryTrim[2] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[2] - 1.0) ));
563   // FT[Xg] factory trim calculation
564   factoryTrim[3] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[3] - 1.0) ));
565   // FT[Yg] factory trim calculation
566   factoryTrim[4] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[4] - 1.0) ));
567   // FT[Zg] factory trim calculation
568   factoryTrim[5] = (float)(2620 / 1 << FS) * (pow(1.01 , ((float)
          selfTest[5] - 1.0) ));
569
570   // Report results as a ratio of (STR - FT)/FT; the change from
          Factory Trim
571   // of the Self-Test Response
572   // To get percent, must multiply by 100
573   for (int i = 0; i < 3; i++)
574   {
575     // Report percent differences
576     destination[i] = 100.0 * ((float)(aSTAvg[i] - aAvg[i])) /
            factoryTrim[i]
577                         - 100.;
578     // Report percent differences
579     destination[i + 3] = 100.0 * ((float)(gSTAvg[i] - gAvg[i])) /
             factoryTrim[i + 3]
580                             - 100.;
581   }
582 }
583
584 // Function which accumulates magnetometer data after device
       initialization.
585 // It calculates the bias and scale in the x, y, and z axes.
```

```
586 void MPU9250::magCalMPU9250(float * bias_dest, float * scale_dest
    ) {
587   uint16_t ii = 0, sample_count = 0;
588   int32_t mag_bias[3]  = {0, 0, 0},
589                           mag_scale[3] = {0, 0, 0};
590   int16_t mag_max[3]  = {0x8000, 0x8000, 0x8000},
591                          mag_min[3]  = {0x7FFF, 0x7FFF, 0x7FFF},
592                                        mag_temp[3] = {0, 0, 0};
593
594   // Make sure resolution has been calculated
595   getMres();
596
597   // Serial.println(F("Mag Calibration: Wave device in a figure 8
        until done!"));
598   // Serial.println(F("  4 seconds to get ready followed by 15
      seconds of sampling)"));
599   delay(4000);
600
601   // shoot for ~fifteen seconds of mag data
602   // at 8 Hz ODR, new mag data is available every 125 ms
603   if (Mmode == M_8HZ) {
604     sample_count = 128;
605   }
606   // at 100 Hz ODR, new mag data is available every 10 ms
607   if (Mmode == M_100HZ) {
608     sample_count = 1500;
609   }
610
611   for (ii = 0; ii < sample_count; ii++) {
612     readMagData(mag_temp);  // Read the mag data
613     for (int jj = 0; jj < 3; jj++) {
614       if (mag_temp[jj] > mag_max[jj]) {
615         mag_max[jj] = mag_temp[jj];
616       }
617       if (mag_temp[jj] < mag_min[jj]) {
618         mag_min[jj] = mag_temp[jj];
619       }
620     }
621     if (Mmode == M_8HZ) {
```

```
622       delay(135); // At 8 Hz ODR, new mag data is available every
              125 ms
623     }
624     if (Mmode == M_100HZ) {
625       delay(12);  // At 100 Hz ODR, new mag data is available
              every 10 ms
626     }
627   }
628
629   // Serial.println("mag x min/max:"); Serial.println(mag_max[0])
      ; Serial.println(mag_min[0]);
630   // Serial.println("mag y min/max:"); Serial.println(mag_max[1])
      ; Serial.println(mag_min[1]);
631   // Serial.println("mag z min/max:"); Serial.println(mag_max[2])
      ; Serial.println(mag_min[2]);
632
633   // Get hard iron correction
634   // Get 'average' x mag bias in counts
635   mag_bias[0]  = (mag_max[0] + mag_min[0]) / 2;
636   // Get 'average' y mag bias in counts
637   mag_bias[1]  = (mag_max[1] + mag_min[1]) / 2;
638   // Get 'average' z mag bias in counts
639   mag_bias[2]  = (mag_max[2] + mag_min[2]) / 2;
640
641   // Save mag biases in G for main program
642   bias_dest[0] = (float)mag_bias[0] * mRes *
      factoryMagCalibration[0];
643   bias_dest[1] = (float)mag_bias[1] * mRes *
      factoryMagCalibration[1];
644   bias_dest[2] = (float)mag_bias[2] * mRes *
      factoryMagCalibration[2];
645
646   // Get soft iron correction estimate
647   // Get average x axis max chord length in counts
648   mag_scale[0]  = (mag_max[0] - mag_min[0]) / 2;
649   // Get average y axis max chord length in counts
650   mag_scale[1]  = (mag_max[1] - mag_min[1]) / 2;
651   // Get average z axis max chord length in counts
652   mag_scale[2]  = (mag_max[2] - mag_min[2]) / 2;
653
```

```
654   float avg_rad = mag_scale[0] + mag_scale[1] + mag_scale[2];
655   avg_rad /= 3.0;
656
657   scale_dest[0] = avg_rad / ((float)mag_scale[0]);
658   scale_dest[1] = avg_rad / ((float)mag_scale[1]);
659   scale_dest[2] = avg_rad / ((float)mag_scale[2]);
660
661   // Serial.println(F("Mag Calibration done!"));
662 }
663 // Wire.h read and write protocols
664 uint8_t MPU9250::writeByte(uint8_t deviceAddress, uint8_t
       registerAddress, uint8_t data) {
665   if (_csPin != NOT_SPI) {
666     return writeByteSPI(registerAddress, data);
667   } else {
668     return writeByteWire(deviceAddress, registerAddress, data);
669   }
670 }
671
672 uint8_t MPU9250::writeByteSPI(uint8_t registerAddress, uint8_t
       writeData) {
673   uint8_t returnVal;
674
675 SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST,
       SPI_MODE));
676   select();
677
678   SPI.transfer(registerAddress);
679   returnVal = SPI.transfer(writeData);
680
681   deselect();
682   SPI.endTransaction();
683 #ifdef SERIAL_DEBUG
684   Serial.print("MPU9250::writeByteSPI slave returned: 0x");
685   Serial.println(returnVal, HEX);
686 #endif
687   return returnVal;
688 }
689
```

```
690 uint8_t MPU9250::writeByteWire(uint8_t deviceAddress, uint8_t
       registerAddress, uint8_t data) {
691   Wire.beginTransmission(deviceAddress);  // Initialize the Tx
        buffer
692   Wire.write(registerAddress);       // Put slave register address
         in Tx buffer
693   Wire.write(data);                  // Put data in Tx buffer
694   Wire.endTransmission();            // Send the Tx buffer
695   // TODO: Fix this to return something meaningful
696   return NULL;
697 }
698
699 // Read a byte from given register on device. Calls necessary SPI
       or I2C
700 // implementation. This was configured in the constructor.
701 uint8_t MPU9250::readByte(uint8_t deviceAddress, uint8_t
       registerAddress) {
702   if (_csPin != NOT_SPI) {
703     return readByteSPI(registerAddress);
704   } else {
705     return readByteWire(deviceAddress, registerAddress);
706   }
707 }
708
709 // Read a byte from the given register address from device using
       I2C
710 uint8_t MPU9250::readByteWire(uint8_t deviceAddress, uint8_t
       registerAddress) {
711   uint8_t data; // `data` will store the register data
712
713   // Initialize the Tx buffer
714   Wire.beginTransmission(deviceAddress);
715   // Put slave register address in Tx buffer
716   Wire.write(registerAddress);
717   // Send the Tx buffer, but send a restart to keep connection
        alive
718   Wire.endTransmission(false);
719   // Read one byte from slave register address
720   Wire.requestFrom(deviceAddress, (uint8_t) 1);
721   // Fill Rx buffer with result
```

```
722   data = Wire.read();
723   // Return data read from slave register
724   return data;
725 }
726
727 // Read a byte from the given register address using SPI
728 uint8_t MPU9250::readByteSPI(uint8_t registerAddress) {
729   return writeByteSPI(registerAddress | READ_FLAG, 0xFF /*0xFF is
          arbitrary*/);
730 }
731
732 // Read 1 or more bytes from given register and device using I2C
733 uint8_t MPU9250::readBytesWire(uint8_t deviceAddress, uint8_t
       registerAddress, uint8_t count, uint8_t * dest) {
734   // Initialize the Tx buffer
735   Wire.beginTransmission(deviceAddress);
736   // Put slave register address in Tx buffer
737   Wire.write(registerAddress);
738   // Send the Tx buffer, but send a restart to keep connection
         alive
739   Wire.endTransmission(false);
740
741   uint8_t i = 0;
742   // Read bytes from slave register address
743   Wire.requestFrom(deviceAddress, count);
744   while (Wire.available()) {
745     // Put read results in the Rx buffer
746     dest[i++] = Wire.read();
747   }
748
749   return i; // Return number of bytes written
750 }
751
752 // Select slave IC by asserting CS pin
753 void MPU9250::select() {
754   digitalWrite(_csPin, LOW);
755 }
756
757 // Select slave IC by deasserting CS pin
758 void MPU9250::deselect() {
```

```
759   digitalWrite(_csPin, HIGH);
760 }
761
762 uint8_t MPU9250::readBytesSPI(uint8_t registerAddress, uint8_t
      count, uint8_t * dest) {
763 SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST,
      SPI_MODE));
764   select();
765
766   SPI.transfer(registerAddress | READ_FLAG);
767
768   uint8_t i;
769
770   for (i = 0; i < count; i++) {
771     dest[i] = SPI.transfer(0x00);
772 #ifdef SERIAL_DEBUG
773     Serial.print("readBytesSPI::Read byte: 0x");
774     Serial.println(dest[i], HEX);
775 #endif
776   }
777
778   SPI.endTransaction();
779   deselect();
780   delayMicroseconds(50);
781   return i; // Return number of bytes written
782
783
784 /*  #ifdef SERIAL_DEBUG
785     Serial.print("MPU9250::writeByteSPI slave returned: 0x");
786     Serial.println(returnVal, HEX);
787     #endif
788     return returnVal;   */
789   /*
790     // Set slave address of AK8963 and set AK8963 for read
791     writeByteSPI(I2C_SLV0_ADDR, AK8963_ADDRESS | READ_FLAG);
792
793     Serial.print("\nBHW::I2C_SLV0_ADDR set to: 0x");
794     Serial.println(readByte(MPU9250_ADDRESS, I2C_SLV0_ADDR), HEX)
        ;
795
```

```
796     // Set address to start read from
797     writeByteSPI(I2C_SLV0_REG, registerAddress);
798     // Read bytes from magnetometer
799     //
800     Serial.print("\nBHW::I2C_SLV0_CTRL gets 0x");
801     Serial.println(READ_FLAG | count, HEX);
802
803     // Read count bytes from registerAddress via I2C_SLV0
804     Serial.print("BHW::readBytesSPI: return value test: ");
805     Serial.println(writeByteSPI(I2C_SLV0_CTRL, READ_FLAG | count)
          );
806   */
807 }
808
809 uint8_t MPU9250::readBytes(uint8_t deviceAddress, uint8_t
      registerAddress, uint8_t count, uint8_t * dest) {
810   if (_csPin == NOT_SPI)  // Read via I2C {
811     return readBytesWire(deviceAddress, registerAddress, count,
          dest);
812   } else  // Read using SPI {
813     return readBytesSPI(registerAddress, count, dest);
814   }
815 }
816
817 bool MPU9250::magInit() {
818   // Reset registers to defaults, bit auto clears
819   writeByteSPI(0x6B, 0x80);
820   // Auto select the best available clock source
821   writeByteSPI(0x6B, 0x01);
822   // Enable X,Y, & Z axes of accel and gyro
823   writeByteSPI(0x6C, 0x00);
824   // Config disable FSYNC pin, set gyro/temp bandwidth to 184/188
        Hz
825   writeByteSPI(0x1A, 0x01);
826   // Self tests off, gyro set to +/-2000 dps FS
827   writeByteSPI(0x1B, 0x18);
828   // Self test off, accel set to +/- 8g FS
829   writeByteSPI(0x1C, 0x08);
830   // Bypass DLPF and set accel bandwidth to 184 Hz
831   writeByteSPI(0x1D, 0x09);
```

```
832   // Configure INT pin (active high / push-pull / latch until
        read)
833   writeByteSPI(0x37, 0x30);
834   // Enable I2C master mode
835   // TODO Why not do this 11-100 ms after power up?
836   writeByteSPI(0x6A, 0x20);
837   // Disable multi-master and set I2C master clock to 400 kHz
838   //https://developer.mbed.org/users/kylongmu/code/MPU9250_SPI/
        calls says
839   // enabled multi-master... TODO Find out why
840   writeByteSPI(0x24, 0x0D);
841   // Set to write to slave address 0x0C
842   writeByteSPI(0x25, 0x0C);
843   // Point save 0 register at AK8963's control 2 (soft reset)
        register
844   writeByteSPI(0x26, 0x0B);
845   // Send 0x01 to AK8963 via slave 0 to trigger a soft restart
846   writeByteSPI(0x63, 0x01);
847   // Enable simple 1-byte I2C reads from slave 0
848   writeByteSPI(0x27, 0x81);
849   // Point save 0 register at AK8963's control 1 (mode) register
850   writeByteSPI(0x26, 0x0A);
851   // 16-bit continuous measurement mode 1
852   writeByteSPI(0x63, 0x12);
853   // Enable simple 1-byte I2C reads from slave 0
854   writeByteSPI(0x27, 0x81);
855
856   // TODO: Remove this code
857   uint8_t ret = ak8963WhoAmI_SPI();
858 #ifdef SERIAL_DEBUG
859   Serial.print("MPU9250::magInit to return ");
860   Serial.println((ret == 0x48) ? "true" : "false");
861 #endif
862   return ret == 0x48;
863 }
864
865 // Write a null byte w/o CS assertion to get SPI hardware to idle
      high (mode 3)
866 void MPU9250::kickHardware() {
```

```
867 SPI.beginTransaction(SPISettings(SPI_DATA_RATE, MSBFIRST,
        SPI_MODE));
868   SPI.transfer(0x00); // Send null byte
869   SPI.endTransaction();
870 }
871
872 bool MPU9250::begin() {
873   kickHardware();
874   return magInit();
875 }
876
877 // Read the WHOAMI (WIA) register of the AK8963
878 // TODO: This method has side effects
879 uint8_t MPU9250::ak8963WhoAmI_SPI() {
880   uint8_t response, oldSlaveAddress, oldSlaveRegister,
          oldSlaveConfig;
881   // Save state
882   oldSlaveAddress  = readByteSPI(I2C_SLV0_ADDR);
883   oldSlaveRegister = readByteSPI(I2C_SLV0_REG);
884   oldSlaveConfig   = readByteSPI(I2C_SLV0_CTRL);
885 #ifdef SERIAL_DEBUG
886   Serial.print("Old slave address: 0x");
887   Serial.println(oldSlaveAddress, HEX);
888   Serial.print("Old slave register: 0x");
889   Serial.println(oldSlaveRegister, HEX);
890   Serial.print("Old slave config: 0x");
891   Serial.println(oldSlaveConfig, HEX);
892 #endif
893
894   // Set the I2C slave addres of AK8963 and set for read
895   response = writeByteSPI(I2C_SLV0_ADDR, AK8963_ADDRESS |
          READ_FLAG);
896   // I2C slave 0 register address from where to begin data
          transfer
897   response = writeByteSPI(I2C_SLV0_REG, 0x00);
898   // Enable 1-byte reads on slave 0
899   response = writeByteSPI(I2C_SLV0_CTRL, 0x81);
900   delayMicroseconds(1);
901   // Read WIA register
902   response = writeByteSPI(WHO_AM_I_AK8963 | READ_FLAG, 0x00);
```

```
903
904   // Restore state
905   writeByteSPI(I2C_SLV0_ADDR, oldSlaveAddress);
906   writeByteSPI(I2C_SLV0_REG, oldSlaveRegister);
907   writeByteSPI(I2C_SLV0_CTRL, oldSlaveConfig);
908
909   return response;
910 }
911
912 bool MPU9250::writeCalibration(const void *data) {
913   const uint8_t *p = (const uint8_t *)data;
914   uint16_t crc;
915   uint8_t i;
916
917   if (p[0] != 117 || p[1] != 84) return false;
918   crc = 0xFFFF;
919   for (i = 0; i < MPU9250_CAL_SIZE; i++) {
920     crc = _crc16_update(crc, p[i]);
921   }
922   if (crc != 0) return false;
923   for (i = 0; i < MPU9250_CAL_SIZE; i++) {
924     EEPROM.write(MPU9250_CAL_EEADDR + i, p[i]);
925   }
926   for (i = 0; i < MPU9250_CAL_SIZE; i++) {
927     if (EEPROM.read(MPU9250_CAL_EEADDR + i) != p[i]) return false
        ;
928   }
929   memcpy(cal, ((const uint8_t *)data) + 2, sizeof(cal));
930   return true;
931 }
932 void MPU9250::getCalibration(float *offsets, float *softiron =
    NULL, float *fieldstrength = NULL) {
933   if (offsets != NULL) {
934     memcpy(offsets, cal, 36);
935   }
936   if (softiron != NULL) {
937     *softiron++ = cal[10];
938     *softiron++ = cal[13];
939     *softiron++ = cal[14];
940     *softiron++ = cal[13];
```

```
941       *softiron++ = cal[11];
942       *softiron++ = cal[15];
943       *softiron++ = cal[14];
944       *softiron++ = cal[15];
945       *softiron++ = cal[12];
946     }
947     if (fieldstrength != NULL) *fieldstrength = cal[9];
948 }
```

A.4 Filtro Madgwick

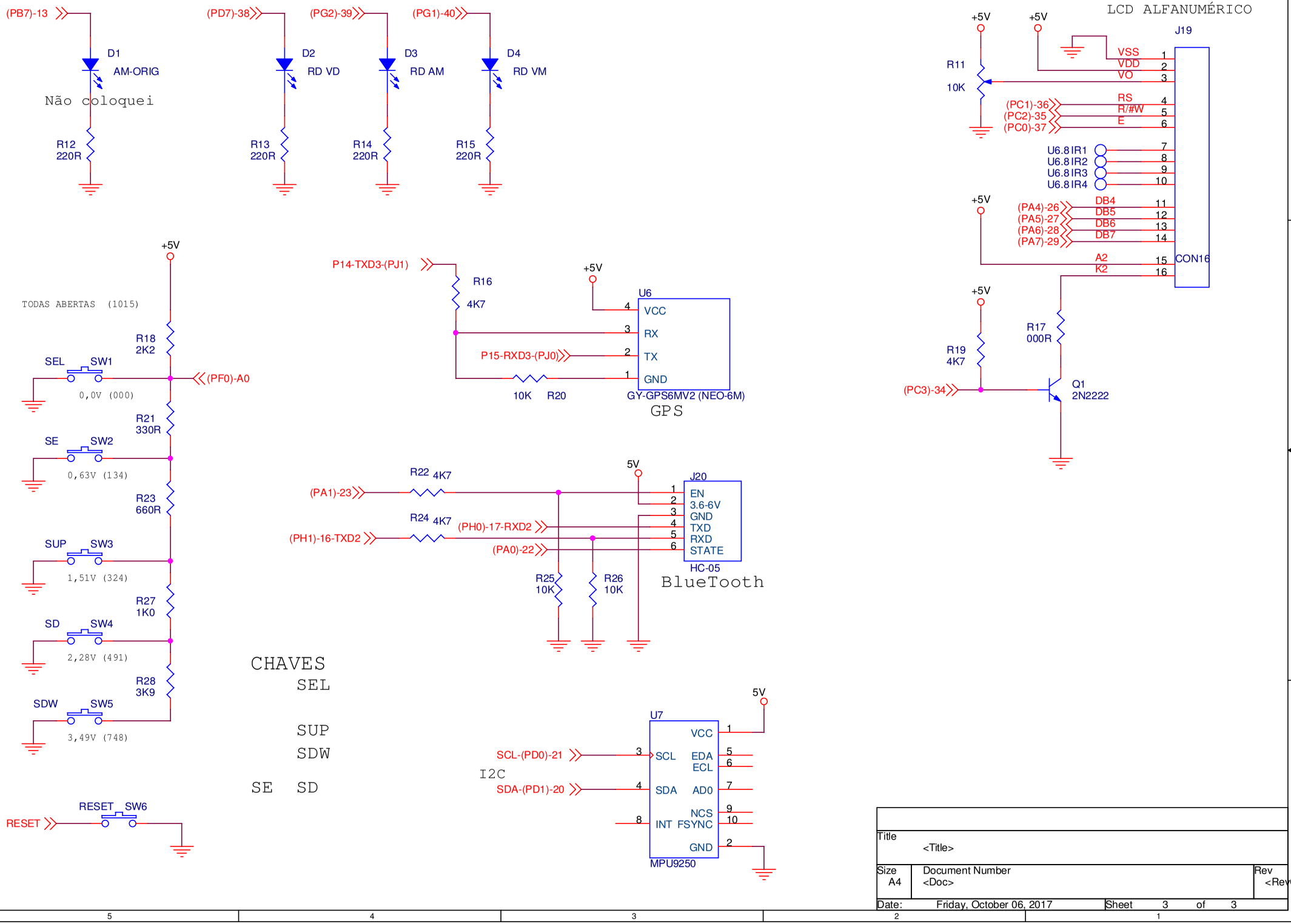
```
/* https://github.com/kriswiner/MPU9250/blob/master/
   quaternionFilters.ino
        Implementation of Sebastian Madgwick's "...efficient
           orientation filter for...inertial/magnetic sensor
           arrays" */
void MadgwickQuaternionUpdate(float ax, float ay, float az, float
    gx, float gy, float gz) {
  // short name local variable for readability
  float q1 = q[0], q2 = q[1], q3 = q[2], q4 = q[3];
  float norm;                    // vector norm
  float f1, f2, f3;              // objetive funcyion elements
  // objective function Jacobian elements
  float J_11or24, J_12or23, J_13or22, J_14or21, J_32, J_33;
  float qDot1, qDot2, qDot3, qDot4;
  float hatDot1, hatDot2, hatDot3, hatDot4;
  // gyro bias error
  float gerrx, gerry, gerrz, gbiasx, gbiasy, gbiasz;

  // Auxiliary variables to avoid repeated arithmetic
  float _halfq1 = 0.5f * q1;
  float _halfq2 = 0.5f * q2;
  float _halfq3 = 0.5f * q3;
  float _halfq4 = 0.5f * q4;
  float _2q1 = 2.0f * q1;
  float _2q2 = 2.0f * q2;
  float _2q3 = 2.0f * q3;
  float _2q4 = 2.0f * q4;
  float _2q1q3 = 2.0f * q1 * q3;
  float _2q3q4 = 2.0f * q3 * q4;

  // Normalise accelerometer measurement
  norm = sqrt(ax * ax + ay * ay + az * az);
  if (norm == 0.0f) return; // handle NaN
  norm = 1.0f/norm;
  ax *= norm;
  ay *= norm;
  az *= norm;

```

```
35  // Compute the objective function and Jacobian
36  f1 = _2q2 * q4 - _2q1 * q3 - ax;
37  f2 = _2q1 * q2 + _2q3 * q4 - ay;
38  f3 = 1.0f - _2q2 * q2 - _2q3 * q3 - az;
39  J_11or24 = _2q3;
40  J_12or23 = _2q4;
41  J_13or22 = _2q1;
42  J_14or21 = _2q2;
43  J_32 = 2.0f * J_14or21;
44  J_33 = 2.0f * J_11or24;
45
46  // Compute the gradient (matrix multiplication)
47  hatDot1 = J_14or21 * f2 - J_11or24 * f1;
48  hatDot2 = J_12or23 * f1 + J_13or22 * f2 - J_32 * f3;
49  hatDot3 = J_12or23 * f2 - J_33 *f3 - J_13or22 * f1;
50  hatDot4 = J_14or21 * f1 + J_11or24 * f2;
51
52  // Normalize the gradient
53  norm = sqrt(hatDot1 * hatDot1 + hatDot2 * hatDot2 + hatDot3 *
       hatDot3 + hatDot4 * hatDot4);
54  hatDot1 /= norm;
55  hatDot2 /= norm;
56  hatDot3 /= norm;
57  hatDot4 /= norm;
58
59  // Compute estimated gyroscope biases
60  gerrx = _2q1 * hatDot2 - _2q2 * hatDot1 - _2q3 * hatDot4 + _2q4
        * hatDot3;
61  gerry = _2q1 * hatDot3 + _2q2 * hatDot4 - _2q3 * hatDot1 - _2q4
        * hatDot2;
62  gerrz = _2q1 * hatDot4 - _2q2 * hatDot3 + _2q3 * hatDot2 - _2q4
        * hatDot1;
63
64  // Compute and remove gyroscope biases
65  gbiasx += gerrx * mpu.deltat * zeta;
66  gbiasy += gerry * mpu.deltat * zeta;
67  gbiasz += gerrz * mpu.deltat * zeta;
68  gx -= gbiasx;
69  gy -= gbiasy;
70  gz -= gbiasz;
```

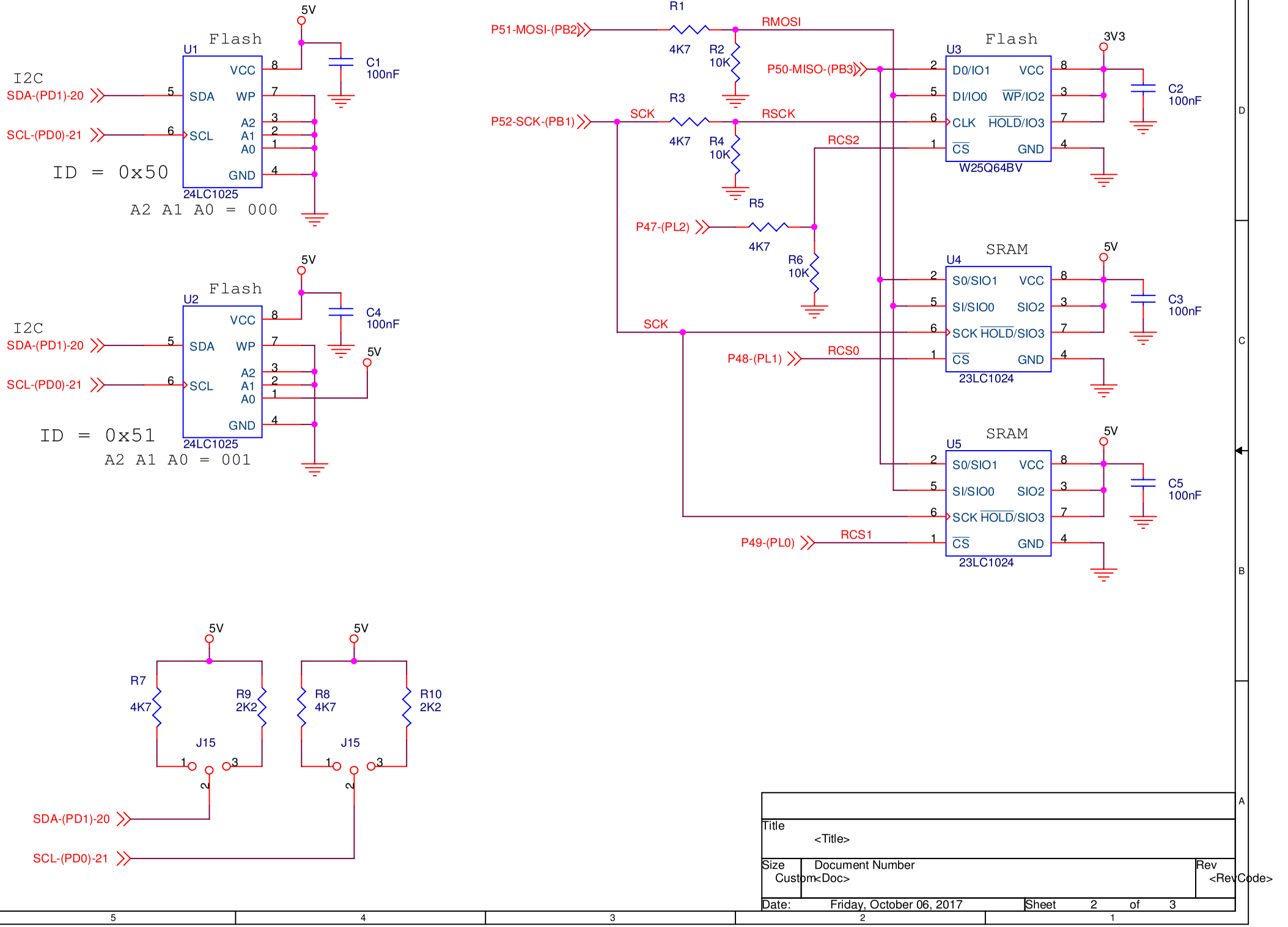
```
71
72    // Compute the quaternion derivative
73    qDot1 = -_halfq2 * gx - _halfq3 * gy - _halfq4 * gz;
74    qDot2 =  _halfq1 * gx + _halfq3 * gz - _halfq4 * gy;
75    qDot3 =  _halfq1 * gy - _halfq2 * gz + _halfq4 * gx;
76    qDot4 =  _halfq1 * gz + _halfq2 * gy - _halfq3 * gx;
77
78    // Compute then integrate estimated quaternion derivative
79    q1 += (qDot1 -(beta * hatDot1)) * mpu.deltat;
80    q2 += (qDot2 -(beta * hatDot2)) * mpu.deltat;
81    q3 += (qDot3 -(beta * hatDot3)) * mpu.deltat;
82    q4 += (qDot4 -(beta * hatDot4)) * mpu.deltat;
83
84    // Normalize the quaternion
85    norm = sqrt(q1 * q1 + q2 * q2 + q3 * q3 + q4 * q4);    //
        normalise quaternion
86    norm = 1.0f/norm;
87    q[0] = q1 * norm;
88    q[1] = q2 * norm;
89    q[2] = q3 * norm;
90    q[3] = q4 * norm;
91  }
```

Apêndice B

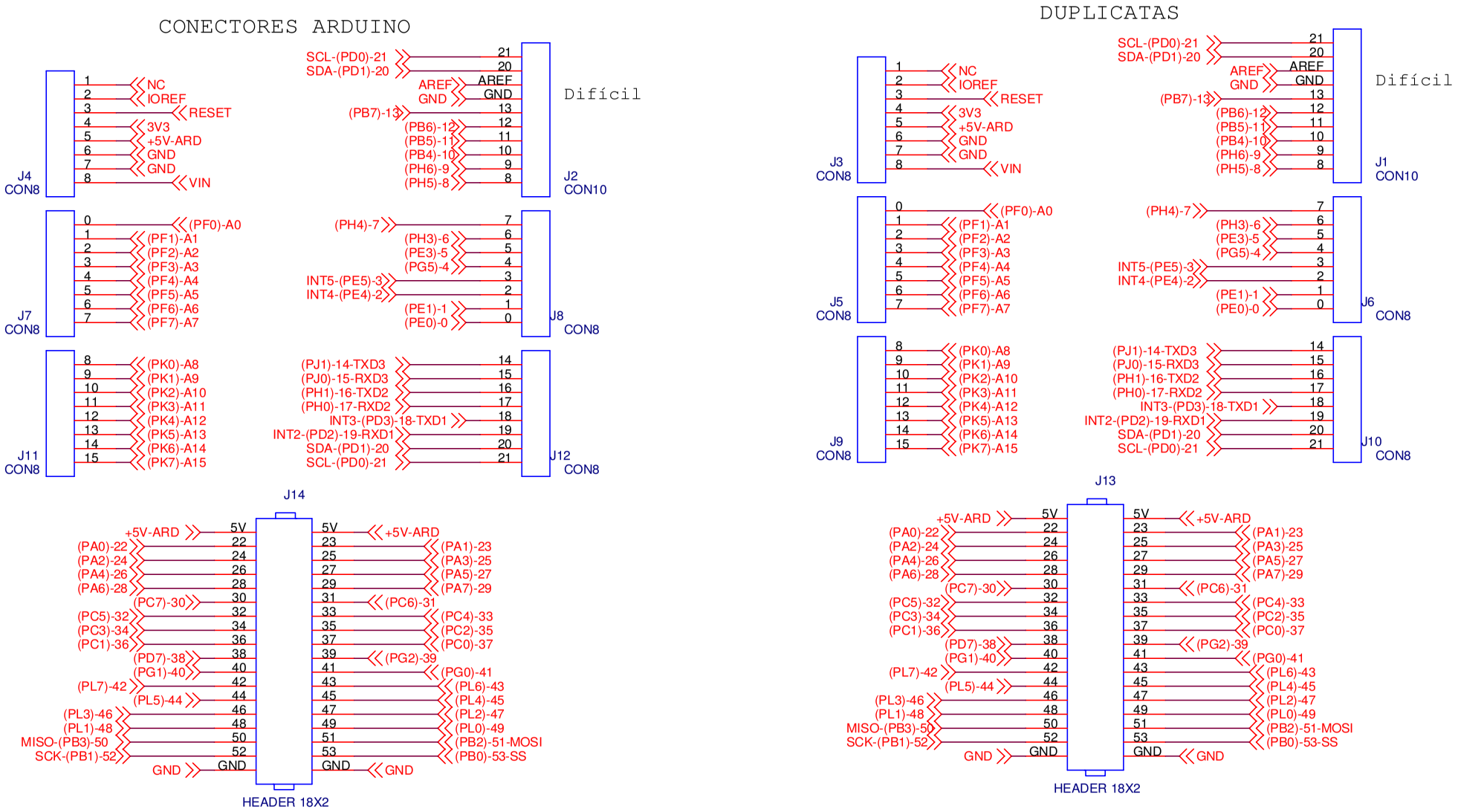
Esquemáticos



Title	<Title>	
Size A4	Document Number <Doc>	Rev <RevCode>
Date: Friday, October 06, 2017	Sheet 3 of 3	



Title <Title>		
Size Custom	Document Number <Doc>	Rev <RevCode>
Date: Friday, October 06, 2017	Sheet 2 of 3	



Title	<Title>	
Size A4	Document Number <Doc>	Rev <RevCode>
Date: Monday, October 02, 2017	Sheet 1 of 3	