Fundação Universidade Federal do ABC

SISTEMA DE DETERMINAÇÃO DE ATITUDE UTILIZANDO ACELERÔMETRO, GIROSCÓPIO

Bruno Bevilacqua Nascimento

Orientador: Luiz S. Martins Filho

Agosto, 2018

Fundação Universidade Federal do ABC

SISTEMA DE DETERMINAÇÃO DE ATITUDE UTILIZANDO ACELERÔMETRO, GIROSCÓPIO

O projeto em questão tem como objetivo apresentar uma proposta de pesquisa no ramo da Engenharia Aeroespacial, buscando definir métodos e materiais a serem utilizados, além da base teórica que sustenta o tema trabalhado.

Orientador: Luiz S. Martins Filho

Agosto, 2018

RESUMO

Este trabalho trata do desenvolvimento de uma plataforma experimental de estimação da atitude para veículos aeroespaciais de pequenas dimensões. As aplicações de maior interesse são nanossatélites (Cubesats), e veículos aéreos não tripulados (VANTs quadrirrotores). A modelagem do movimento do veículo adota o exemplo de um nanossatélite, mas pode ser facilmente adaptado ao caso de um VANT. O sistema de determinação da atitude adota um dispositivo do tipo MEMS contendo sensores triaxiais (magnetômetro, acelerômetro, giroscópio). A estimação de atitude usa algoritmos de 2 tipos: os que consideram as informações em um dado instante, e os que consideram a dinâmica de movimento para integrar informações de um dado instante e aquelas propagadas de um instante anterior usando o modelo dessa dinâmica. Os algoritmos são implantados num dispositivo processador na forma de um sistema embarcado/dedicado.

ABSTRACT

This work deals with the development of an experimental attitude estimation platform for small aerospace vehicles. The most interesting applications are nanossatellites (Cubesats), and unmanned aerial vehicles (UAVs). Vehicle motion modeling takes the example of a nanosatellite, but can be easily adapted to the case of an UAV. The attitude determination system adopts a MEMS type device containing triaxial sensors (magnetometer, accelerometer, gyroscope). Attitude estimation uses algorithms of two types: those that consider the information at a given instant, and those that consider the dynamics of movement to integrate information of a given instant and those propagated from an earlier instant using the model of this dynamic. The algorithms are deployed in a processor device in the form of an embedded / dedicated system.

Sumário

1. INTRODUÇÃO

1.1. Tema e Delimitação

Este trabalho trata do desenvolvimento de uma plataforma experimental de estimação da atitude visando aplicação num veículo aeroespacial de pequenas dimensões. A atitude é uma denominação do setor aeroespacial para a orientação de um objeto (satélite, espaçonaves, etc.) em relação a algum referencial de interesse, como o referencial orbital ou o referencial inercial centrado na Terra. As aplicações de maior interesse desse estudo são em nanossatélites (como os Cubesats), e em veículos aéreos não tripulados (VANTs) do tipo quadrirrotores. A modelagem do movimento do veículo adota o exemplo de um nanossatélite, mas pode ser facilmente adaptado ao caso de um VANT quadrirrotor.

O sistema de determinação da atitude adota um dispositivo do tipo MEMS contendo sensores triaxiais (magnetômetro, giroscópio e acelerômetro). A estimação de atitude usará algoritmos de 2 tipos: os que consideram as informações de sensores em um dado instante (campo magnético terrestre – magnetômetros, e campo gravitacional – acelerômetros), e os que consideram a dinâmica de movimento para integrar informações de um dado instante e aquelas propagadas de um instante anterior usando o modelo dessa dinâmica.

Os algoritmos do primeiro tipo, aqui denominados de estimação estática, são: método TRIAD, método Q, e algoritmo QUEST (Shuster & Oh, 1981; Hall, 2003). O segundo tipo pode usar o resultado da estimação estática para combinação com as informações provindas do modelo do movimento através do estimador bayesiano ótimo Filtro de Kalman (Lefferts et al., 1982).

A Engenharia Aeroespacial da UFABC dispõe de duas unidades do VANT modelo Gyro 200 ED, fabricado pela empresa Gyrofly (Fig. 1), que desenvolve e comercializa VANTs para serviços de segurança e para outras aplicações baseadas na captura de imagens. Ela desenvolveu uma versão para fins acadêmicos, ensino e pesquisa, com características que permitem acessar dados de sensores e até modificar sua programação (GYROFLY, 2016).

Além disso, a UFABC participa de iniciativas no projeto NanoSatC-BR2, que é um satélite do programa do Instituto Nacional de Pesquisas Espaciais (INPE), em conjunto com a Universidade Federal de Santa Maria (UFSM), para o desenvolvimento de nanossatélites (NANOSATC-BR, 2016; Garcia et al., 2016).

A UFABC, a Universidade Federal de Minas Gerais (UFMG) e o INPE estão cooperando para a preparação de um experimento embarcado (carga útil). Esse experimento é um sistema de determinação de atitude baseado em dados de sensores (magnetômetros e sensor solar), que adota uma estratégia de tolerância a falhas (múltiplos processadores, algoritmos de checagem de erros e falhas). As atividades da equipe da UFABC consistem de preparação e testes de algoritmos de determinação de atitude e de cálculo de campo magnético terrestre e posição relativa do Sol (Garcia et al., 2016). A Fig. 2 mostra o modelo de engenharia do NanoSatC-BR2, e a Fig. 3 mostra o sistema de determinação de atitude tolerante a falhas (SDATF) desenvolvido por UFMG-UFABC-INPE.

Figura 1: VANT quadrirrotor Gyro 200 ED (GYROFLY).

Figura 2: Cubesat NanosatC-BR2, que tem carga útil em desenvolvimento conjunto do INPE, da UFMG e da UFABC (Durão & Essado, 2014).

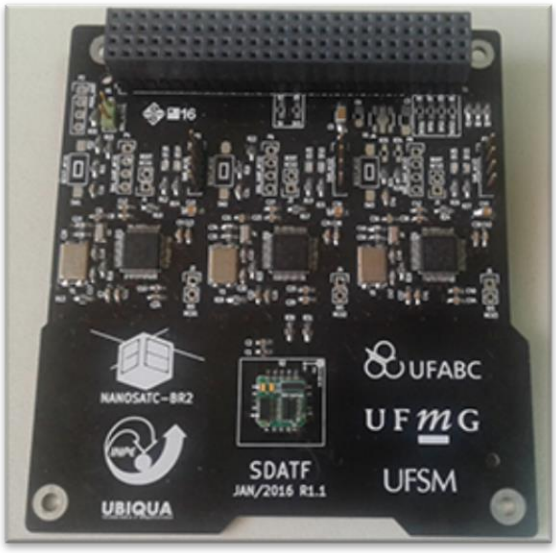


Figura 3: O Sistema de Determinação de Atitude Tolerante a Falhas (SDATF), desenvolvido por INPE, UFMG e UFABC (Garcia et al., 2016).

1.2. Problema de Pesquisa

O projeto em questão busca avaliar a viabilidade da implementação de um Sistema de Determinação de Atitude utilizando dispositivo MEMS. Aspectos como custo benefício, precisão dos dados obtidos e eficiência do aparato serão estudados com cuidado a fim de que se chegue em uma conclusão válida sobre a viabilidade de tal sistema.

1.3. Objetivos

1.3.1. Objetivo Principal

Estudar e implantar um sistema de determinação de atitude para veículos aeroespaciais, utilizando sensores embarcados do tipo MEMS.

1.3.2. Metas

- Revisar a modelagem do movimento de atitude de veículos aeroespaciais;
- Estudar os algoritmos de determinação de atitude;

- Desenvolver e programar algoritmos de determinação de atitude em Matlab e Linguagem C;
- Implantar os algoritmos em dispositivos de processamento com capacidade limitada em termos de memória e de velocidade.

2. REVISÃO DE LITERATURA

2.1. Sistemas Embarcados

As principais tarefas funcionais de um veículo aeroespacial são realizadas por sistemas denominados embarcados. Esses sistemas consistem basicamente de um microprocessador dedicado a tarefas específicas e que obtém como produto pacotes de informações que podem levar a ações de sistemas como os de propulsão e outros para interferir no movimento do veículo. Os processadores recebem informações de sensores e comandos de outros sistemas ou de operadores do veículo. São dotados de recursos de tratamento e adequação de dados, de memórias, e de capacidade de cálculos, dentro de limitações consideráveis de consumo de energia, de dimensões físicas, e outras. Para as aplicações em nanossatélites e VANTS, os sistemas embarcados contam com a disponibilidade de sensores e outros dispositivos de tecnologia MEMS (do inglês Micro-machined electromechanical systems).

2.2. Dispositivo MEMS

Sistemas MEMS são sistemas eletromecânicos com dimensões reduzidas de maneira a que possam ser implementados em circuitos integrados (CIs) para a composição de um dispositivo contendo sensores e processamento. Esses dispositivos são de custo muito baixo, o que os tornou tão acessíveis e tão populares em equipamentos eletrônicos de uso pessoal, como smartphones e jogos eletrônicos.

Um possível dispositivo MEMS a ser utilizado no projeto é a Unidade de Medidas Inerciais (IMU) modelo GY-80, equipada com giroscópio de 3 eixos, acelerômetro de 3 eixos, e magnetômetro de 3 eixos, além de outros recursos como sensor de pressão e temperatura, e conversor AD. Esse dispositivo é facilmente integrado numa plataforma de processamento Arduino.

Outras especificações: protocolo de comunicação I2C, Chip acelerômetro ADXL345, faixa do acelerômetro ±2, ±4, ±8, ±16g, Chip giroscópio L3G4200D, faixa do giroscópio: ±250, 500, 2000°/s, Chip magnetômetro HMC5883L, Chip barômetro BMP085, tensão de operação: 3,3-5V, peso 5g, dimensões: 25,8 x 16,8mm.

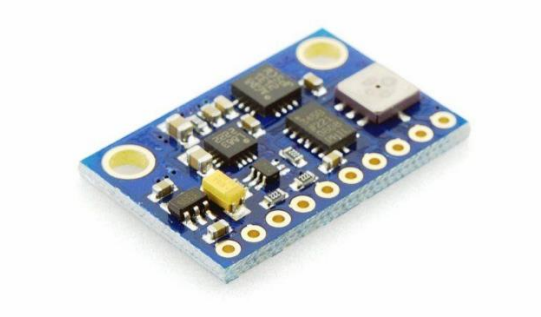


Figura 4: Dispositivo GY-80.

Outras especificações: tensão de operação 3-5V, conversor AD 16 bits, comunicação com protocolo padrão I2C, faixa do giroscópio ±250, 500, 1000, 2000°/s, faixa do acelerômetro ±2, ±4, ±8, ±16g, dimensões 20 x 16 x 1mm. O dispositivo GY-80 é mostrado na Fig. 4.

Um possível processador a ser adotado para a montagem experimental do sistema de determinação de atitude é a placa Arduino Mega 2560 (Fig. 5), que possui recursos bem interessantes para prototipagem de sistemas embarcados (ARDUINO, 2017). Essa plataforma de desenvolvimento tem como processador um microcontrolador ATmega2560. Suas especificações principais são: 54 pinos de entradas e saídas digitais (15 destes podem ser utilizados como saídas PWM), 16 entradas analógicas, 4 portas de comunicação serial, memória com 256 KB de Flash, 8 KB de RAM e 4 KB de EEPROM, protocolo de comunicação SPI, I2C e 6 pinos de interrupções externas.

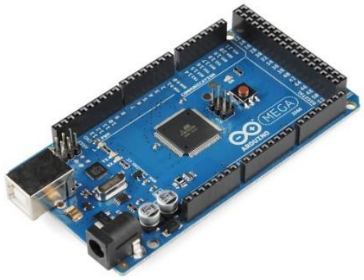


Figura 5: Plataforma Arduino Mega 2560.

2.3. Acelerômetro

O acelerômetro é um equipamento utilizado para mensurar a aceleração própria. A aceleração própria é diferente daquela estabelecida através da relação entre velocidade e tempo. Sendo que esta considera a sensação de peso medida em um dado referencial.

Acelerômetros são dispositivos que podem funcionar a partir de diversos efeitos físicos e apresenta uma extensa faixa de valores de aceleração. Esses dispositivos são utilizados principalmente em sistemas de posicionamento, sensores de inclinação e sensores de vibração.

Na Terra, quando consideramos o acelerômetro colocado em uma superfície plana, a medição será de aproximadamente 9,81 m/s^2. Na maioria dos casos, a aceleração é medida em força-g, que é basicamente a aceleração sentida como peso. Na superfície da Terra e em condições normais experimentamos 1g.

Na maioria dos casos, a aceleração é tratada como um vetor que pode ser usado para detectar a orientação do dispositivo, mais precisamente *pitch* (inclinação) e *roll* (rotação). Quando se aciona o dispositivo, a aceleração de 1g é distribuída entre os três eixos. Com isso é possível calcular o ângulo do dispositivo em cada um dos 3 eixos.

2.4. Giroscópio

Os giroscópios são utilizados para medir orientação levando em conta a velocidade angular do objeto estudado. Um giroscópio mecânico normalmente consiste de um disco rotativo, onde os eixos ligados a ele são capazes de se deslocar livremente em qualquer orientação.

Um giroscópio microeletromecânico (MEMS) é muito semelhante, mas em vez de um disco giratório, ele consiste em um tipo de ressonador vibrando. A ideia é a mesma; um objeto de vibração tende a continuar vibrando no mesmo plano que as suas bases de apoio.

Um giroscópio MEMS mede a velocidade angular, a partir do qual se pode calcular o ângulo.

2.5. Determinação de Atitude

A determinação de atitude pode considerar as informações apenas de um dado instante, ou então utilizar informações que proveem de instantes anteriores (que permitem fazer uma predição). Os métodos de determinação da atitude a partir de informações do presente podem ser divididos entre métodos determinísticos, como, por exemplo, o conhecido método TRIAD, onde eles retornam a atitude real do sistema, e os métodos estatísticos, que determinam a atitude através na otimização de funções de custo.

Alguns exemplos de métodos estatísticos são o método Q e o algoritmo QUEST (Shuster & Oh, 1981; Hall, 2003; Castro, 2006, Duarte et al, 2009).

O quaternion $\bar{q}$ é um vetor 4x1 como mostrado na Equação (1) (Hall D., 2003)

$$\bar{q} = \begin{bmatrix} \boldsymbol{q_1} \\ \boldsymbol{q_2} \\ \boldsymbol{q_3} \\ q_4 \end{bmatrix} = \begin{bmatrix} \boldsymbol{q} \\ q_4 \end{bmatrix} \quad (1)$$

A componente $\boldsymbol{q}$ é a parte vetorial e $q4$ representa a parte escalar. Essas duas componentes podem ser expressas em função do ângulo de rotação φ e do eixo de rotação **n** assim como mostrado nas Equações (2) e (3):

$$\boldsymbol{q} = \boldsymbol{n}.\sin\left(\frac{\varphi}{2}\right) \quad (2)$$

$$q_4 = cos\left(\frac{\varphi}{2}\right) \quad (3)$$

Assim, podemos escrever a matriz atitude em função dos quaternions como mostrado na Equação (4):

$$A = (q_4^2 - |\boldsymbol{q}|^2)I_{3x3} + \boldsymbol{q}\boldsymbol{q}^T - 2q_4[\boldsymbol{q}_x] \quad (4)$$

Sendo I_{3x3} uma matriz identidade 3x3 e $[\boldsymbol{q}^x]$ segundo a Equação (5).

$$[\boldsymbol{q}^x] = \begin{bmatrix} 0 & -q_3 & q_2 \\ q_3 & 0 & -q_1 \\ -q_2 & q_1 & 0 \end{bmatrix} \quad (5)$$

Também podemos escrever a matriz atitude de maneira explicita em termos das componentes do quaternion como mostrado na Equação (6):

$$A = \begin{bmatrix} q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_1q_2 + q_4q_3) & 2(q_3q_1 - q_4q_2) \\ 2(q_2q_1 - q_4q_3) & -q_1^2 - q_2^2 - q_3^2 + q_4^2 & 2(q_3q_2 + q_4q_1) \\ 2(q_3q_1 + q_4q_2) & 2(q_3q_2 - q_4q_1) & -q_1^2 - q_2^2 + q_3^2 + q_4^2 \end{bmatrix} \quad (6)$$

O método Q procura uma solução que maximiza a função ganho dada pela Equação (7):

$$g(A) = 1 - L(A) = \sum_{i=1}^{n} a_i (w_i^T A v_i) \quad (7)$$

Onde a_i é o peso associado a cada vetor i, e $\sum_{i=1}^{n} a_i = 1$. v_i são os vetores de referência e w_i são os vetores observados, sendo $\boldsymbol{n}$ os números de sensores instalados no corpo do satélite (Duarte et al, 2009).

A função custo pode ser escrita em termos de quaternions e é dada pela Equação (8):

$$g(A) = (q_4 - \boldsymbol{q}.\boldsymbol{q}) tr B^T + 2tr[\boldsymbol{q}\boldsymbol{q}^T \boldsymbol{B}^T] - 2q_4 tr[[\boldsymbol{q}^x]\boldsymbol{B}^T] \quad (8)$$

Ou $g(\bar{q}) = \bar{q}^T K \bar{q}$ onde K é dado pela Equação (9).

$$K = \begin{bmatrix} \boldsymbol{S} - \sigma \boldsymbol{I} & \boldsymbol{Z} \\ \boldsymbol{Z}^T & \sigma \end{bmatrix} \quad (9)$$

Sendo que $\sigma, S\ e\ Z$ podem ser escritos segundo está demonstrado nas Equações (10), (11) e (12), respectivamente:

$$\sigma = tr\boldsymbol{B} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} a_i v_i w_i \quad (10)$$

$$\boldsymbol{S} = \boldsymbol{B} + \boldsymbol{B}^T = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} a_i (w_i v_i^T + v_i w_i^T) \quad (11)$$

$$Z = \sum_{i=1}^{n} a_i (w_i \, x \, v_i) \quad (12)$$

Então o problema de determinar a atitude é reduzido a achar o quaternion que maximiza a Equação (13):

$$g'(\bar{q}) = \bar{q}^T K \bar{q} - \lambda \bar{q}^T \bar{q} \qquad (Função\ Custo) \qquad (13)$$

Onde λ é escolhido para satisfazer a restrição, e então a expressão se torna a Equação (14):

$$K\bar{q}_{opt} = \lambda \bar{q}_{opt} \qquad (14)$$

Isso significa que λ pode ser calculado como um autovalor de K e $\bar{q}_{opt}$ como autovetor de K. A atitude corresponde ao máximo autovalor λ e seu autovetor $\bar{q}_{opt}$, ou seja, o quaternion ótimo, fornece a atitude estimada.

Além desses dois algoritmos, durante o projeto, será estudado e testado o algoritmo denominado QUEST, proposto por Shuster e Oh (1981), que tem como objetivo minimizar a função custo.

2.6. Filtro de Kalman

O Filtro de Kalman é um estimador bayesiano ótimo (baseado no Teorema de Bayes acerca da probabilidade condicional), que trata da filtragem de ruídos aleatórios em informações através de um algoritmo que estima os estados de um sistema a partir de medidas e de previsões de modelos da dinâmica desse sistema (Maybeck, 1979).

Esse algoritmo trata de sistemas dinâmicos lineares. Mas há uma extensão para sistemas não lineares, chamado de Filtro Estendido de Kalman, que permite sua aplicação nesses casos. Essa versão do filtro perde a característica de estimador ótimo, porém, ela permanece como uma ferramenta poderosa para filtragem de incertezas nas medidas e nos modelos dinâmicos de sistemas.

O modelo do sistema dinâmico assume que o estado real no tempo k é obtido através do estado no tempo (k − 1) de acordo com a Equação (15)

$$\boldsymbol{x}_k = \boldsymbol{F}_k \boldsymbol{x}_{k-1} + \boldsymbol{B}_k \boldsymbol{u}_k + \boldsymbol{w}_k \qquad (15)$$

onde aparecem as entradas de controle $\boldsymbol{u}_k$ e o ruído do processo $\boldsymbol{w}_k$, assumido como sendo amostrado de uma distribuição normal multivariada de média zero e

covariância $\boldsymbol{Q}_k$. No tempo k, uma observação (ou medição) $\boldsymbol{z}_k$ do estado real $\boldsymbol{x}_k$ é feita de acordo com a Equação (16)

$$\boldsymbol{z}_k = \boldsymbol{H}_k \boldsymbol{x}_{k-1} + \boldsymbol{v}_k \tag{16}$$

onde $\boldsymbol{v}_k$ é o ruído da observação, assumido como sendo um ruído branco gaussiano de média zero e covariância $\boldsymbol{R}_k$. O estado inicial e os vetores de ruído a cada passo $\{\boldsymbol{x}_0,\ \boldsymbol{x}_1,\ ...,\ \boldsymbol{w}_k,\ \boldsymbol{v}_1\ ...\ \boldsymbol{v}_k\}$ são assumidos como sendo mutuamente independentes. O algoritmo é composto de duas etapas distintas, a de predição ou de propagação dos estados a partir do instante anterior de estimação, e a de correção ou de atualização da estimação através de integração de medidas realizadas no instante atual, descritas nas Equações (17) à (23). Esse processo compreende tanto as variáveis de estado **x**, quanto a matriz de covariância do estado **P**. Na notação, $\boldsymbol{x}_{n,m}$ significa estimativa no instante n, obtida com informações até o instante m. Suas equações ficam:

Etapa de predição

Predição do estado (estimativa a priori):

$$\hat{\boldsymbol{x}}_{k/k-1} = \boldsymbol{F}_k \hat{\boldsymbol{x}}_{k-1/k-1} + \boldsymbol{B}_k \boldsymbol{u}_k \tag{17}$$

Predição da covariância (estimativa a priori):

$$\boldsymbol{P}_{k/k-1} = \boldsymbol{F}_k \boldsymbol{P}_{k-1/k-1} \boldsymbol{F}_k + \boldsymbol{Q}_k \tag{18}$$

Etapa de atualização

Resíduo da medição:

$$\tilde{\boldsymbol{y}}_k = \boldsymbol{z}_k - \boldsymbol{H}_k \hat{\boldsymbol{x}}_{k/k-1} \tag{19}$$

Resíduo da covariância:

$$\boldsymbol{S}_k = \boldsymbol{H}_k \boldsymbol{P}_{k/k-1} \boldsymbol{H}_k^T + \boldsymbol{R}_k \tag{20}$$

Ganho de Kalman:

$$\boldsymbol{K}_k = \boldsymbol{P}_k \boldsymbol{H}_k^T \boldsymbol{S}_k^{-1} \tag{21}$$

Estado atualizado (estimativa a posteriori):

$$\hat{\boldsymbol{x}}_k = \hat{\boldsymbol{x}}_{k/k-1} + \boldsymbol{K}_k \tilde{\boldsymbol{y}}_k \tag{22}$$

Covariância estimada (estimativa a posteriori):

$$\boldsymbol{P}_{k/k} = (I - \boldsymbol{K}_k \boldsymbol{H}_k) \boldsymbol{P}_{k/k-1} \tag{23}$$

---

3. METODOLOGIA

3.1. Resumo Geral

A metodologia adotada para alcançar o objetivo e cumprir as metas do projeto pretende ser uma oportunidade de praticar os procedimentos clássicos de uma pesquisa científica e tecnológica. Em suma, deverá ser estudada a modelagem matemática do problema, i.e., dos procedimentos de determinação de atitude, a partir de literatura sobre o assunto (livros, artigos, relatórios técnicos); em seguida, estudar e analisar soluções já propostas para escolher quais serão testadas e avaliadas; e finalmente programar essas soluções para testes em simulações numéricas como primeira forma de avaliação e de validação.

Os algoritmos testados e otimizados deverão ser implantados em sistemas embarcados, i.e., programados em processadores dedicados para usar dados fornecidos diretamente por sensores. Os resultados de processamento de dados pelo sistema embarcado, a partir de movimentação física (mudança de orientação) deverão ser enviados a um computador para análise desses resultados numéricos e para geração de uma ferramenta de computação gráfica para visualização da orientação espacial do dispositivo experimental.

3.2. Materiais e dispositivos utilizados

- **CI MPU-6050 –** Circuito Integrado que possui em único chip um acelerômetro, um giroscópio além de possuir também um sensor de temperatura. É um sensor de 6 eixos (6 DOF – 6 degrees of freedom ou 6 graus de liberdade) fornecendo 6 valores de saída, 3 do acelerômetro, 3 do giroscópio;
- **Placa Arduino UNO –** Responsável por fazer a comunicação entre o circuito integrado (MPU-6050) e o notebook. O módulo utiliza a comunicação via interface I2C, assim a conexão com o Arduino passa a ser bastante simplificada, utilizando apenas os pinos analógicos A4 (SDA) e A5 (SCL) e a alimentação de 5v;
- **Protoboard –** Placa de ensaio com furos e conexões que viabilizam a montagem de circuitos eletrônicos experimentais;

- **Jumpers –** pequenos fios de cobre usados para a conexão dos dispositivos no protoboard, na placa Arduino, ou no módulo MPU-6050.

3.3. Montagem do Circuito Eletrônico e Obtenção de Dados

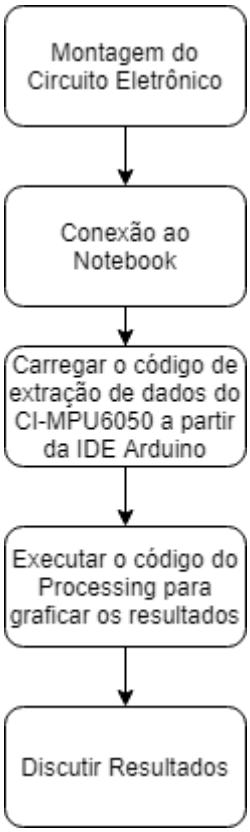


Figura 6: Fluxograma exemplificando os processos para obtenção e discussão dos resultados.

A Montagem do circuito e obtenção dos dados se deu da maneira apresentada no fluxograma da Figura 6. A princípio, monta-se o circuito de comunicação entre o Circuito Integrado, a placa Arduino UNO e o Notebook. Posteriormente, carrega-se o código de extensão “.ino” a partir da IDE Arduino (código descrito no APÊNDICE - A). Com o código gravado no processador da placa Arduino UNO, rodamos outro código de extensão “.pde” no software “Processing” para que os resultados da atitude sejam plotados em gráficos (código descrito no APÊNDICE - A).

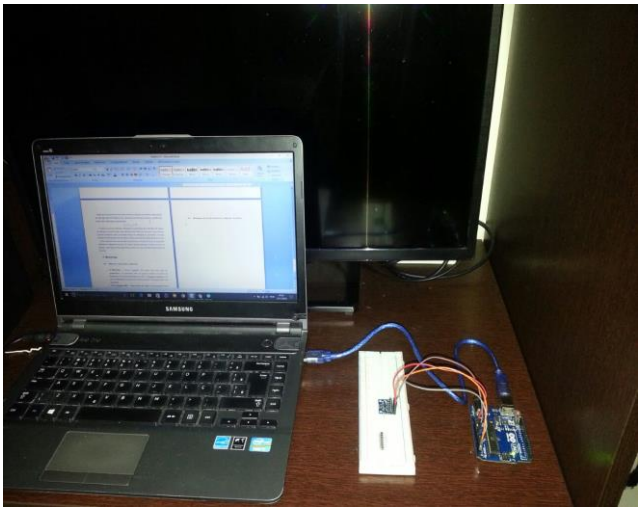


Figura 7: Circuito eletrônico montado ligado ao notebook.

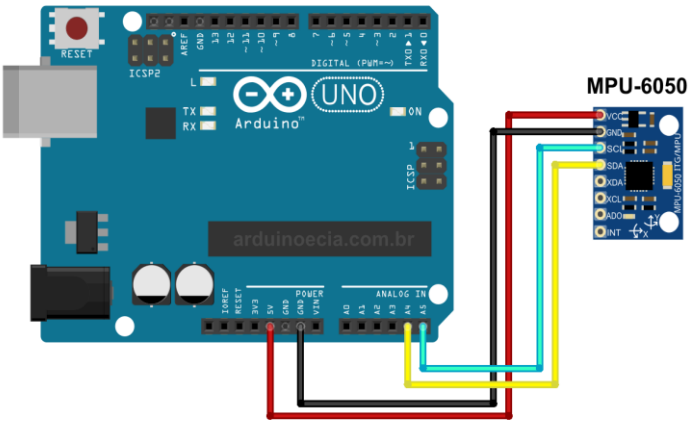


Figura 8: Esquema simplificado da montagem eletrônica do experimento.

4. RESULTADOS E DISCUSSÕES

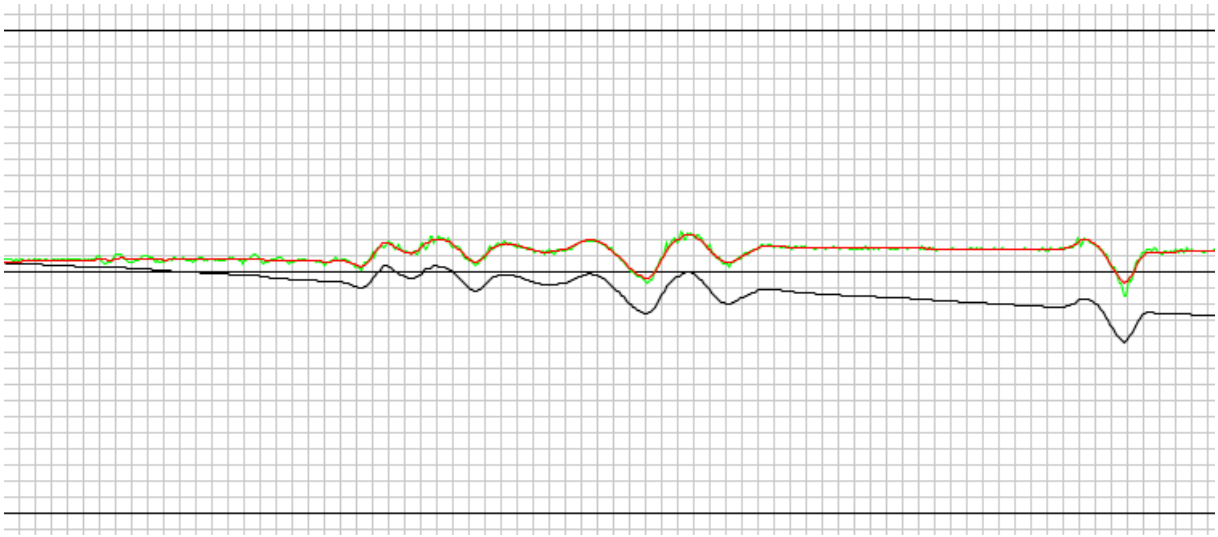
Na aquisição de dados foi feito um estudo em tempo real para o comportamento do CI tanto no eixo X quanto no eixo Y. O código utilizado no software “Processing” permitiu que o comportamento dos sensores fosse plotado em tempo real, exibindo uma comparação entre os dados fornecidos pelo giroscópio, os dados fornecidos pelo acelerômetro e os dados obtidos aplicando o Filtro de Kalman.

4.1. Análise para o Eixo X

Inicialmente foi analisado o comportamento do sensor no eixo X. O Gráfico 1 ilustra os instantes iniciais em que se iniciou a coleta de dados.

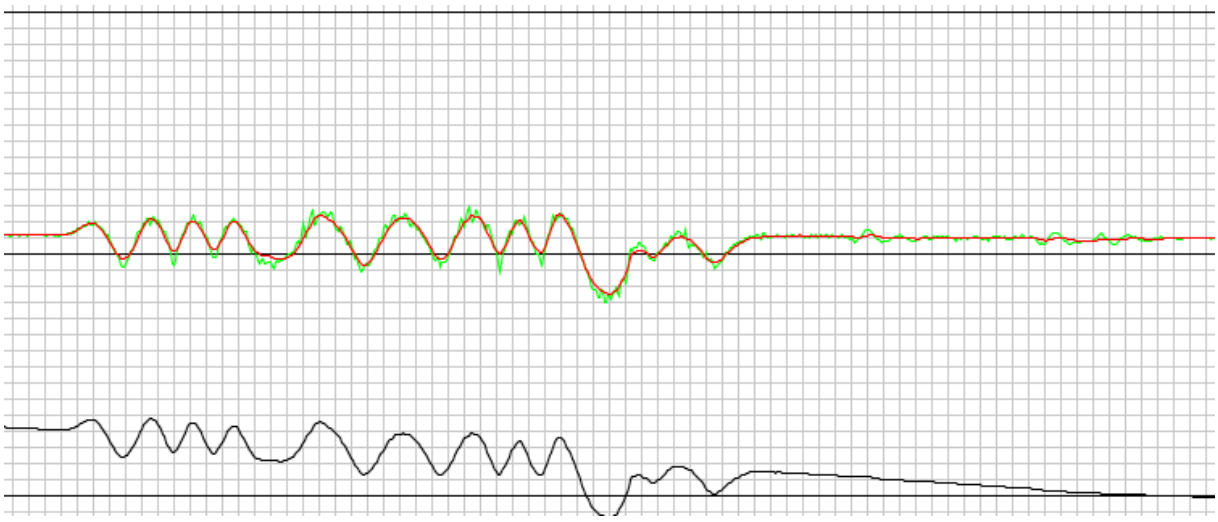
Para todos os gráficos apresentados a linha Vermelha representa a aplicação do Filtro Kalman, a linha Verde representa o acelerômetro e a linha Preta representa o Giroscópio.

GRÁFICO 1. Comportamento do sensor no eixo X (instantes iniciais)



No Gráfico 2 temos o comportamento no Eixo X 15 segundos após o início da coleta de dados.

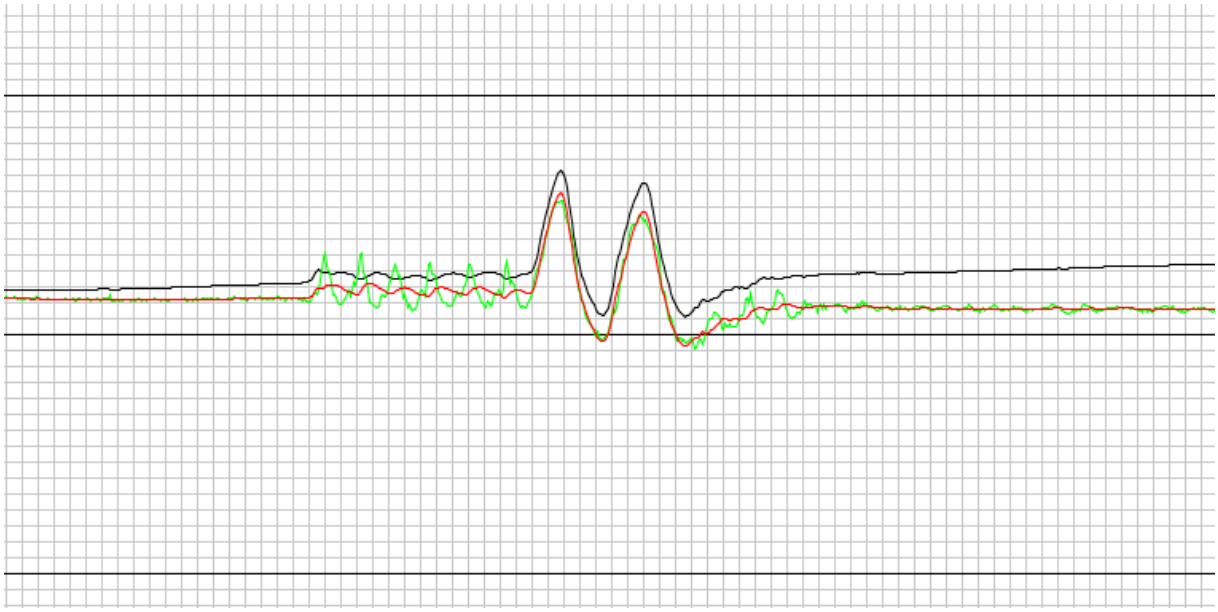
GRÁFICO 2. Comportamento do sensor no eixo X (15 segundos após o início)



4.2. Análise para o Eixo Y

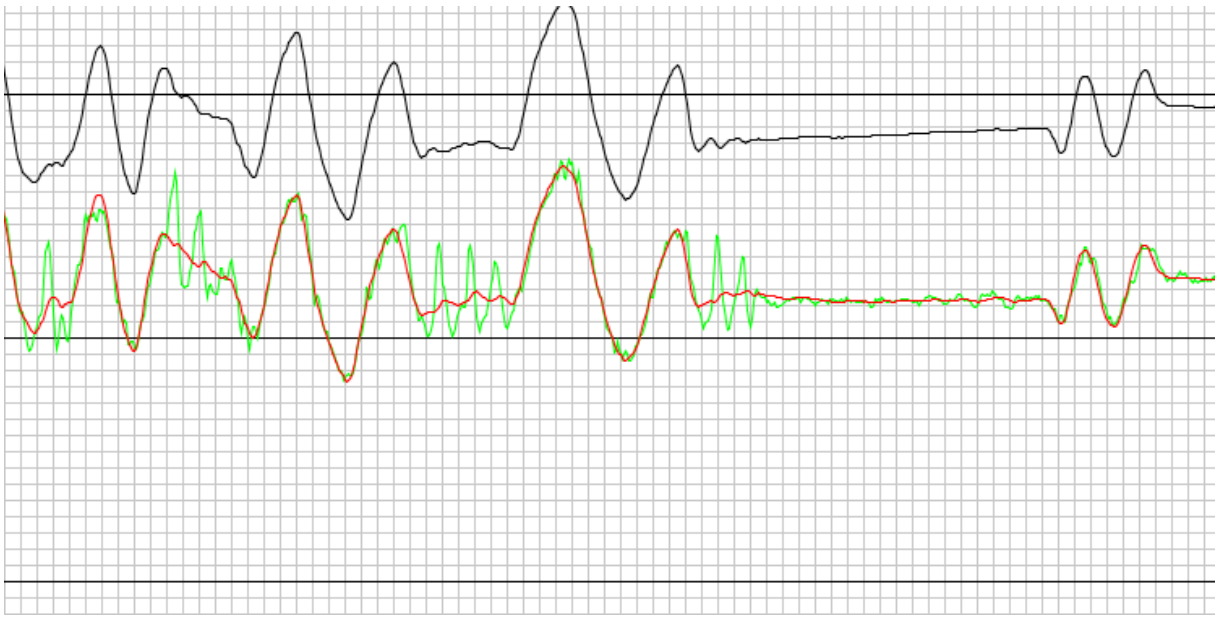
Posteriormente foi analisado o comportamento do sensor no eixo Y. O Gráfico 3 ilustra os instantes iniciais em que se iniciou a coleta de dados para o Eixo Y.

GRÁFICO 3. Comportamento do sensor no eixo Y (instantes iniciais)



No Gráfico 4 temos o comportamento no Eixo Y 15 segundos após o início da coleta de dados para este Eixo.

GRÁFICO 4. Comportamento do sensor no eixo Y (15 segundos após o início)



4.3. Simulação utilizando “Processing”

Além disso, foi utilizado um segundo código no Software “Processing” (código descrito no APÊNDICE - B) que simulava a movimentação do sensor em tempo real. Neste caso, não foi utilizado o Filtro Kalman, podendo-se observar algumas anomalias na movimentação do sensor.

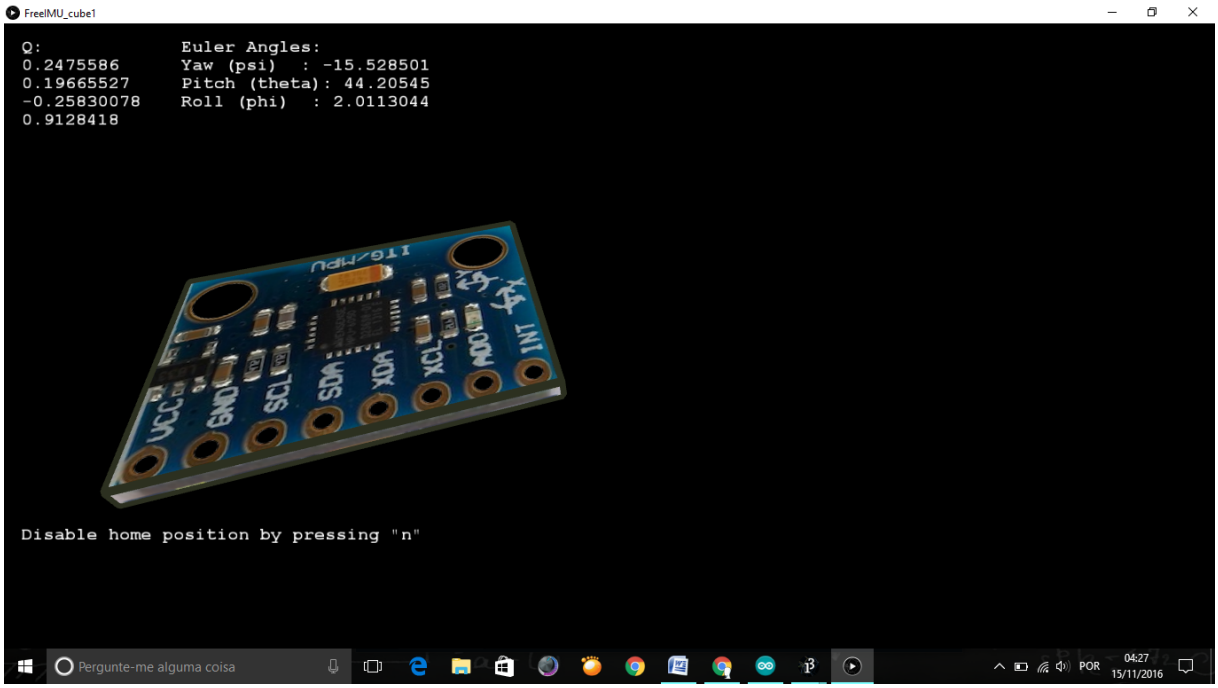


Figura 9: Simulação em tempo real da movimentação do sensor.

5. CONCLUSÕES

Nas análises gráficas o principal objetivo foi analisar e comparar os dados provenientes do giroscópio, acelerômetro e os dados em que foi aplicado o Filtro de Kalman.

Para as análises nos instantes iniciais, tanto em X quanto em Y, observa-se que os todos os dados que modelam o comportamento do sensor ainda estão convergindo. Os dados do Giroscópio (Preto) é o que apresenta maior divergência com relação aos dados do acelerômetro (Verde) e Filtro de Kalman (Vermelho).

Nas análises para os instantes posteriores essa divergência dos dados do giroscópio fica ainda mais eminente, se separando totalmente da modelagem do filtro de Kalman e do acelerômetro.

Apesar de divergir, os dados do giroscópio apresentam muito pouco ruído, tendo a forma muito semelhante à modelagem do Filtro de Kalman (Vermelho), caracterizando-o como um sinal muito preciso, mas ruim em sua exatidão.

Já o acelerômetro (Verde), tem seus dados convergentes com relação à modelagem do Filtro de Kalman, porém apresenta muito ruído, caracterizando-o como sinal pouco preciso, mas melhor em exatidão se comparado ao sinal do giroscópio.

De maneira geral, é perceptível a importância da aplicação de um filtro de correção para os dados obtidos a partir do giroscópio e do acelerômetro, já que ambos apresentam erros que prejudicam tanto na precisão, quanto na exatidão do resultado.

Por fim, na simulação de movimento do sensor sem a utilização do Filtro Kalman foi possível observar um movimento constante, mesmo no caso de estar com o sensor parado, que possivelmente caracterizou a divergência do sinal do giroscópio e os ruídos do sinal do acelerômetro com a ausência de um filtro de correção.

6. PLANO DE TRABALHO E CRONOGRAMA EXECUTADO

6.1. Plano de Trabalho

As principais atividades que deverão ser realizadas para o desenvolvimento do projeto são as seguintes:

1. **Pesquisa bibliográfica**. Deverá ser realizada pesquisa bibliográfica abrangente em livros, periódicos e anais de congressos, visando o estabelecimento de referências sobre o estado da arte do assunto, e sua familiarização com os meios de divulgação científica da área.
2. **Algoritmos de determinação de atitude**. Deverá ser estudado os diferentes métodos e algoritmos utilizados na determinação de atitude, levando em conta os dados obtidos pelos sensores e seus valores teóricos (a partir de modelos matemáticos).
3. **Implementação dos algoritmos**. Os algoritmos estudados deverão ser implementados, através de programação em Linguagem C padronizada para dispositivos embarcados.
4. **Interface para visualização da atitude**. Deverá ser estabelecida uma interface entre o sistema embarcado e um computador para visualização dos resultados

(geração de dados numéricos e de figuras dinâmicas ilustrativas de computação gráfica).

5. **Simulações numéricas e testes experimentais**. A validação da solução para a determinação de atitude será obtida através de testes de simulação numérica, em computador e em processadores.
6. **Preparação de relatórios técnicos e de artigos de divulgação dos resultados das pesquisas, e de relatórios do desenvolvimento da pesquisa**. Toda a evolução da execução do presente projeto deverá ser registrada na forma de apresentação de seminários, preparação de relatórios técnicos e de artigos de divulgação.

6.2. Cronograma

Atividade	***Mês***											
	Jan	***Fev***	***Mar***	***Abr***	***Mai***	***Jun***	***Jul***	***Ago***	***Set***	***Out***	***Nov***	***Dez***
1												
2												
3												
4												
5												
6												

7. REFERÊNCIAS BIBLIOGRÁFICAS

ARDUINO. https://www.arduino.cc/en/Main/arduinoBoardMega2560. Acessado em 15 de abril de 2017.

Castro, João C. V.. Desenvolvimento de um Dispositivo para Determinação de Atitude de Satélites Artificiais Baseado em Magnetômetro de Estado Sólido. Monografia de Graduação em Engenharia de Controle e Automação. Universidade Federal de Ouro Preto, 2006.

Duarte, R. O. et al.. Performance Comparison of Attitude Determination Algorithms Developed to Run in a Microprocessor. 20th International Congress of Mechanical Engineering. Gramado, RS, Brazil. November 15-20, 2009.

Durão, Otávio C. S.; Essado, Marcelo. Programa NanosatC-BR – Desenvolvimento de Cubesats. 1° Convenção Nacional de Radioamadorismo, outubro, 2014.

Garcia, C. et al. Validação de softwares de sistema de determinação de atitude para nanossatélites. 7ª Conferência Sul em Modelagem Computacional, Rio Grande/RS, 2016.

GYROFLY. GYRO200 quad ED. Folha de dados. http://www.gyrofly.com.br. Acessado em 4 de abril de 2017.

Hall, Christopher D. Spacecraft Attitude Dynamics and Control. Notas de aulas. Virginia Tech, 2003.

Lefferts, E. J.; Markley, F. L.; and Shuster, M. D. “Kalman Filtering for Spacecraft Attitude Estimation,” Journal of Guidance, Control, and Dynamics, Vol. 5, No. 5, pp. 417429, 1982.

Maybeck, Peter S. Stochastic Models, Estimation and Control: Volume 1. New York: Academic Press, 1979.

NANOSATC-BR. INPE. Website. Disponível em: http://www.inpe.br/crs/nanosat/index.php Acessado em 2 de abril de 2016.

Shuster, M.D; Oh, S.D. Three-axis attitude determination from vector observations. Journal of Guidance and Control, Vol. 4, No. 1, 1981, pp. 70-77.

ARDUINO. Guide to gyro and accelerometer with arduino including kalman filtering. Disponível em: <http://forum.arduino.cc/index.php/topic,58048.0.html>. Acesso em: 19 ago. 2018.

FILIPEFLOP. Tutorial: acelerômetro mpu6050 com arduino. Disponível em: <https://www.filipeflop.com/blog/tutorial-acelerometro-mpu6050-arduino/>. Acesso em: 19 ago. 2018.

ANEXO A – CÓDIGOS UTILIZADOS NA ANÁLISE GRÁFICA

Código Arduino utilizado para extração de dados

```
#include <Wire.h>
#include <Kalman.h> // Source: https://github.com/TKJElectronics/KalmanFilter

#define RESTRICT_PITCH // Comment out to restrict roll to ±90deg instead - please read: http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf

Kalman kalmanX; // Create the Kalman instances
Kalman kalmanY;

/* IMU Data */
double accX, accY, accZ;
double gyroX, gyroY, gyroZ;
int16_t tempRaw;

double gyroXangle, gyroYangle; // Angle calculate using the gyro only
double compAngleX, compAngleY; // Calculated angle using a complementary filter
double kalAngleX, kalAngleY; // Calculated angle using a Kalman filter

uint32_t timer;
uint8_t i2cData[14]; // Buffer for I2C data

// TODO: Make calibration routine

void setup() {
  Serial.begin(115200);
  Wire.begin();
#if ARDUINO >= 157
  Wire.setClock(400000UL); // Set I2C frequency to 400kHz
#else
```

```
  TWBR = ((F_CPU / 400000UL) - 16) / 2; // Set I2C frequency to 400kHz
#endif

  i2cData[0] = 7; // Set the sample rate to 1000Hz - 8kHz/(7+1) = 1000Hz
  i2cData[1] = 0x00; // Disable FSYNC and set 260 Hz Acc filtering, 256 Hz Gyro filtering, 8 KHz sampling
  i2cData[2] = 0x00; // Set Gyro Full Scale Range to ±250deg/s
  i2cData[3] = 0x00; // Set Accelerometer Full Scale Range to ±2g
  while (i2cWrite(0x19, i2cData, 4, false)); // Write to all four registers at once
  while (i2cWrite(0x6B, 0x01, true)); // PLL with X axis gyroscope reference and disable sleep mode

  while (i2cRead(0x75, i2cData, 1));
  if (i2cData[0] != 0x68) { // Read "WHO_AM_I" register
    Serial.print(F("Error reading sensor"));
    while (1);
  }

  delay(100); // Wait for sensor to stabilize

  /* Set kalman and gyro starting angle */
  while (i2cRead(0x3B, i2cData, 6));
  accX = (int16_t)((i2cData[0] << 8) | i2cData[1]);
  accY = (int16_t)((i2cData[2] << 8) | i2cData[3]);
  accZ = (int16_t)((i2cData[4] << 8) | i2cData[5]);

  // Source: http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf eq. 25 and eq. 26
  // atan2 outputs the value of -π to π (radians) - see http://en.wikipedia.org/wiki/Atan2
  // It is then converted from radians to degrees
#ifdef RESTRICT_PITCH // Eq. 25 and 26
  double roll  = atan2(accY, accZ) * RAD_TO_DEG;
  double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) * RAD_TO_DEG;
#else // Eq. 28 and 29
```

```
  double roll  = atan(accY / sqrt(accX * accX + accZ * accZ)) * RAD_TO_DEG;
  double pitch = atan2(-accX, accZ) * RAD_TO_DEG;
#endif

  kalmanX.setAngle(roll); // Set starting angle
  kalmanY.setAngle(pitch);
  gyroXangle = roll;
  gyroYangle = pitch;
  compAngleX = roll;
  compAngleY = pitch;

  timer = micros();
}

void loop() {
  /* Update all the values */
  while (i2cRead(0x3B, i2cData, 14));
  accX = (int16_t)((i2cData[0] << 8) | i2cData[1]);
  accY = (int16_t)((i2cData[2] << 8) | i2cData[3]);
  accZ = (int16_t)((i2cData[4] << 8) | i2cData[5]);
  tempRaw = (int16_t)((i2cData[6] << 8) | i2cData[7]);
  gyroX = (int16_t)((i2cData[8] << 8) | i2cData[9]);
  gyroY = (int16_t)((i2cData[10] << 8) | i2cData[11]);
  gyroZ = (int16_t)((i2cData[12] << 8) | i2cData[13]);;

  double dt = (double)(micros() - timer) / 1000000; // Calculate delta time
  timer = micros();

  // Source: http://www.freescale.com/files/sensors/doc/app_note/AN3461.pdf eq. 25
and eq. 26
  // atan2 outputs the value of -π to π (radians) - see http://en.wikipedia.org/wiki/Atan2
  // It is then converted from radians to degrees
#ifdef RESTRICT_PITCH // Eq. 25 and 26
  double roll  = atan2(accY, accZ) * RAD_TO_DEG;
```

```
  double pitch = atan(-accX / sqrt(accY * accY + accZ * accZ)) * RAD_TO_DEG;
#else // Eq. 28 and 29
  double roll  = atan(accY / sqrt(accX * accX + accZ * accZ)) * RAD_TO_DEG;
  double pitch = atan2(-accX, accZ) * RAD_TO_DEG;
#endif

  double gyroXrate = gyroX / 131.0; // Convert to deg/s
  double gyroYrate = gyroY / 131.0; // Convert to deg/s

#ifdef RESTRICT_PITCH
  // This fixes the transition problem when the accelerometer angle jumps between -180 and 180 degrees
  if ((roll < -90 && kalAngleX > 90) || (roll > 90 && kalAngleX < -90)) {
    kalmanX.setAngle(roll);
    compAngleX = roll;
    kalAngleX = roll;
    gyroXangle = roll;
  } else
    kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt); // Calculate the angle using a Kalman filter

  if (abs(kalAngleX) > 90)
    gyroYrate = -gyroYrate; // Invert rate, so it fits the restriced accelerometer reading
  kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt);
#else
  // This fixes the transition problem when the accelerometer angle jumps between -180 and 180 degrees
  if ((pitch < -90 && kalAngleY > 90) || (pitch > 90 && kalAngleY < -90)) {
    kalmanY.setAngle(pitch);
    compAngleY = pitch;
    kalAngleY = pitch;
    gyroYangle = pitch;
  } else
```

```
    kalAngleY = kalmanY.getAngle(pitch, gyroYrate, dt); // Calculate the angle using a Kalman filter

  if (abs(kalAngleY) > 90)
    gyroXrate = -gyroXrate; // Invert rate, so it fits the restriced accelerometer reading
  kalAngleX = kalmanX.getAngle(roll, gyroXrate, dt); // Calculate the angle using a Kalman filter
#endif

  gyroXangle += gyroXrate * dt; // Calculate gyro angle without any filter
  gyroYangle += gyroYrate * dt;
  //gyroXangle += kalmanX.getRate() * dt; // Calculate gyro angle using the unbiased rate
  //gyroYangle += kalmanY.getRate() * dt;

  compAngleX = 0.93 * (compAngleX + gyroXrate * dt) + 0.07 * roll; // Calculate the angle using a Complimentary filter
  compAngleY = 0.93 * (compAngleY + gyroYrate * dt) + 0.07 * pitch;

  // Reset the gyro angle when it has drifted too much
  if (gyroXangle < -180 || gyroXangle > 180)
    gyroXangle = kalAngleX;
  if (gyroYangle < -180 || gyroYangle > 180)
    gyroYangle = kalAngleY;

  /* Print Data */
#if 0 // Set to 1 to activate
  Serial.print(accX); Serial.print("\t");
  Serial.print(accY); Serial.print("\t");
  Serial.print(accZ); Serial.print("\t");

  Serial.print(gyroX); Serial.print("\t");
  Serial.print(gyroY); Serial.print("\t");
  Serial.print(gyroZ); Serial.print("\t");
```

```
  Serial.print("\t");
#endif

  Serial.print(roll); Serial.print("\t");
  Serial.print(gyroXangle); Serial.print("\t");
  Serial.print(compAngleX); Serial.print("\t");
  Serial.print(kalAngleX); Serial.print("\t");

  Serial.print("\t");

  Serial.print(pitch); Serial.print("\t");
  Serial.print(gyroYangle); Serial.print("\t");
  Serial.print(compAngleY); Serial.print("\t");
  Serial.print(kalAngleY); Serial.print("\t");

#if 0 // Set to 1 to print the temperature
  Serial.print("\t");

  double temperature = (double)tempRaw / 340.0 + 36.53;
  Serial.print(temperature); Serial.print("\t");
#endif

  Serial.print("\r\n");
  delay(2);
}
```

Código .cpp para o Filtro Kalman

```
#include "Kalman.h"

Kalman::Kalman() {
    /* We will set the variables like so, these can also be tuned by the user */
    Q_angle = 0.001f;
    Q_bias = 0.003f;
    R_measure = 0.03f;

    angle = 0.0f; // Reset the angle
    bias = 0.0f; // Reset bias

    P[0][0] = 0.0f; // Since we assume that the bias is 0 and we know the starting angle (use setAngle), the error covariance matrix is set like so - see: http://en.wikipedia.org/wiki/Kalman_filter#Example_application.2C_technical
    P[0][1] = 0.0f;
    P[1][0] = 0.0f;
    P[1][1] = 0.0f;
};

// The angle should be in degrees and the rate should be in degrees per second and the delta time in seconds
float Kalman::getAngle(float newAngle, float newRate, float dt) {
    // KasBot V2  -  Kalman filter module - http://www.x-firm.com/?page_id=145
    // Modified by Kristian Lauszus
    // See my blog post for more information: http://blog.tkjelectronics.dk/2012/09/a-practical-approach-to-kalman-filter-and-how-to-implement-it

    // Discrete Kalman filter time update equations - Time Update ("Predict")
    // Update xhat - Project the state ahead
    /* Step 1 */
    rate = newRate - bias;
    angle += dt * rate;
```

```
    // Update estimation error covariance - Project the error covariance ahead
    /* Step 2 */
    P[0][0] += dt * (dt*P[1][1] - P[0][1] - P[1][0] + Q_angle);
    P[0][1] -= dt * P[1][1];
    P[1][0] -= dt * P[1][1];
    P[1][1] += Q_bias * dt;

    // Discrete Kalman filter measurement update equations - Measurement Update ("Correct")
    // Calculate Kalman gain - Compute the Kalman gain
    /* Step 4 */
    float S = P[0][0] + R_measure; // Estimate error
    /* Step 5 */
    float K[2]; // Kalman gain - This is a 2x1 vector
    K[0] = P[0][0] / S;
    K[1] = P[1][0] / S;

    // Calculate angle and bias - Update estimate with measurement zk (newAngle)
    /* Step 3 */
    float y = newAngle - angle; // Angle difference
    /* Step 6 */
    angle += K[0] * y;
    bias += K[1] * y;

    // Calculate estimation error covariance - Update the error covariance
    /* Step 7 */
    float P00_temp = P[0][0];
    float P01_temp = P[0][1];

    P[0][0] -= K[0] * P00_temp;
    P[0][1] -= K[0] * P01_temp;
    P[1][0] -= K[1] * P00_temp;
    P[1][1] -= K[1] * P01_temp;
```

```
    return angle;
};

void Kalman::setAngle(float angle) { this->angle = angle; }; // Used to set angle, this should be set as the starting angle
float Kalman::getRate() { return this->rate; }; // Return the unbiased rate

/* These are used to tune the Kalman filter */
void Kalman::setQangle(float Q_angle) { this->Q_angle = Q_angle; };
void Kalman::setQbias(float Q_bias) { this->Q_bias = Q_bias; };
void Kalman::setRmeasure(float R_measure) { this->R_measure = R_measure; };

float Kalman::getQangle() { return this->Q_angle; };
float Kalman::getQbias() { return this->Q_bias; };
float Kalman::getRmeasure() { return this->R_measure; };
```

Código Processing ultilizado para os gráficos

```
void drawAxisX() {
  // Draw gyro x - axis
  noFill();
  stroke(0, 0, 0); // Black
  // Redraw everything
  beginShape();
  vertex(0, gyroX[0]);
  for (int i = 1; i < gyroX.length; i++) {
    if ((gyroX[i] < height/4 && gyroX[i - 1] > height/4*3) || (gyroX[i] > height/4*3 && gyroX[i
- 1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, gyroX[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < gyroX.length;i++)
    gyroX[i-1] = gyroX[i];

  // Draw acceleromter x-axis
  noFill();
  stroke(0, 255, 0); // Green
  // Redraw everything
  beginShape();
  vertex(0, accX[0]);
  for (int i = 1; i < accX.length; i++) {
    if ((accX[i] < height/4 && accX[i - 1] > height/4*3) || (accX[i] > height/4*3 && accX[i -
1] < height/4)) {
      endShape();
      beginShape();
```

```
    }
    vertex(i, accX[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < accX.length;i++)
    accX[i-1] = accX[i];

  /* Draw complementary filter x-axis
  noFill();
  stroke(0, 0, 255); // Blue
  // Redraw everything
  beginShape();
  vertex(0, compX[0]);
  for (int i = 1; i < compX.length; i++) {
    if ((compX[i] < height/4 && compX[i - 1] > height/4*3) || (compX[i] > height/4*3 &&
compX[i - 1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, compX[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < compX.length; i++)
    compX[i-1] = compX[i]; */

  // Draw kalman filter x-axis
  noFill();
  stroke(255, 0, 0);// Red
  // Redraw everything
  beginShape();
```

```
  vertex(0, kalmanX[0]);
  for (int i = 1; i < kalmanX.length; i++) {
    if ((kalmanX[i] < height/4 && kalmanX[i - 1] > height/4*3) || (kalmanX[i] > height/4*3
&& kalmanX[i - 1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, kalmanX[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < kalmanX.length; i++)
    kalmanX[i-1] = kalmanX[i];
}

void drawAxisY() {
  // Draw gyro y-axis
  noFill();
  stroke(0, 0, 0); // Black
  // Redraw everything
  beginShape();
  vertex(0, gyroY[0]);
  for (int i = 1; i < gyroY.length; i++) {
    if ((gyroY[i] < height/4 && gyroY[i - 1] > height/4*3) || (gyroY[i] > height/4*3 && gyroY[i
- 1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, gyroY[i]);
  }
  endShape();

  // Put all data one array back
```

```
  for (int i = 1; i < gyroY.length;i++)
   gyroY[i-1] = gyroY[i];

  // Draw acceleromter y-axis
  noFill();
  stroke(0, 255, 0); // Green
  // Redraw everything
  beginShape();
  vertex(0, accY[0]);
  for (int i = 1; i < accY.length; i++) {
    if ((accY[i] < height/4 && accY[i - 1] > height/4*3) || (accY[i] > height/4*3 && accY[i -
1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, accY[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < accY.length;i++)
    accY[i-1] = accY[i];

  /* Draw complementary filter y-axis
  noFill();
  stroke(124, 252, 0); // Lawn Green
  // Redraw everything
  beginShape();
  vertex(0, compY[0]);
  for (int i = 1; i < compY.length; i++) {
    if ((compY[i] < height/4 && compY[i - 1] > height/4*3) || (compY[i] > height/4*3 &&
compY[i - 1] < height/4)) {
      endShape();
      beginShape();
```

```
    }
    vertex(i, compY[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i < compY.length;i++)
    compY[i-1] = compY[i];*/

  // Draw kalman filter y-axis
  noFill();
  stroke(255, 0, 0); // Red
  // Redraw everything
  beginShape();
  vertex(0, kalmanY[0]);
  for (int i = 1; i < kalmanY.length; i++) {
    if ((kalmanY[i] < height/4 && kalmanY[i - 1] > height/4*3) || (kalmanY[i] > height/4*3
&& kalmanY[i - 1] < height/4)) {
      endShape();
      beginShape();
    }
    vertex(i, kalmanY[i]);
  }
  endShape();

  // Put all data one array back
  for (int i = 1; i<kalmanY.length;i++)
    kalmanY[i-1] = kalmanY[i];
}
```

ANEXO B – CÓDIGO PROCESSING UTILIZADO PARA SIMULAÇÃO

```
Serial myPort;  // Create object from Serial class

//final String serialPort = "/dev/ttyUSB9"; // replace this with your serial port. On windows you will need something like "COM1".
final String serialPort = "COM3"; // replace this with your serial port. On windows you will need something like "COM1".


float [] q = new float [4];
float [] hq = null;
float [] Euler = new float [3]; // psi, theta, phi

int lf = 10; // 10 is '\n' in ASCII
byte[] inBuffer = new byte[22]; // this is the number of chars on each line from the Arduino (including /r/n)

PFont font;
//final int VIEW_SIZE_X = 1024, VIEW_SIZE_Y = 768;
final int VIEW_SIZE_X = 800, VIEW_SIZE_Y = 600;

PImage topside,downside,frontside,rightside;

void setup()
{
  size(100, 100, P3D);
  textureMode(NORMAL);
  fill(255);
  stroke(color(44,48,32));

  myPort = new Serial(this, serialPort, 115200);
```

```
  // The font must be located in the sketch's "data" directory to load successfully
  font = loadFont("CourierNew36.vlw");


  // Loading the textures to the cube
  // The png files alow to put the board holes so can increase  realism

  topside = loadImage("MPU6050 A.png");//Top Side
  downside = loadImage("MPU6050 B.png");//Botm side
  frontside = loadImage("MPU6050 E.png"); //Wide side
  rightside = loadImage("MPU6050 F.png");// Narrow side

  delay(100);
  myPort.clear();
  myPort.write("1");
}


float decodeFloat(String inString) {
  byte [] inData = new byte[4];

  if(inString.length() == 8) {
    inData[0] = (byte) unhex(inString.substring(0, 2));
    inData[1] = (byte) unhex(inString.substring(2, 4));
    inData[2] = (byte) unhex(inString.substring(4, 6));
    inData[3] = (byte) unhex(inString.substring(6, 8));
  }

  int intbits = (inData[3] << 24) | ((inData[2] & 0xff) << 16) | ((inData[1] & 0xff) << 8) |
(inData[0] & 0xff);
  return Float.intBitsToFloat(intbits);
}
```

```
void readQ() {
  if(myPort.available() >= 18) {
    String inputString = myPort.readStringUntil('\n');
    //print(inputString);
    if (inputString != null && inputString.length() > 0) {
      String [] inputStringArr = split(inputString, ",");
      if(inputStringArr.length >= 5) { // q1,q2,q3,q4,\r\n so we have 5 elements
        q[0] = decodeFloat(inputStringArr[0]);
        q[1] = decodeFloat(inputStringArr[1]);
        q[2] = decodeFloat(inputStringArr[2]);
        q[3] = decodeFloat(inputStringArr[3]);
      }
    }
  }
}

/*
From
* Texture Cube
* by Dave Bollinger.
I only Added multiple sides textured. AHV Ago 2013
*/

void topboard(PImage imag) {
  beginShape(QUADS);
  texture(imag);
  // -Y "top" face
  vertex(-20, -1, -15, 0, 0);
  vertex( 20, -1, -15, 1, 0);
  vertex( 20, -1,  15, 1, 1);
  vertex(-20, -1,  15, 0, 1);

  endShape();
}
```

```
void botomboard(PImage imag) {
  beginShape(QUADS);
  texture(imag);

  // +Y "bottom" face
  vertex(-20,  1,  15, 0, 0);
  vertex( 20,  1,  15, 1, 0);
  vertex( 20,  1, -15, 1, 1);
  vertex(-20,  1, -15, 0, 1);

  endShape();
}

void sideboarda(PImage imag) {
  beginShape(QUADS);
  texture(imag);

  // +Z "front" face
  vertex(-20, -1,  15, 0, 0);
  vertex( 20, -1,  15, 1, 0);
  vertex( 20,  1,  15, 1, 1);
  vertex(-20,  1,  15, 0, 1);

  // -Z "back" face
  vertex( 20, -1, -15, 0, 0);
  vertex(-20, -1, -15, 1, 0);
  vertex(-20,  1, -15, 1, 1);
  vertex( 20,  1, -15, 0, 1);


  endShape();
}
```

```
void sideboardb(PImage imag) {
  beginShape(QUADS);
  texture(imag);

   // +X "right" face
  vertex( 20, -1,  15, 0, 0);
  vertex( 20, -1, -15, 1, 0);
  vertex( 20,  1, -15, 1, 1);
  vertex( 20,  1,  15, 0, 1);

  // -X "left" face
  vertex(-20, -1, -15, 0, 0);
  vertex(-20, -1,  15, 1, 0);
  vertex(-20,  1,  15, 1, 1);
  vertex(-20,  1, -15, 0, 1);

  endShape();
}

void drawCube() {
  pushMatrix();
    translate(VIEW_SIZE_X/2, VIEW_SIZE_Y/2 + 50, 0);
    //scale(5,5,5);
    scale(10);

    // a demonstration of the following is at
    //          http://www.varesano.net/blog/fabio/ahrs-sensor-fusion-orientation-filter-3d-graphical-rotating-cube
    rotateZ(-Euler[2]);
    rotateX(-Euler[1]);
    rotateY(-Euler[0]);

    topboard(topside);
    botomboard(downside);
```

```
    sideboarda(frontside);
    sideboardb(rightside);

  popMatrix();
}

void draw() {
  background(#000000);
  //fill(#ffffff);

  readQ();

  if(hq != null) { // use home quaternion
    quaternionToEuler(quatProd(hq, q), Euler);
    text("Disable home position by pressing \"n\"", 20, VIEW_SIZE_Y - 30);
  }
  else {
    quaternionToEuler(q, Euler);
    text("Point FreeIMU's X axis to your monitor then press \"h\"", 20, VIEW_SIZE_Y -
30);
  }

  textFont(font, 20);
  textAlign(LEFT, TOP);
  text("Q:\n" + q[0] + "\n" + q[1] + "\n" + q[2] + "\n" + q[3], 20, 20);
  text("Euler Angles:\nYaw (psi)   : " + degrees(Euler[0]) + "\nPitch (theta): " +
degrees(Euler[1]) + "\nRoll (phi)  : " + degrees(Euler[2]), 200, 20);

  drawCube();
}

void keyPressed() {
  if(key == 'h') {
    println("pressed h");
```

```
    // set hq the home quaternion as the quatnion conjugate coming from the sensor fusion
    hq = quatConjugate(q);

  }
  else if(key == 'n') {
    println("pressed n");
    hq = null;
  }
}

// See Sebastian O.H. Madwick report
// "An efficient orientation filter for inertial and intertial/magnetic sensor arrays" Chapter 2 Quaternion representation

void quaternionToEuler(float [] q, float [] euler) {
  euler[0] = atan2(2 * q[1] * q[2] - 2 * q[0] * q[3], 2 * q[0]*q[0] + 2 * q[1] * q[1] - 1); // psi
  euler[1] = -asin(2 * q[1] * q[3] + 2 * q[0] * q[2]); // theta
  euler[2] = atan2(2 * q[2] * q[3] - 2 * q[0] * q[1], 2 * q[0] * q[0] + 2 * q[3] * q[3] - 1); // phi
}

float [] quatProd(float [] a, float [] b) {
  float [] q = new float[4];

  q[0] = a[0] * b[0] - a[1] * b[1] - a[2] * b[2] - a[3] * b[3];
  q[1] = a[0] * b[1] + a[1] * b[0] + a[2] * b[3] - a[3] * b[2];
  q[2] = a[0] * b[2] - a[1] * b[3] + a[2] * b[0] + a[3] * b[1];
  q[3] = a[0] * b[3] + a[1] * b[2] - a[2] * b[1] + a[3] * b[0];

  return q;
}

// returns a quaternion from an axis angle representation
```

```
float [] quatAxisAngle(float [] axis, float angle) {
  float [] q = new float[4];

  float halfAngle = angle / 2.0;
  float sinHalfAngle = sin(halfAngle);
  q[0] = cos(halfAngle);
  q[1] = -axis[0] * sinHalfAngle;
  q[2] = -axis[1] * sinHalfAngle;
  q[3] = -axis[2] * sinHalfAngle;

  return q;
}

// return the quaternion conjugate of quat
float [] quatConjugate(float [] quat) {
  float [] conj = new float[4];

  conj[0] = quat[0];
  conj[1] = -quat[1];
  conj[2] = -quat[2];
  conj[3] = -quat[3];

  return conj;
}
```